Accurate and Resource-Efficient Monitoring for Future Networks

Gioacchino Tangari

A dissertation submitted in fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

Department of Electronic and Electrical Engineering
University College London

I, Gioacchino Tangari, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Monitoring functionality is a key component of any network management system. It is essential for profiling network resource usage, detecting attacks, and capturing the performance of a multitude of services using the network. Traditional monitoring solutions operate on long timescales producing periodic reports, which are mostly used for manual and infrequent network management tasks. However, these practices have been recently questioned by the advent of Software Defined Networking (SDN). By empowering management applications with the right tools to perform automatic, frequent, and fine-grained network reconfigurations, SDN has made these applications more dependent than before on the accuracy and timeliness of monitoring reports. As a result, monitoring systems are required to collect considerable amounts of heterogeneous measurement data, process them in real-time, and expose the resulting knowledge in short timescales to network decision-making processes. Satisfying these requirements is extremely challenging given today's larger network scales, massive and dynamic traffic volumes, and the stringent constraints on time availability and hardware resources. This PhD thesis tackles this important challenge by investigating how an accurate and resource-efficient monitoring function can be realised in the context of future, software-defined networks. Novel monitoring methodologies, designs, and frameworks are provided in this thesis, which scale with increasing network sizes and automatically adjust to changes in the operating conditions. These achieve the goal of efficient measurement collection and reporting, lightweight measurement-data processing, and timely monitoring knowledge delivery.

Impact Statement

This PhD thesis contributes to the research advances in software-defined networks by investigating key enablers to support the design of advanced, programmable and automated network management approaches.

Specifically, this thesis proposes and develops efficient monitoring designs, mechanisms, and frameworks that allow to generate accurate and frequent monitoring reports of fine granularity concerning a variety of network events and emerging conditions such as attacks and network performance bottlenecks. The monitoring advances presented in this thesis allow network management processes to reconfigure the network frequently and at high levels of granularity. As a result, the technical advancements achieved by this thesis are key enablers for (i) recovering from performance issues in short times and with very precise and selective countermeasures, (ii) addressing attacks immediately, before users and services are adversely impacted, and (iii) more efficient use of network resources to reduce operational and infrastructure costs. Such innovations enable the introduction of a variety of new services that are extremely time-sensitive, demanding in their security and privacy requirements, and heavy in resource consumption, with key examples being virtual reality, holographic communications, and cloud gaming.

The contributions presented by this thesis have been published in high-quality journals and presented in competitive conferences. This has contributed to extend the technological developments in the areas of network monitoring and software-defined networking, but also influence future advancements in these domains by transferring relevant knowledge to a wide research community.

Acknowledgements

I am thankful to all the people whose support and advice made this work possible. First of all, I would like to express my gratitude to my supervisor, Prof. George Pavlou, for giving me the opportunity to pursue this research, for his advice and encouragement, and for the freedom I had in my research work.

I am also extremely grateful to Dr. Marinos Charalambides and Dr. Daphne Tuncer for the constant guidance and support I received from them, the help and the insightful discussions that allowed me to shape my PhD research, and all the inspiration they offered me in these 4 years.

During these years I came across a very long list of advisors and inspiring people, to whom I am sincerely thankful. In particular, I would like to thank my mentors Dr. Diego Perino and Dr. Alessandro Finamore from Telefonica Research, all current and former members of my research group at UCL, and my thesis examiners Dr. Dimitrios Pezaros and Dr. Shi Zhou.

I also received a great amount of encouragement from many friends with whom I had the pleasure to share my time in London, Oxford, and in my hometown.

Lastly, and most importantly, I am deeply grateful to Clara, my mum Angela, my dad Giuseppe and my sister Arianna. They offered me constant encouragement and support throughout these years and they have been a major source of inspiration for all my achievements.

Contents

List of Figures

List of Tables

Table of Acronyms

AIMD	Additive-Increase/Multiplicative-Decrease
API	Application Programming Interface
CPU	Central Processing Unit
DDoS	Distributed Denial-Of-Service
DPDK	Data Plane Development Kit
EWMA	Exponentially Weighted Moving Average
HH	Heavy Hitter
ISP	Internet Service Provider
LB	Load Balancing
LM	Local Manager
MA	Management Application
MM	Monitoring Module
NIC	Network Interface Card
PAM	Prediction-based Adaptive Monitoring
PISA	Protocol Independent Switch Architecture
RMSE	Root Mean Square Error
RSS	Receive Side Scaling
SAM	Self-tuning Adaptive Monitoring
SDN	Software-Defined Networking
SLA	Service-Level Agreement
SRAM	Static Random Access Memory
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
TAM	Threshold-based Adaptive Monitoring
TCAM	Ternary Content-Addressable Memory
TCP	Transmission Control Protocol
TLB	Translation Look-aside Buffer
UDP	User Datagram Protocol
WAN	Wide Area Network

Publications

- **G.Tangari**, A. Finamore, D. Perino, M. Charalambides, G. Pavlou. "Tackling Mobile Traffic Critical Path Analysis with Active and Passive Measurements". To appear in *Network Traffic Measurements and Analysis* (TMA) Conference, Jun. 2019
- **G. Tangari**, M. Charalambides, D. Tuncer, Y.Qi, G. Pavlou. "Self-Adaptive Decentralized Monitoring in Software-Defined Networks". In *IEEE Transactions of Network and Service Management* (TNSM), Special issue on Novel Techniques for Managing Softwarized Networks, vol.15, issue 4, 2018 *[Impact Factor – 3.286]*
- D. Tuncer, M. Charalambides, **G.Tangari**, G. Pavlou. "A Northbound interface for Software-Based networks". In *14th International Conference on Network and Service Management* (CNSM), Nov. 2018 *[Acceptance Rate – 15.9%]*
- **G. Tangari**, D. Tuncer, M. Charalambides, G. Pavlou. "Adaptive Traffic Monitoring for Software Dataplanes". In *13th International Conference on Network and Service Management* (CNSM), Nov. 2017 *[Acceptance Rate – 17.6%]*
- **G. Tangari**, M. Charalambides, D. Tuncer, G. Pavlou. "Decentralized Monitoring for Large-Scale Software-Defined Networks". In *IFIP/IEEE International Symposium on Integrated Network Management* (IM), May 2017 *[Acceptance Rate – 28.5%]*
- **G. Tangari**, M. Charalambides, D. Tuncer, G. Pavlou. "Decentralized Solutions for Monitoring Large-Scale Software-Defined Networks". In *Autonomous Infrastructure Management and Security* (AIMS) PhD workshop, Jun. 2016

Chapter 1

Introduction

The demand of modern Internet applications for high availability and service quality, as well as the tremendous growth in the number of Internet users and traffic volumes, have made the management of networks a complex task. To manage today's networks a combination of various processes needs to be deployed, ranging from traffic engineering and network planning, to performance diagnosis and attack prevention. These processes rely on accurate monitoring information, which provides vital input to effectively decide on new network configurations and to warrant precision when troubleshooting failures or detecting anomalies.

The generalised shift towards *Software-Defined Networking* (SDN) is deeply affecting the way networks are managed [1]. Traditional management processes restrict network operators to manual and infrequent reconfigurations using limited sets of low-level, device-specific commands. In contrast, future networks can rely on the principles of SDN and the associated technological novelty to enable management applications that reconfigure the network automatically, frequently [2] [3] and at a fine granularity [4] [5]. These novel capabilities pose strict requirements in terms of monitoring information, which cannot be satisfied by traditional monitoring approaches. Standard practices based on SNMP logs, packet sampling [6], and passive traffic trace analysis [7] are limited by their coarse report granularity and frequency, and thus fail in supporting automated and highly-reactive management applications [8].

To keep pace with the evolution of network management technologies, recent research has proposed a number of new monitoring solutions. These leverage novel enablers at the data and control planes to extract detailed views of the network state in real-time. However, it is still an open issue how monitoring systems can efficiently cope with massive and dynamic amounts of traffic, large network scales, and stringent limitations on time and hardware resources.

1.1 Motivation

The ultimate goal of network monitoring systems is to expose a variety of events and emerging conditions concerning potential attacks, resource usage, and changing traffic patterns [9] [10]. Detecting attacks on time is essential for securing the network, while accurate reports of the network

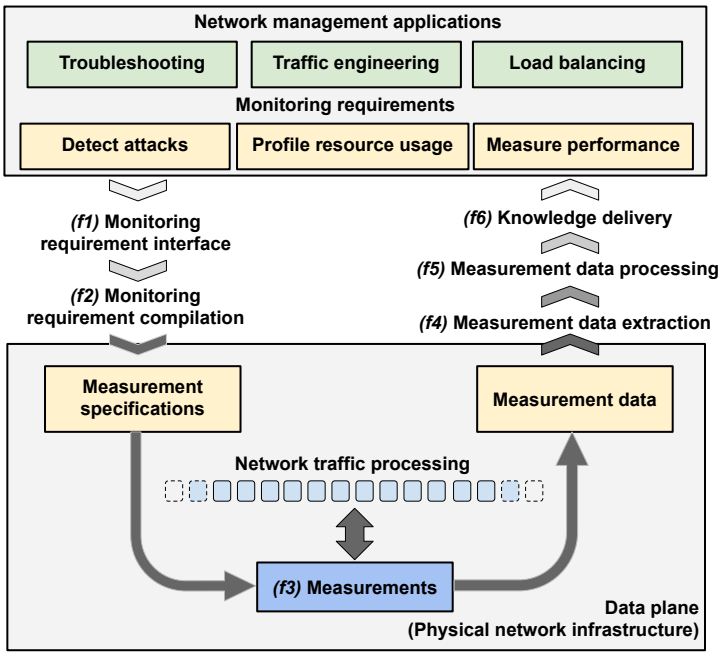


Figure 1.1: Illustration of main network monitoring functionalities

state and traffic characteristics allow the diagnosis of performance problems and are useful for reducing network operational costs [11]. From a logical perspective, a monitoring system combines a diverse set of functionalities, which operate on two main workflows as shown in Figure 1.1. The first starts from the monitoring requirements of network management applications and follows a top-down direction. The requirements are registered through a monitoring interface (f_1), based on a set of APIs or a monitoring-specific language. Monitoring commands are then translated (f_2) into measurement specifications that are deployed in the dataplane (f_3), where network traffic is processed. Conversely, the second workflow operates on the measurement data collected at the dataplane, in a bottom-up fashion. Specifically, the measurement data is acquired from the dataplane (f_4) and processed, e.g., aggregated and correlated, to form relevant monitoring knowledge (f_5), which is delivered to applications (f_6).

The recent development of software-defined networking has been questioning the design and implementation of the aforementioned functionalities. On the one hand, the advent of SDN has offered new enablers for the design of measurements. In particular, novel technologies, including protocols, hardware architectures, and software libraries have allowed a programmable dataplane to run on commodity switches (e.g., OpenFlow [12]), hosts (e.g., software switching [13] [14], packet capture libraries [15] [16]), and new protocol-independent switch architectures [17] [18]. Monitoring can benefit from the improved programmability to execute more elaborate measurement tasks in real-time within dataplanes, and to collect highly-configurable sets of measurement data.

On the other hand, by enabling automatic network reconfigurations [1] [19], SDN has allowed management processes to apply new settings more frequently and at a finer granularity, which makes

them extremely sensitive to the timeliness and precision of monitoring reports [8]. Inaccurate monitoring information can lead to chains of wrong reactions, e.g., in the case of automated management applications [11], while long monitoring delays can leave short-lived network events unhandled and configuration *control-loops* operating at tight intervals with outdated knowledge [20]. As a result of this evolution, monitoring systems are nowadays required to (i) extract larger amounts of heterogeneous and fine-grained measurement data (e.g., at the granularity of flowlets [5] or even individual packets [9]), (ii) process such data in real-time, and (iii) swiftly expose (e.g., in the order of milliseconds [8]) the resulting knowledge to decision-making processes.

Satisfying these requirements is a hard task as monitoring systems must face stringent resource constraints (e.g., limited memory and/or computational power) and limited time availability (e.g., no more than few tens of nanoseconds to process a packet at a 10Gbps traffic rate). Specifically, in order to achieve the goal of timely and accurate monitoring reports, the following efficiency issues should be overcome.

A1) Use of dataplane resources for measurements Monitoring systems can face dynamic workloads at the dataplane, due to changes in the network traffic (e.g., increase of the traffic rate or number of concurrent traffic flows) or spikes in the demand for monitoring information. To avoid performance bottlenecks, network operators can choose between (i) providing measurements with more hardware resources than actually needed (overprovisioning), or (ii) allowing for only a limited set of measurement operations in the dataplane to keep resource utilisation low. Intuitively, both solutions lead to *inefficient* use of the available resources.

A2) Measurement result acquisition from the dataplane Retrieving measurement information from the dataplane can be tedious due to the limited capacity of a dataplane to transfer information to network controllers and managers [21] [8]. A holistic approach entailing continuous pull/push of all measurement data will inevitably saturate dataplane resources [22] [21], while relaxed synchronisations can result into imprecise network views [20] [23].

A3) Processing of measurement results Not only should measurement data be carefully extracted from the dataplane, but also processed in real-time to discover a variety of events such as network attacks, congestions, or traffic load imbalance. Dealing with large volumes of data to be *evaluated*, *aggregated*, and *correlated*, is prone to incurring prohibitive costs in terms of computational resources as well as processing latencies.

A4) Monitoring information delivery With networks growing in scale and management processes able to operate on network-wide views [2] [24], decisions on network reconfigurations might be taken at any distance from where the measurement data is produced. State-of-the-art monitoring systems [8] [9] [10] rely on a centralised core entity to handle all data and deliver knowledge. Such an approach can be time-inefficient as it inflates reconfiguration control-loop delays when decisions-making happens close to where measurements are collected.

This PhD thesis addresses these issues with the overarching goal *of an efficient and accurate*

monitoring function, which can satisfy the demanding requirements posed by software-defined networking and cope with the evolution of networks towards higher utilisation, scale and traffic volumes.

1.2 Methodology and contributions

Although the functionalities in Figure 1.1 can be considered as general attributes of any monitoring system, their design and implementation can vary substantially, as a result of growing network complexity and the increased flexibility introduced by SDN. Overall, the design and implementation of monitoring systems reflect (i) the type of monitoring data collected, and (ii) the source of the measurement data.

Regarding the type of monitoring data collected, the position of this thesis is not to focus on individual metrics or statistics of interest. A vast amount of research has already addressed how specific measurements can be deployed using the new dataplane enablers of software-defined networking [25] [26] [27], or how the new measurement primitives can outpace traditional tools such as Netflow, SNMP, and Tcpdump [8] [9]. In contrast, this thesis investigates monitoring designs where multiple, diverse *measurement tasks* participating in the monitoring process compete for resources at any step of the workflows of Figure 1.1.

With respect to the source of the measurement data, two challenging scenarios are considered in this thesis, which cover a wide range of use-cases in terms of monitoring design, implementation, and information requirements.

1) Monitoring software-defined networks In this scenario, the measurement data is produced by SDN switches, e.g., OpenFlow-enabled devices, distributed over the network. Monitoring systems operating in such a scenario can fall short in meeting the scalability requirements of large networks and in supporting management applications with short latency requirements [1][20]. One key problem is posed by traditional centralised monitoring architectures, which are prone to high processing workloads and monitoring latencies when handling large numbers of information sources. A second limiting factor is the reduced bandwidth SDN switches can use to report monitoring information to controllers/managers, which can adversely affect the granularity and timeliness of monitoring reports.

2) Monitoring in a software dataplane In this scenario, the measurement data is produced by a programmable dataplane running on a commodity server, using the flexibility of software to perform traffic measurements. Such monitoring implementations are required to tolerate high traffic rates (10 Gbps+) while executing elaborate measurement operations on a per-packet basis. Meeting these requirements is challenging, given that significant disruptions can occur under adverse operating conditions [28] [8] if monitoring-related operations are not dynamically adapted to ensure an efficient use of available dataplane resources.

Satisfying the requirements posed by these two demanding scenarios is a complex task, which

requires to overcome a number of problems related to measurement execution (*A1*) and retrieval (*A2*), monitoring data processing (*A3*) and knowledge delivery (*A4*). This PhD thesis addresses the challenge of an efficient and accurate monitoring function by tackling the aforementioned problems. It investigates how to efficiently extract information from network devices and timely deliver knowledge, in order to meet the requirements of large scale software-defined networks and support highly-reactive network reconfigurations. It explores how efficient use of the available dataplane resources can be achieved dynamically in a software dataplane, thus ensuring accurate monitoring reports and resilience under adverse operating conditions. Lastly, it investigates how to efficiently process measurement data extracted by a dataplane to produce elaborate monitoring reports in real-time. Specifically, the contributions made by this thesis are the following.

B1) Scalable monitoring architecture for software-defined networks. A scalable monitoring architecture for SDN has been designed and implemented to provide timely and consistent monitoring updates to heterogeneous management applications. It satisfies the requirements of large-scale networks by reducing monitoring information delivery delays and avoiding processing bottlenecks.

B2) Efficient retrieval of monitoring information from the SDN dataplane. Focusing on the problem of retrieving measurement data from SDN switches, this thesis proposes a novel approach enabling accurate data collection with limited burden on the switch resources. The associated rationale is to adjust data retrieval operations frequently and automatically based on traffic dynamics.

B3) Adaptive monitoring functions for software dataplanes. This thesis introduces *adaptive* solutions where measurement-related operations of software dataplanes are timely reconfigured in response to changes in traffic and resource availability. The proposed solutions allow to achieve resilience in the face of bottlenecks for a wide range of conditions, and can be deployed with minimal overhead.

B4) Accuracy-preserving monitoring functions for software dataplanes This thesis investigates how to achieve global accuracy goals of monitoring reports under dynamic operating conditions (e.g., traffic characteristics, monitoring workloads). In particular, this thesis provides solutions to (i) infer potential accuracy degradation for any measurement task at run time, and (ii) timely recover from accuracy degradation while efficiently using available dataplane resources.

B5) Efficient processing of measurement data This thesis explores the problem of producing elaborate monitoring reports from raw measurement data captured by dataplanes. A generalised approach, inspired by machine learning practices, is proposed to reduce data processing costs, which does not depend on specific monitoring designs and is applicable to a wide range of different monitoring information.

1.3 Thesis outline

The remainder of the thesis is organised as follows.

Chapter 2 provides an overview on the monitoring requirements of modern network management,

it highlights monitoring-related issues associated with software-defined networking, and describes how recent research has addressed the challenge of developing efficient monitoring functionalities.

Chapter 3 proposes a self-adaptive and decentralised monitoring framework for SDN. This relies on a distributed monitoring architecture to cope with the requirements of large-scale networks consisting of a large number of geographically dispersed devices, and is designed to support a wide range of measurement tasks and requirements in terms of monitoring rates and information granularity levels. The proposed framework introduces SAM (Self-tuning Adaptive Monitoring), an adaptive solution that enables efficient extraction of monitoring information from the network switches through accurate reconfigurations of the switch query rate. In contrast to other solutions proposed in the past, SAM requires almost zero tuning effort, as the algorithm parameters are automatically updated based on the evolution of the traffic shape.

Chapter 4 presents MONA, an adaptive traffic monitoring framework for software dataplanes. MONA solves two key problems under increasing workload conditions in the monitoring pipeline. On the one hand, it guarantees resilience to bottlenecks by timely reconfiguring the monitoring operations under dynamic operating conditions. On the other hand, it preserves the accuracy of monitoring reports according to user-specified accuracy thresholds. To maintain high accuracy levels, MONA estimates the potential loss of monitoring information at run time, and reconfigures the monitoring process to recover possible accuracy degradations. Accuracy reductions are quantified using a novel, task-independent, accuracy estimation technique, which guarantees high levels of confidence by computing estimates adjusted to recently-observed traffic characteristics.

Chapter 5 investigates how the measurement data extracted at the dataplane can be efficiently processed, focusing on the case of modern network *telemetry* systems, *i.e.,* monitoring systems responding in real-time to monitoring queries issued by management applications and operators. This chapter introduces a generalised approach for reduced-cost processing of measurement data inspired by machine learning workflows. The proposed approach is based on fast classifiers that are (i) trained over recent traffic, (ii) automatically tuned to match accuracy requirements of the monitoring queries, and (iii) applied at run time to infer elaborate monitoring results from subsets of the measurement data.

Chapter 6 summarises the main proposals and findings of this thesis before identifying future research directions.

Overall, this PhD thesis addresses the challenge of accurate and resource-efficient monitoring from different viewpoints. Chapter 4 addresses the execution of multiple measurement tasks at the dataplane through mechanisms that avoid performance bottlenecks and achieve global accuracy

objectives by using dataplane resources efficiently (*A1*). Chapter 3 targets the efficient extraction of monitoring information from the dataplane (*A2*), and investigates how monitoring systems can handle a large number of (geographically dispersed) devices, while avoiding processing bottlenecks (*A3*) and delivering knowledge in short timescales (*A4*). Chapter 5 tackles the problem of reducing data-processing costs (*A3*) when building elaborate monitoring reports in response to a wide range of monitoring queries using the raw information extracted at the dataplane.

Chapter 2

Background and Related Work

In the last years, monitoring systems have been challenged by novel, more stringent requirements posed by network management applications. These reflect the deep changes of management processes pushed by the advent of Software-Defined Networking (SDN). First, SDN has enabled the development of applications that can automatically react to network events and perform fine-grained resource reconfigurations. This has made management processes more dependent than before on the accuracy and timeliness of monitoring information. Imprecise monitoring reports can lead to chains of wrong reactions, while high monitoring delays can leave transient network behaviours and short-lived problems undetected. Second, novel network programming abstractions have enabled a variety of new management applications by simplifying the translation, as well as the composition, of high-level operator policies. As a result, increased flexibility is required from monitoring systems in terms of variety of knowledge offered and different information granularities.

Collectively satisfying these needs is not a trivial task. It requires the monitoring systems to extract and process huge amounts of heterogeneous measurement data in real-time, while facing massive traffic volumes, a large number of switches and hosts, and limited hardware resources. In other words, it poses the need of *efficient* monitoring functionalities, which scale to large traffic amounts and network sizes, consume limited resources at switches and hosts, and support diverse, fine-grained, and real-time network information.

This chapter explores how the challenge of efficient monitoring has been addressed in the recent literature. At first, Section 2.1 provides an overview of the key monitoring information requirements of modern network management. Section 2.2 focuses on monitoring-related issues introduced by SDN, highlighting the limitations of traditional measurement practices. In Section 2.3, an overview of recent approaches is presented, focusing on the main functionalities of monitoring systems. Lastly, Section 2.4 presents final remarks on the related work.

It should be noted that, while a general discussion on monitoring for SDN is provided here, additional details on related work, specific to the main thesis contributions, will be included in the remaining chapters.

2.1 Monitoring requirements of network management

To effectively manage a network, a variety of monitoring knowledge is required, concerning a number of different events and emerging network conditions. A brief overview of the key monitoring information requirements is presented here using a top-down approach, *i.e.,* starting from the ultimate goals of network management [11]: (i) guaranteeing network security; (ii) ensuring high network utilisation (thus reducing costs); (iii) improving performance of services using the network.

Securing the network To make networks secure, it is necessary to detect a number of potential malicious behaviours. This requires the monitoring systems to report on a wide range of specific traffic patterns and anomalies at short timescales and with high level of precision. Short timescales allow to handle short-lived attacks and to enforce countermeasures before services could be adversely affected. At the same time, high reporting accuracy is important to avoid false alarms and disgraceful countermeasures. In general, three monitoring approaches can be adopted to detect attacks based on network traffic:

- *Signature-based attack detection* This approach consists in searching for specific attack fingerprints, for example specific protocol messages or end-hosts communication patterns [33].

- *Threshold-based attack detection* This approach consists in raising alarms when thresholds are exceeded on specific traffic characteristics. For example, alerts for DDoS (distributed denial of service) victims can be triggered when a host is contacted by more than x hosts [34]. In [10], SYN flood attacks are notified when more than x incomplete TCP handshakes are found between the same source and destination.

- *Anomaly-based attack detection* Anomaly detection solutions [35] [36] [37] are based on the assumption that attacks will change the usual network behaviour in terms of traffic properties. The idea is to use measurements to build *baselines*, *i.e.,* models summarising the normal network behaviours, and to trigger alerts when the behaviour of the network differs from the baseline. For example, the solution in [35], which detects Port Scan attacks, uses hypothesis testing to discriminate possible scanners from benign hosts (baseline).

High network utilisation and reduced costs A combination of management tasks participates to this objective, including efficient network planning, provisioning, and traffic engineering. These rely on accurate profiling of the resource usage. In particular, the main attributes of interest [38] are (i) the available network link bandwidth and (ii) the size of network flows. The former, which is used to determine how much traffic can be allocated to a specific network path, can be monitored by measuring the total byte rate through a switch/router interface (if link capacity is known), e.g., [39], or using probes [40], *i.e.,* sending packet trains and studying the time gap between the arrivals of two successive probes at the receiver. The latter can be used for traffic matrix estimation [41] [42], *i.e.,* to determine a vectorial representation of Origin-Destination pair traffic-rates, or to compute

top-k elephant flows [43] and heavy hitters [44]. Such knowledge is key to drive traffic engineering decisions [4], as well as for solving network resource planning and provisioning problems.

High application performance To preserve application performance, management systems must ensure timely diagnosis of performance problems and be able to take effective reconfigurations on traffic engineering, load balancing, or job scheduling. This requires quick identification of changes in traffic patterns and events such as congestions, load imbalance, routing problems, software bugs and failures at servers. Representative monitoring use cases on performance diagnosis are described below, based on the examples found in [8].

- *Diagnose server load balance problems* One of the main objectives of network operators is to maintain the service-level agreements (SLAs) for applications running on the network hosts, such as cloud services [45] and networked caches. Load impairments, as well as short-lived, imbalanced request bursts can lead to higher service latencies (long tail latency [46]). To avoid performance degradations, the monitoring systems should timely report on latency changes (e.g., inflated TCP round-trip times), and on the traffic volume anomalies that caused the additional delay.

- *Diagnose congestion* It is generally hard to diagnose network congestions, especially if related to transient, short-lived episodes such as TCP incast [32], where multiple applications send traffic through the same switch for a short time, causing transient packet loss. To diagnose such problems, monitoring should provide detailed information, on fine timescales, on flow packet-loss, large traffic aggregates, and switch queueing delays.

- *Diagnose packet loss* Root cause analysis of packet loss can be daunting for network management systems, due to large network scales and increasing complexity in network-application interactions. To achieve such goal, monitoring should detect even small scale loss (e.g., by measuring retransmissions), and correlate it to short-lived packet bursts.

2.2 Software-defined networking: monitoring opportunities and open issues

Software-Defined Networking (SDN) relies on separating the data plane and the control plane, thus making data plane nodes (*i.e.,* network switches) simple packet forwarding devices, and leaving the control and management of the entire network to a logically centralised software program. This novel view has deeply changed the way networks are managed, by allowing for frequent changes on resource configurations following dynamic network conditions, and by enabling the expression of complex network configuration and operators' policies using high-level languages.

Such a shift has largely impacted the design and evolution of monitoring systems, which have been facing in the last years the technological novelty of SDN, new measurement information

sources, and novel, more diverse and stringent monitoring requirements of management applications. In this section, a brief overview on the main opportunities offered by SDN is provided with a focus on monitoring-related benefits (Section 2.2.1). Then, the key problems that such opportunities have posed on monitoring systems are highlighted (Section 2.2.2).

2.2.1 Opportunities offered by software-defined networking

The shift towards SDN brings two key opportunities for improving the monitoring functionality. First, it empowers monitoring with new, programmable data-plane primitives and protocols such as OpenFlow [12] and P4 [17], which allow to extract highly-configurable sets of measurement data on different granularities. Second, it provides novel programming interfaces and abstractions, which can be leveraged to build automated monitoring tasks.

New data-plane primitives With the advent of SDN, several new data-plane primitives have been introduced, specific to different types of hardware targets.

- *Commodity switches* By supporting new protocols such as OpenFlow (this is the case for many vendors such as HP, NetGear, NEC), commodity switches have started offering flexible flow-based counters based on the use of ternary content-addressable memory (TCAM). These can be used to measure traffic aggregate size on different granularities (e.g., source-destination IP pairs, or packet header 5-tuple).

- *Programmable switches* Novel protocol independent switch architectures (PISA), e.g., Tofino switch [47], have enabled features such as programmable header parsing, customisable hash functions, flexible match/action tables pipelines, and stateful computations through packet header writing and state registers at the switch. In addition, the P4 language has provided the syntax and the constructs to build complex processing pipelines in the switch dataplane. These technologies allow to implement very diverse measurements using hash-based data structures [48] [49] and match-action tables [9].

- *Hosts* The proliferation of efficient packet capture libraries (e.g., DPDK [15], Netmap [16], eXpress Data Path [50]), high speed network cards, as well as the developement of software switching (e.g., Click [13], OpenVSwtich [51]) and Network Function Virtualisation, have enabled fast packet-processing platforms in software supporting a wide range of network functions. Monitoring can benefit from such deployments to perform fine-grained traffic analysis using the processing power of servers.

Network programming interfaces and abstractions Apart from the new data-plane enablers, a variety of new abstractions, programming languages and APIs has been recently introduced to simplify and enhance network programming. Part of these efforts have focused on the interactions with the data-plane. In particular, SDN control planes (e.g., NOX, Floodlight) allow operators to write network software in C++ or Java, by offering a set of application programming interfaces (APIs)

to interact with switches. Other proposals have instead provided ways to process (e.g., compose) high-level management policies and convert them into control-plane software. For example, Procera [2] enables event-driven network policies using a high-level functional programming language, while Pyretic [3] language introduces abstractions to build complex management applications from the composition of independent modules.

Although these solutions are not measurement-specific, monitoring systems can largely benefit from the improved network programmability. In particular, they can build upon SDN control-planes to automate the interactions with network devices, and use similar abstractions to improve flexibility with respect to different application requirements – as for the case of NetAssay [52] or Path Query [53] monitoring-query languages, both based on Pyretic.

2.2.2 New requirements and issues raised by software-defined networking

By relying on highly-configurable data-planes, and using new programming interfaces and abstractions, software-defined networks enable a wide range of applications that reconfigure the network automatically. The enhanced automation enables more frequent reconfigurations, as operators' manual effort is not involved anymore in reconfiguration control loops. At the same time, the use of programmable data planes allows for more fine-grained network configurations, up to the granularity of 5-tuple flows or even single packets [54] [9] [8].

As a result of these conditions, supporting automatic reactions requires monitoring systems to report on a wide range of network events, including very fine-grained ones (e.g., transient traffic bursts and flow volume changes) with high precision and over short time scales. High accuracy is needed to avoid misconfigurations or chains of wrong reactions in a fully automated management process. Fine time scales are required to react to failures, intrusions, or bottlenecks on resources in a timely way, *i.e.,* before these episodes can cause any damage to those services and tenants using the network. Furthermore, monitoring reports are not only expected to be more accurate and timely, but also more diverse in terms of results/metrics and more *network-wide*. This is because SDN has enabled a variety of new management applications (e.g., from the composition of operators' policies [3]) which operate based on a unified view of the network.

To satisfy such stringent requirements, and cope with the evolution of networks towards larger scales and traffic volumes [55] [8], monitoring systems must extract and process massive amounts of measurement data, deliver it very frequently and with short delays, and satisfy heterogenous needs in terms of metrics and statistics offered. A number of problems is associated with these challenges, which can be summarised as (i) hardware-resource limitations and (ii) time-related limitations.

Hardware-resource limitations To satisfy all aforementioned requirements, monitoring systems may consume considerable amounts of resources at data-planes, as well as on the infrastructures where monitoring information is processed and distributed. Bottlenecks, *i.e.,* resource shortages, can arise due to the following reasons:

- *Limited resources devoted to monitoring* On switches, vendors allocate most of the limited memory resources to network functions other than measurements, e.g., firewall, forwarding. At the same time, on hosts most of the computational resources (e.g., CPU cores) are assigned to revenue-generating applications.

- *Increasing network scales* Monitoring systems should face high traffic rates and large number of flows, and deal with large numbers of devices. Measuring more flows translates into more monitoring state information to maintain, hence more competition on switch memory resources, which is an issue on commodity switches (e.g., due to limited TCAM space [56]), as well as on novel programmable switches (e.g., only small key-value store can be allocated to fast on-chip SRAM [54]). In addition, higher traffic rates entail more competition on hosts' processing capabilities. Finally, dealing with higher number of network devices can cause unsustainable workloads when processing monitoring information from many sources (e.g., at information stream processors [9]).

- *Dynamic traffic and monitoring demands* Changes in traffic and network characteristics can force monitoring to consume more resources, with the risk of exceeding hardware constraints. For example, packet rate spikes and variations in the traffic distribution (e.g., decreasing traffic skew [28]) can overload the packet-processing pipelines especially on hosts [8], while the increase of the number of concurrent flows can saturate the switch memory [21] [22]. In addition to this, more resources are needed when increasing numbers of *queries* are raised by monitoring applications in reaction to specific events and emerging network conditions (e.g., anomalous traffic volumes).

Time-related limitations Excessive time consumption and/or scarce time availability have become key issues for monitoring systems. In SDN, automated management tasks can operate at reduced time-scales and improved level of detail, and as such they require frequent monitoring reports carrying detailed results in terms of metrics and statistics. As a result, monitoring should include more elaborate measurement operations, while reporting on shorter time intervals to align with the control loops of management processes. Nonetheless, with the evolution of networks towards larger traffic volumes (*i.e.,* more flows, higher rates), less time is available for processing individual packets or reporting granular traffic statistics. Overall, the following time-related issues have arisen.

- *Limited per-packet times at dataplanes* Programmable switches need to process up to a billion packets per second (for a 64-port switch, and 10Gbps per port [57]). Under these conditions, each packet should be processed in roughly 1ns at each pipeline stage of the switch, hence only a limited set of measurement operations can be performed. Similarly, on a host supporting 10Gbps on a single CPU-core [8], per-packet time should be less than 70ns. Exceeding such constraints would result in starving and possibly dropping packets in the input buffers.

- *Slow information extraction from dataplanes* Switches can generally push very limited monitoring information onto the switch-controller channel. In the case of OpenFlow-enabled switches [22] [21], no more than a few hundreds of flow-rules can be reported per second. This results into excessive monitoring latencies for applications that need to take granular, flow-level, reconfigurations for thousands of flows on fine time scales [5].

- *Limited time to process monitoring information* Monitoring systems should rapidly process (e.g., aggregate, evaluate, correlate) the measurement data extracted from the dataplane to provide real-time knowledge on network conditions/events. This may not be possible under large network and traffic volume sizes, as the increasing workloads can inflate processing latencies.

- *Slow delivery of monitoring knowledge* Software-defined networks allow management processes to operate on a network-wide view. As a result, the delay incurred by the dissemination of huge amounts of monitoring information over long geographical distances is a key limitation for network management reactivity [1].

2.3 Overview of recent proposals

Generally speaking, traditional network monitoring practices, *i.e.,* the ones based on Netflow [6], sFlow [58], SNMP, or packet traces (e.g., tcpdump [7]), can neither cope with the new monitoring requirements of software-defined networking, nor overcome the aforementioned issues. First, common measurement tools at network devices, e.g., Netflow/sFlow flow-counters or SNMP logs, can only provide infrequent and coarse-grained information due to well known scalability issues. In particular, Netflow or sFlow can be sustainable in data center networks only with sampling rates in the range of 1 in 1000 [11], while SNMP can provide port counters only every few minutes [8]. Second, traditional approaches generally lack of effective abstractions for representing elaborate operators' queries, and express selective measurement operations. As such, they force applications to collect huge amounts of raw monitoring data "just in case", which makes it extremely costly to find needles in haystacks [9]. Lastly, they do not provide appropriate abstractions for expressing network-wide monitoring objectives, or efficient methods to automatically map objectives to individual measurement operations at specific network locations [9] [53].

With the objective to fill these gaps, a number of monitoring solutions have been recently proposed, which have tackled the challenge of efficient monitoring from three different angles: (i) improving measurement design; (ii) improving monitoring abstractions; (iii) improving monitoring runtime systems.

Measurement design A number of proposals have addressed how to perform different types of measurements for large volumes of network traffic, in real-time and supporting diverse measurements requirements. These solutions entail not only the adoption of the new data-plane primitives

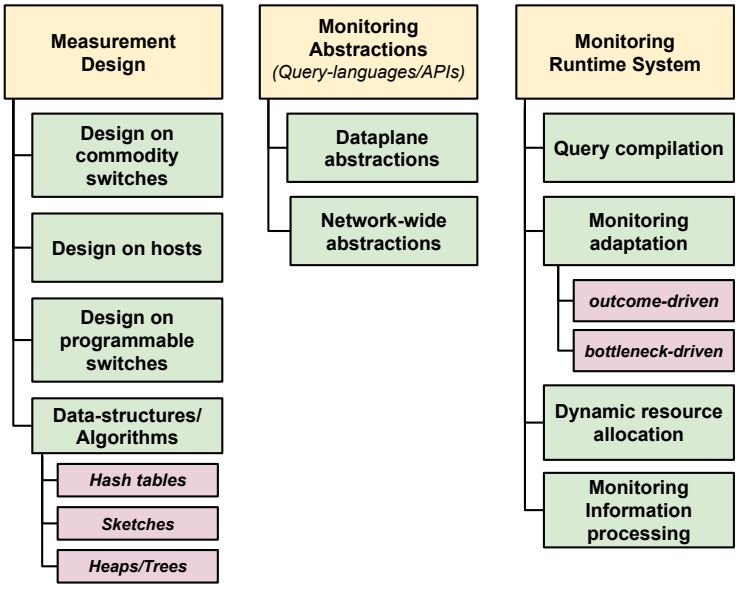


Figure 2.1: Overview of recent proposals on measurement design, monitoring abstractions and monitoring runtime system

of programmable networks for performing measurements, but also the design of new, measurement specialised, data structures (e.g., universal sketch [48]), new protocols (e.g., In-band Network Telemetry [59]), as well as system (*i.e.,* on-device) optimisations enhancing the measurement capabilities of data-planes.

Monitoring abstractions Recent research efforts have provided languages and interfaces that allow operators (or management applications) to declare network-wide monitoring objectives in the form of high-level *queries*, *i.e.,* with abstractions hiding the specific devices involved or specific packet characteristics. Highly-expressive monitoring abstractions (e.g., languages, programming interfaces) are fundamental to guarantee support for more diverse sets of management tasks, but also play a role towards the goal of efficient monitoring. Indeed, poor abstractions can result, as discussed before, into too much monitoring information collected and delivered to applications, leading to increasing information processing latencies and additional stress on the hardware resources.

Monitoring runtime system Finally, several recent approaches have targeted those tasks sitting between the monitoring queries and the measurement primitives that form the so-called monitoring *runtime system*. The main operations involved in the monitoring *runtime* are (i) translating high-level monitoring requirements into specific measurement primitives, (ii) adapting the measurement operations according to traffic dynamics and emerging network conditions, (iii) dynamically allocating available hardware-resources and time to the different monitoring queries, and (iv) processing the measurement data to form relevant monitoring knowledge. Improving these operations is essential for monitoring efficiency, as it allows for high accuracy in monitoring reports while reducing the resource consumption (*i.e.,* the monitoring overhead).

An overview of the main proposals in literature is presented in the following subsections for each of the three research directions.

2.3.1 Measurement design

This subsection presents how recent research has addressed the design of efficient measurements on commodity switches, programmable switches and hosts, focusing on the enablers (e.g., the data-plane primitives) and system optimisations used. The presentation is followed by a description of the main measurement data-structures designed/adopted and the associated algorithms.

Measurement design on commodity switches By enabling OpenFlow, commodity switches allow to specify which flows to monitor based on different combinations of packet header fields (including IP addresses, TCP/UDP ports, Type of Service, etc), and count the number of bytes and packets. Measurements can be then performed *passively* by the controller, i.e., by waiting for notification messages about flow expiration, which contains the latest flow counters, or *actively*, by polling a switch for updated counters using ad-hoc Read State messages. These enablers have been extensively used to implement a wide range of measurements. Flowsense [27] determines the utilisation of links between OpenFlow-enabled switches by using the PacketIn and FlowRemoved messages sent by switches to the controller upon the arrival of a new flow or upon the expiration of a flow entry, respectively. In OpenTM [60], the controller periodically pulls the switch flow rules with fresh flow-counters, which are used to generate traffic matrices. The work in [25] proposes pulls of flow counters with random inter query times that are effective to measure the flow autocorrelation. OpenNetMon [61] use a mix of passive and active approach to measure the throughput of flows.

Openflow counters have been also adopted to unveil heavy traffic aggregates [23], e.g., *heavy hitters* (IP prefixes accounting for more than $x\%$ of the total traffic) and *hierarchical heavy hitters* (IP prefixes that contribute to more than $x\%$ of traffic, after excluding any heavy descendants in the prefix tree) [62] [63] [64]. The main leverage for these solutions is the use of *wildcarded* flow-rules (based on switch TCAM), *i.e.,* matching on subsets of bits of the packet header fields, that enable counting at different granularity levels.

A few attempts to derive *latency* measurements in OpenFlow networks can also be found in the literature. In [65], the authors use probe packets, sent by the network controller to switch x and retrieved from a remote switch y, in order to compute the link latency between x and y. In SLAM [66] and OpenNetMon [61] a similar mechanism is adopted to monitor the path latencies in data center networks.

Measurement design on hosts Over the last years, increasing attention has been devoted to the deployment of per-packet measurements on the network end hosts, using packet-processing pipelines in software, especially in the context of modern data center networks [8] [67]. Such trend has been favoured by the increased speed in packet-acquisition, the evolution of network cards (NICs), and new packet capture libraries such as DPDK [15] and Netmap [16]. These allow to cope with

increasing traffic rate from the network cards (10Gbps or more), while by-passing the OS kernel so that packets can be conveniently processed in the userspace.

The work in [68] represents one of the first attempts to deploy high speed traffic measurements on servers. The proposed design, supporting up to 4 Gbps rate, relies on the PF_RING capture library to schedule packets to multiple server cores. This library implements full zero-copy, i.e., it maps userspace memory into the memory region of the network card driver allowing user applications to direct access the card registers and data, without the intermediation of kernel packet buffers. In [69] the authors presents nDPI, an open-source packet inspection tool for commodity hardware that can handle up to 10 Gbps traffic. Similarly to [68], this solution exploits PF_RING to process packets at line rate, and can classify on average 3.5 millions packets per second using a single core. The work in [70] addresses the challenge of processing multi 10Gbps traffic for per-flow measurements by improving packets processing stages such as buffering and (flow-preserving) load balancing. To this end, it also explores how to improve the use of Receive Side Scaling (RSS [71]), *i.e.,* the hardware queues offered by modern network adapters.

More recently, a few optimisations have been proposed to improve the measurements performance on servers (*i.e.,* more throughput, or less per-packet latency). The work in [8] leverages processor cache prefetching to reduce per-packet delays and stores per-flow information in huge memory pages to minimise TLB (Translation Look-aside Buffer) misses. Conversely, the approach of Agg-Evict [72] is to improve cache locality by aggregating packets based on the flow id before the measurements take place.

Measurement design on programmable switches Novel enablers such as the P4 language and PISA (Protocol Independent Switch Architecture) have been recently embraced to implement a variety of measurements at line rate on the switch hardware. A representative set of proposals is presented here.

HashPipe [73] prototypes heavy hitter detection entirely in the switch data-plane, at line rate. It relies on multiple match-action stages, each using a set of P4 registers for recording flow counters. Furthermore, it uses packet metadata – P4 allows placing state information (e.g., register values) in the packet header – to track intermediate per-flow results. Dapper [74] is another P4-based measurement design, which enables TCP performance diagnosis in a P4-enabled data plane. It exploits all main functionalities of P4 (and P4-enabled PISA switches): P4 flexible header parsing is used to retain different packet header information, packet metadata are used to carry information to the following processing stages in the diagnosis, P4 registers are used to store the state of TCP connections, and match-action tables allow to check test conditions. Sonata [9] relies on the same features/functionalities to add data-plane support for generic, streaming-based, traffic analytics, by implementing operators such as *map*,*reduce* and *filter*. Lastly, the proposal Marple [54] introduces a new programmable key-value store primitive for PISA switch hardware, which is able to perform *stateful* measurement operations, such as moving average of per-flow latencies, at line rate and with

support to millions of keys.

Orthogonal to previously described approaches, In-Band Network Telemetry [59] consists in a protocol and a set of measurement primitives to detect latency changes and congestions at switches. It uses the customisable header parsing/re-writing offered by P4 to record the queueing delays experienced by a packet at the switch, so that latency spikes can be unveiled.

Measurement data structures and algorithms To conclude, an overview is presented on the main classes of algorithms that have been recently adopted at data-planes to efficiently produce measurement data from packet streams.

Hash tables Algorithms based on simple hash tables consist in computing a hash function from a flow key extracted from the packet header (e.g., using source/destination IP and source/destination port) and using the result to locate a bucket in the table, where flow key and measurement values are stored. A few recent approaches use hash tables to implement measurements on hardware switches, for example in HashPipe [73] a pipeline of hash-tables is deployed on programmable switch hardware, with one table per match-action stage. Meanwhile, hash tables have been more-widely adopted on hosts to perform measurements in software [8] [28], due to the ease in applying them to a wide range of measurements. However, in such cases their implementation can significantly affect the packet-processing performance [75]. For example, chained hash tables, where colliding keys are placed into the same bucket and chained, can inflate the per-packet time if collision rate is high due to the additional memory accesses needed. Several optimisations are available, the most common being Cuckoo hashing [76] (with $O(1)$ worst-case insert), which is widely adopted by software switches [14] [77].

Sketches Sketches are compact data structures used to *approximately* summarise data streams, where the output size (i.e. the summary size) is much smaller than the input size (e.g., the total amount of flow-counters to be stored). The most simple sketch is the *bitmap*, that maintains a single array of bits to count the number of unique elements, e.g., the different IP source addresses. Another widely-used sketch, especially for size-based measurements such as heavy hitters, is the Count-Min sketch [78], which keeps a two dimensional array of integer counters with d rows and w columns. It computes d hash functions per packet, and updates the corresponding d positions in each row. To find the counter for a given IP address, the minimum counter in the d locations is returned. If the minimum counter is above the threshold, the corresponding IP address is added to the set of heavy hitters. The use of sketches is particularly convenient at switches. First, they can fit into SRAM, which is usually larger and cheaper than switch TCAM. Second, they can produce fairly elaborate summaries at the data-planes, e.g., not only raw flow counters, but also flow size distribution and counts of unique elements. This means that measurements are already aggregated at the switch, and as such their retrieval is more lightweight. The main drawback of these techniques is that they require ad-hoc support from commodity switch components.

Heaps and trees Lastly, heap and tree-based approaches should be mentioned, both geared

towards low memory consumption. Heap-based algorithms only keep in memory the most important values for a measurement task. The most known is the Space Saving algorithm [79], which relies on a small hash table and is mainly use for elephant flow and heavy hitter detection. Tree-based solutions keep in memory hierarchical sets of counters, and as such they are naturally fit to zoom in and out portions of the traffic while using limited memory, e.g., to implement flow counting [23] and detect hierarchical heavy hitters [20] with the switch TCAM.

2.3.2 Measurement abstractions

This subsection reviews the recent work on monitoring abstractions (e.g., query-languages, programming interfaces) that allow operators and management processes to provide their monitoring objectives without dealing with network and packets' details. Guaranteeing rich, highly expressive abstractions plays a non marginal role towards the goal of monitoring efficiency. In particular, by using precise representations of selective operator's *intents*, monitoring systems are relieved from the need of indiscriminately collecting, storing and processing huge volumes of data [11] [52].

Two classes of monitoring abstractions are discussed below: *data-plane* and *network-wide*. The former enable dealing with packet stream processing while hiding the data-plane details of switches and hosts, and the latter allow to express network-wide monitoring objectives while hiding the measurement operations needed at different network locations.

Data-plane abstractions SDN allows operators to specify network control tasks based on high-level resource representations [80] [24]. A similar approach has been recently explored for monitoring, using abstractions that enable expressing traffic in terms of applications and hosts rather than data-plane details. A few representative proposals are summarised here.

NetAssay [52] is a monitoring programming interface based on Pyretic [3]. It relies on the notion of *virtual packet header* to allow a programmer to express policies in terms of queries on network traffic. A virtual packet header includes standard packet header fields, as well as metadata associated with that packet (e.g., the switch where the packet is located). Furthermore, virtual headers support additional metadata expressing higher-level features such as a user, device, or application.

Other proposals have instead focused on the definition of monitoring-query languages. NetQRE [10] is a declarative language for *quantitative* network monitoring, *i.e.,* to combine network and application-layer performance metrics with traffic patterns such as known attacks. It allows to write monitoring functions that take as input a stream of packets and return result values, or subsets of the packet stream. This approach enables highly-selective monitoring tasks by using particular filters on traffic, namely Quantitative Regular Expressions, which integrate regular expressions with numerical computations. Sonata [9] is a declarative language for streaming analytics on network traffic, which relies on the *packet-as-a-tuple* abstraction. In other words, operators can express their monitoring queries directly over packet tuples, with each tuple capturing the properties of a packet.

A similar, packet-tuple, approach is adopted by Marple query-language [54], which additionally enables to express "stateful" aggregation over multiple packets in a stream. This allows monitoring queries to support more elaborate functions such as EWMA (exponentially weighted moving average).

Network-wide abstractions To provide an accurate picture of how the network behaves, monitoring systems should not only disclose how traffic looks like at a specific switch or host, but also expose measurements on the traffic routes in the network. As such, efficient network-wide monitoring abstractions should be offered to enable queries such as *what path does the traffic follow?* or *is path delay acceptable?*. Two representative approaches are overviewed below.

The first one is based on the notion of *packet history*, *i.e.,* the full journey of a packet throughout the network. This approach is followed by NetSight [81], which offers an API to nofity (or act upon) packet histories. To enable the monotoring system to sustain large traffic volumes and network scales, the NetSight API provides a packet-history filter, based on regular expressions, which allows to express "selective" interest in histories with specific trajectories or encountered switch state.

The second approach, adopted by Path Query [53], relies on the notion of path. Path Query provides a monitoring-query language, based on regular expressions similarly to NetSight, with supports for boolean conditions on packet location and header contents. The main difference with Netsight is that Path Query also supports in-band measurements, *i.e.,* it can discover network paths by tagging packets with metadata.

2.3.3 Monitoring runtime system

Having the right measurement enablers at data-planes and the right monitoring abstractions is not enough to ensure an *efficient* monitoring functionality satisfying the requirements of software-defined networks and the larger demand in terms of traffic amounts and network sizes. The overall monitoring efficiency largely depends on those "middle-layer" tasks between monitoring queries and measurement primitives, which (i) compile high-level queries into measurement-specific operations (e.g., algorithms running at data-planes), (ii) adapt the measurement operations when the operating conditions change, (iii) dynamically allocate resources to different measurements to satisfy concurrent monitoring objectives, and (iv) process raw measurement data extracted at data-planes to deliver relevant monitoring knowledge to network management. In the following, the recent approaches and research directions for these tasks are overviewed.

Query compilation Representative examples of query compilation are briefly presented here starting from the aforementioned monitoring abstractions.

In the case of the NetAssay [52] monitoring programming interface, higher-level metadata on traffic specified on the API are translated into flow rules for SDN switches. To produce selective flow-rule sets, and thus reduce monitoring overhead on switches, the compilation integrates the high-level metadata with external sources of information such as BGP routing updates and DNS

messages. More elaborate is the approach of Sonata [9] to query compilation, which consists in splitting query execution between stream processors (on hosts) and programmable switches, trying to run as much as possible of the query on the match-action tables of PISA switches (guarantee line-rate packet processing), and minimise the number of packet-tuples sent to the stream processors (reduce processing workloads on hosts). Query compilation in NetQRE is instead geared towards a low memory footprint of monitoring query implementation. In particular, to avoid storing too many packets for answering queries on a packet stream, the compiler automatically infers what state needs to be maintained at a time, and optimises the query execution accordingly. To this end, it translates the regular expressions used in NetQRE queries into specific types of finite state machines, which are updated at run time when query-related events are detected on packets (e.g., a given source IP address x is observerd).

Considering network-wide monitoring queries, a key example is the Path Query [53] compiler, which maps queries to data-plane programs whose implementation is distributed across the switches of a path. In a similar fashion to NetQRE, data-plane programs are represented by finite state machines. However, one big difference compared to NetQRE is that in Path Query the state is stored on each packet and updated while it traverses the network.

Monitoring adaptation As discussed in Section 2.2.2, satisfying the information needs of network management processes can translate into heavy usage of hardware resources and available time, especially when frequent and detailed monitoring updates are required. This is naturally the case of SDN, where management tasks are automated and operate on fine granularities, thus being extremely dependent on the precision and timeliness of monitoring information [8]. The overhead associated with the monitoring functionality can be a problem on hardware-switches, e.g., due to well-known TCAM memory scarcity [20] [62] or limited control channel bandwith [22] [21], as well as on servers, due to limited available time per packet, e.g., less than 70ns for 10Gbps traffic.

To improve monitoring efficiency, a number of adaptive monitoring approaches have been proposed. These solutions reconfigure measurement operations at run time to deal with temporary resource shortages [8] [82] or to find good tradeoffs between the monitoring overhead and the accuracy of monitoring reports [20]. Overall, these approaches can be classified based on the conditions that trigger adaptations.

Adaptations triggered by monitoring outcome A first class of solutions is represented by algorithms that modify the monitoring behaviour according to recent measurement results. One example of such approach is the SDN monitoring framework Payless [83], which includes an algorithm for tuning at run time the frequency at which switches are polled for fresh flow counters. The algorithm used is threshold-based, *i.e.,* the polling rate is increased when counters are growing more than x, and decreased under smaller variations. Alternative proposals have focused on how to adapt switch memory consumption to the measurement outcome. A representative example is the algorithm proposed by Zhang in [23] for flow-size measurements in SDN. This operates continuous expansions

and reductions of the set of switch flow-rules devoted to a specific traffic aggregate based on the behaviour of the aggregate itself: on significant volume changes, the algorithm *zooms in* (*i.e.,* more flow-rules) to unveil more fine-grained traffic components, while granularity is reduced if the aggregate volume is stable. A similar case is the work in [62], where the same zoom in/out approach is adopted for hierarchical heavy hitter detection, with the main difference being the algorithm used, which just relies on thresholds, while the solution in [23] adopts linear prediction.

Adaptations triggered by limited time/hardware-resource availability A second class of approaches perform measurement adaptations under bottleneck conditions, *i.e.,* at times when available resource are not enough to cope with the monitoring demand. One simple solution is provided by the SDN monitoring framework Dream [56] in the case of switch TCAM-based measurements. Dream includes an admission control algorithm, which checks the available TCAM space left in the switch at the reception of a new monitoring query. If not enough space is available, the algorithm simply drops the incoming query. More elaborate solutions have been developed in the context of traffic monitoring on hosts. In [28], Alipourfard et al. propose to resize the hash tables used for storing per-flow measurement information in order to obtain smaller per packet times, thus solving the bottleneck on available measurement time. However, the resizing involves a substantial cost. For a single hash table, the average "amortised" cost of resizing – both for increasing and decreasing the table size – is $O(1)$ per insert, which means that doubling the table size from m to $2m$ has a cost in the order of $O(2m)$ additional insertions. Lastly, SketchVisor [82], a framework for sketch-based measurements, adds a fast measurement path to the data-plane and redirects portions of traffic to it so that time can be saved under bottlenecks.

Dynamic resource allocation In software-defined networks, large numbers of concurrent measurement tasks can be dynamically (and automatically) instantiated by network controllers/managers. When many measurement operations compete for scarce resources, as in the case of hardware switch TCAM, it is important to apply configurations that guarantee efficient use of the resource. To improve the overall monitoring *efficiency*, such configurations should allow for more concurrent measurement operations supported at data-planes, without penalising too much the accuracy of monitoring reports. Different approaches rely on solving optimisation problems to derive optimal resource allocation setups. A seminal work in this area is CSamp [84], which solves a linear programming problem with a MaxFlow formulation for allocating flow-sampling tasks to different switches. A similar, optimisation-based approach has been adopted for allocating TCAM memory of SDN switches, one representative example being the work by Nguyen et al [85]. All these solutions generally suffer from scalability problems, as convex optimisation can hardly provide real-time reconfigurations if the number of measurement tasks is large, or when allocation should be distributed over many switches.

On a different line of research, the proposals DREAM [56] and SCREAM [86] dynamically adjust the resource allocation configuration based on traffic conditions and resource availability. The

former addresses the case of switch TCAM resources at SDN switches. The approach consists of estimating, for each monitoring task using ad-hoc algorithms, the accuracy degradation deriving from a reduced allocation of TCAM space to the task. For example, for a Heavy Hitter detection task, DREAM estimates how many heavy IP prefixes would be "missed" when less flows are tracked (*i.e.,* less TCAM space used). Based on such estimation, part of the TCAM resources is moved at run time from high-accuracy tasks to low-accuracy ones. The framework SCREAM, proposed by the same authors, addresses instead the case of sketch-based measurements. In this case, the allocation is performed dynamically by assigning to each sketch the *right* size, *i.e.,* the minimum one that provides sufficient measurement task accuracy. Sketches are then resized when traffic conditions change. To quantify the accuracy level of a sketch, the framework estimates the hash collisions associated with the current sketch size and traffic characteristics. As in the case of DREAM, accuracy estimation algorithms are task-dependent.

Monitoring information processing To support the needs of automated management processes producing frequent and fine-grained network reconfigurations, monitoring systems should not only efficiently extract measurement data from the data-plane, but also process such data in real-time to swiftly respond to a variety of monitoring queries. Data processing efficiency has been addressed in SDNs with methods to aggregate network state [24] or split the total data processing workload, but these solutions have mostly targeted network control (e.g., distributed SDN control planes) instead of traffic monitoring. More recently, however, a few monitoring-related approaches have been proposed, pushed by the deployment of more complex and *stateful* measurement operations at data-planes, especially on hosts and programmable switches, which entail larger data processing costs. The approach taken by Sonata [9], for example, is to exploit the capabilities of programmable switches to reduce the data volumes sent to analytics stream processors. In particular, it offloads the execution of part of the data processing operations (e.g., filter, reduce) to match-action stages at the data plane. A different approach is followed by the monitoring framework Trumpet [8], in which the processing of measurement data is carried out using server computational resources only. Its solution is to split processing (e.g., aggregation, evaluation of flow-related characteristics) into multiple, time-limited batches, interleaved by the processing of new traffic packets so that measurement data processing can run on the same processor core as packet-processing.

2.4 Summary

This chapter has provided an overview of the key issues related to monitoring future, software-defined networks and has presented the main research directions in this domain. As discussed in Section 2.3, recent work in the literature has been tackling the challenge of an efficient monitoring function at different levels.

A first research direction has focused on improving the design of measurements (Section 2.3.1). Proposals in this research area have investigated how to exploit software-defined networking tech-

nologies to improve measurements at hardware switches (e.g., using OpenFlow, P4, protocol independent switch architectures) and network hosts (e.g., using software switching, fast packet-capture libraries). Moreover, the proposed approaches have designed algorithms/data-structures to inspect packet streams at the data-plane with reduced memory footprint or time consumption. A second direction of research (Section 2.3.2) has been on the design of monitoring abstractions, such as monitoring-specific languages and programming interfaces, that can represent elaborate, diverse and network-wide monitoring objectives. By allowing operators to express their exact monitoring needs, these solutions have enabled more selective, and thus more resource-efficient, measurement operations. The last research direction has been towards improving the monitoring runtime system (Section 2.3.3). This consists of all monitoring-related processes meant to (i) map high-level monitoring queries to specific "on-device" measurement primitives, (ii) coordinate the execution of different measurements at the data-plane and (iii) handle (*i.e.,* extract, process, deliver) the measurement data in order to provide elaborate monitoring reports to network operators and management applications.

The contributions of this thesis are along the last line of research. The following three chapters cover all the main aspects related to the design of efficient monitoring runtime systems. The next chapter presents a monitoring framework designed to achieve reactive delivery of monitoring knowledge, to avoid monitoring information processing bottlenecks, and to ensure lightweight extraction of measurement data from the network switches. Moreover, it proposes a modular architecture that allows for efficient translation of high-level monitoring requirements into implementation-specific measurement commands. Chapter 4 focuses on the problem of running different measurements at the data-plane. It provides the necessary adaptive mechanisms to make an efficient use of data-plane resources, and under dynamic operating conditions. Lastly, Chapter 5 presents a methodology, based on intelligent filtering, for reducing processing costs when elaborate monitoring results are generated from the raw measurement data collected at data-planes.

Chapter 3

Decentralised and Self-Adaptive Monitoring for Software-Defined Networks

3.1 Overview

At first, this thesis considers the monitoring of software-defined networks, a scenario where measurements are performed based on SDN-enabled switches (e.g., OpenFlow [12] compatible devices) distributed over the network, and the resulting information is used to support management applications that reconfigure the network automatically and frequently [30] [19] [31]. Over the recent years, a number of research proposals on SDN monitoring have explored the implementation of task-specific measurements [61][26] and investigated how to efficiently allocate finite memory resources of SDN switches to different measurement operations [56][23][87] under heterogeneous traffic workloads and for different operator's objectives. These solutions can however fall short in supporting management applications with short latency requirements [1][20] and in meeting the requirements of large-scale networks (*i.e.,* with large number of geographically dispersed devices), due to two main conditions that result to inefficient extraction, processing and delivery of monitoring information.

Firstly, these approaches mainly rely on the assumption of a centrally-managed network, which is an important limiting factor for the case of large-scale networks. As the network diameter grows, the associated latencies can become considerable, thus penalising the responsiveness of the network management system. Also, monitoring a large number of nodes can generate unsustainable loads on the central controller/manager due to the increasing amount of measurement traffic converging to it, with the effect of inflating the network configuration times [88]. Decentralised solutions for SDN have been proposed in the literature, *e.g.,* [88][89][24], but their main focus is on the control plane (*i.e.,* routing functionality), devoting less attention to monitoring, which is reduced to periodically synchronising topology databases.

Secondly, monitoring solutions for SDN generally extract information from the network devices based on regular measurement intervals, *e.g.,* based on a fixed switch query period. As such, they can

fail in detecting short-lived network episodes, especially if the switch query rate is too low, or can saturate the switch control bandwidth when measurements are too frequent. Although adaptive SDN monitoring approaches exist in the literature [23][83][25], they all require some complex parameter tuning and need continuous adjustments under dynamic traffic patterns.

This chapter addresses the above limitations by proposing a novel monitoring framework for SDN that achieves the goal of lightweight information extraction from network devices and realises highly-reactive information delivery while avoiding processing bottlenecks.

To satisfy the needs of large-scale networks, the framework relies on a *decentralised* architectures involving multiple monitoring entities, each able to perform monitoring tasks autonomously without relying on a central manager and without maintaining a global view of the network run-time state. Although decentralised monitoring is not a new topic in network management, previous solutions like [90][91] are not directly applicable to the new domain of software-defined networks due to: *i)* the technological novelty of SDN, *ii)* the shift towards new measurement enablers and *iii)* the heterogenous requirements of management applications that can reconfigure the network at a wide range of timescales and granularity levels (*e.g.* up to a single TCP flow). The architecture proposed is designed to operate within a distributed management environment such as the one in [1], where local managers hosting the application logic can adaptively reconfigure the network resources under their scope of responsibility at short timescales. To deal with the diversity of controller implementations and improve configuration flexibility, it abstracts most of the monitoring functionality from the control plane, and interacts with SDN controllers through a minimal interface(s) which could be extended to support new control software without requiring significant changes. Such a design can support the monitoring requirements of a wide range of management applications, and effectively aggregate monitoring-related operations to reduce the overall hardware resource consumption.

To enable the efficient extraction of monitoring information from the network switches, the proposed framework incorporates SAM (*Self-tuning Adaptive Monitoring*), a novel adaptive monitoring method that guarantees timely and accurate reconfigurations of the switch query rate. As opposed to previous solutions, such as [83] and [23], SAM requires minimal tuning effort as the algorithm parameters are automatically updated based on the evolution of the traffic shape. A comparative performance analysis shows that SAM always provides more predictable and reliable results in terms of both monitoring precision and resource consumption with respect to existing approaches. Moreover, it demonstrates that SAM can always match – and in some cases even outperform – the best-case accuracy of previous methods [83] [23], *i.e.,* the one obtained with the optimal parameter settings, while using less amount of resources.

The benefits of the proposed framework are evaluated based on two realistic and demanding use cases. In the first one, a distributed management application coordinates a content distribution service in an ISP network, while in the second, the network operator runs an on-demand gaming service by offering processing resources (*e.g.,* specialised hardware) as part of the network infrastructure.

To evaluate the performance of the decentralised monitoring approach in terms of monitoring latency, as well as traffic overhead, this is compared against a centralised solution based on two realistic network topologies. In addition, the effect of monitoring operations on the two use case services is extensively evaluated by focusing on the impact of monitoring data extraction (through the use of SAM), as well as monitoring information synchronisation. The results show that the proposed decentralised monitoring approach can reduce the monitoring delays by up to 60% compared to a centralised one, which translates to more reactive control loops. They also highlight that SAM can produce significant benefits on the use case services at a reduced cost in terms of utilisation of the switch resources. Finally, the evaluation shows that although relaxing the synchronisation of monitoring information can impact the service performance, it is possible to achieve substantial reductions in monitoring overhead while minimising potential service disruption.

The remainder of this chapter is organised as follows. Section 3.2 provides background information on the distributed network management framework considered in the chapter, and presents open issues of SDN monitoring. In Section 3.3, the design of the proposed architecture is described in detail. Section 3.4 presents the SAM approach and compare its performance with state-of-the-art monitoring solutions. Section 3.5 describes the use case services considered for the evaluation of the proposed solution. Experiment setup and evaluation results are presented in Section 3.6. Section 3.8 describes related work close in spirit to the proposed framework, and Section 3.9 summarises the chapter.

3.2 Background

This section provides background information on the SDN-based resource management framework considered for the design of the proposed monitoring solution, as well as an overview of the techniques used for performing measurements in SDN infrastructures.

3.2.1 Distributed resource management framework

The work in [1] presented a novel SDN-based network management and control framework that supports dynamic resource management applications in fixed backbone infrastructures. This chapter adopts the design principles of the relevant architecture, which separates management and control functionality, allowing the two to evolve independently. A set of *local managers* (LMs), distributed over the network, host various management applications (MAs) that implement the necessary logic to decide on network (re)configurations. MAs are instantiated on the local managers as modules embedding information data structures and running on a common execution environment offered by the LMs. Each MA can execute in all LMs or in a subset of them (e.g. the ones operating at edge network nodes). Configuration decisions taken by LMs are translated into sets of commands, transmitted to the forwarding hardware through a *southbound* interface (e.g., *OpenFlow*), which defines the sequence of actions to be enforced for updating the network parameters.

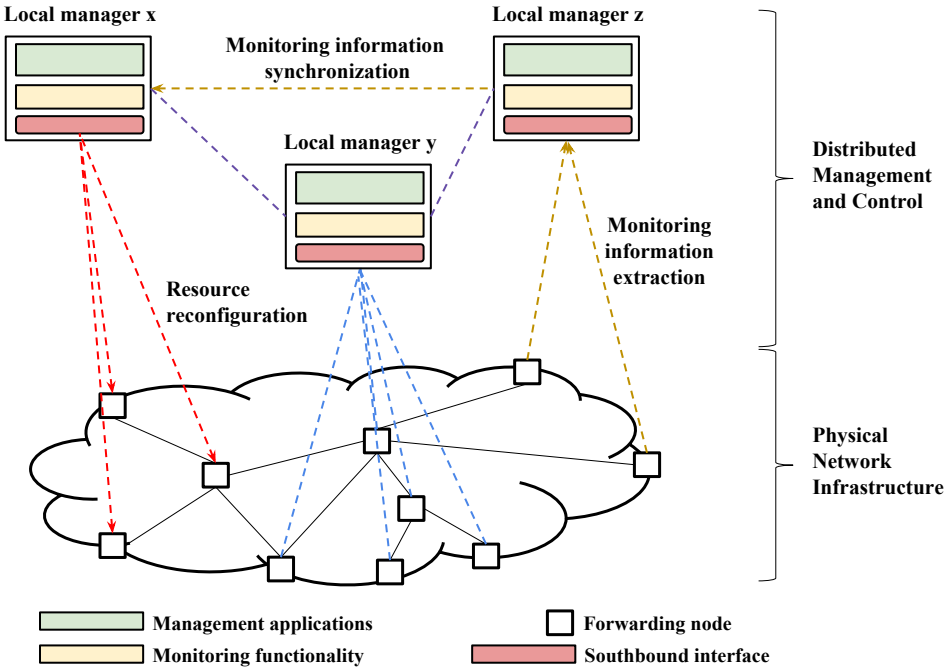


Figure 3.1: Distributed resource management framework for software-defined networks

Monitoring is an essential component of LMs. First, it is concerned with extracting raw statistics from the physical resources and generating useful information for applications. In this context, each LM needs to implement the necessary capabilities to collect the status of variables (*e.g.,* links, traffic flows) within its local scope and make this information available to local MA instances. Second, since MA instances operating at different locations may need monitoring data gathered from outside their local scope, the monitoring functionality is therefore also concerned with disseminating network state updates to remote LMs. In a SDN environment, such synchronisation phase is essential for reconfiguring the network parameters based on a global, unified network view. This information can be exchanged between instances of a distributed MA through the signaling framework proposed in [92], which provides a communication protocol and the necessary primitives to share the monitoring information and to coordinate decisions between two or more MA instances.

Figure 3.1 depicts a simplified representation of the resource management framework considered in this chapter. The forwarding nodes are partitioned in clusters, each under the control of a LM. The monitoring functionality initially retrieves raw data from the forwarding nodes, which is subsequently processed (*e.g.,* filtered, aggregated) to form knowledge, and is made available to local MA instances, *i.e.,* the ones operating on the same LM. In addition, a subset of the generated knowledge can be shared with a remote manager (*e.g.,* between *LM z* and *LM x* in Figure 3.1) on a communication channel established using the signaling protocol in [92]. Using the synchronised information, a remote manager (*LM x* in Figure 3.1) can reconfigure its own partition of the network infrastructure by interacting with the forwarding nodes in the cluster.

3.2.2 Monitoring software-defined networks

Compared to traditional computer networks, where monitoring solutions require ad-hoc software installation / configuration and low-level tools, SDN has introduced a set of simple and configurable primitives for the collection of measurement information at switches, which make them suitable to a wide range of management tasks. SDN flow-based switches (*e.g.,* OpenFlow enabled devices) allow network operators to flexibly specify the flows to monitor based on different packet fields (*e.g.,* source and/or destination IP addresses), and to count the number of bytes or packets for these flows. Counters can be fetched by polling a switch, e.g., using OpenFlow read-state messages.

This measurement approach is affected by several hardware technology issues. First, flow-based counters are maintained in expensive and power-hungry TCAMs and, as such, only a limited number of entries can be used for measurements. Another issue is the limited bandwidth between the switch and the SDN controller, which limits flow fetching to no more than a few thousand per second [22]. Finally, SDN-enabled switches may also exhibit inaccuracies when updating the flow counters. For example, as discussed in [21], some devices do not update the counters every time a new packet matches a rule, but perform the updates periodically instead. Furthermore, devices from different vendors introduce different biases in measurements and may even present some limitations in terms of protocol support. Despite these open issues, the framework presented in this chapter relies on the *counting* approach – *i.e.,* polling the network devices for raw counters – due to its implementation simplicity, the wide support by different vendors, and configuration flexibility in terms of information granularity and measurement frequency. Alternative methods, which implement hashing techniques (*e.g., sketches*) on the network hardware [34], or require enhanced programmability (*i.e.,* beyond OpenFlow) of the forwarding plane [59] [17] [18], still have very limited support on hardware switches, which makes their applicability uncertain. At the same time, alternative solutions based on packet sampling and stream processing [93] [94] can pose a much higher processing burden on local managers, *e.g.,* in the case of large-scale networks with a limited number of LMs, and are unsuitable for many management applications due to the adoption of packet sampling [95].

3.3 System architecture

This section presents the proposed monitoring system and motivates the design principles of the associated architecture.

3.3.1 Requirements and design choices

Effective monitoring system design has to consider a number of key issues. If very intrusive, monitoring operations can adversely affect the network performance. At the same time, these operations need to be frequent and fast to enable management applications to operate in short timescales. In addition, they should provide accurate and high-granularity information to support configuration de-

cisions. The impact of these issues is amplified in the case of large-scale SDNs, since configuration decisions might be taken far away from the locations where monitoring is performed. Three main requirements are identified below, which have been taken into account for the design of the proposed monitoring architecture.

- **Scalability** The monitoring system should be able to cope with a large number of information sources. As the number of physical resources under the scope of a single MM increases, the monitoring traffic converging to it and the associated computational load could drastically impact the system reactivity, as was shown in [24][22]. While in dense networks with small diameter (*e.g.,* data centers) this drawback can be mitigated through replication or by investing more CPU cycles and memory, in wide area networks (WANs) the monitoring responsiveness is significantly affected by network latencies.

- **Programmability** The frequency and granularity of measurements have to be highly configurable based on the requirements of heterogeneous management applications. While some applications such as elephant-flow detection need fine-grained flow-based measurements, others only require aggregate statistics. In such a case, low-granularity measurements which can be retrieved at a lower cost are preferable (*e.g.,* switch port measurements as opposed to individual flow measurements).

- **Responsiveness** MAs can change their monitoring requirements based, for example, on the analysis of measured metrics. The MM should be responsive in adapting measurement parameters, such as the polling frequency or the flow-level granularity, according to new requirements. Fast adaptations, as argued in [83][23], are essential for warranting acceptable information accuracy and can additionally reduce the monitoring overhead.

Overall, the monitoring system should (i) scale well with increasing network size in terms of both number of information sources and network diameter, (ii) be highly configurable to meet the needs of a variety of management applications and possibly reduce costs, and (iii) responsively adapt its operations to strike the right tradeoff between the accuracy of monitoring reports and the monitoring overhead. With the goal to address these requirements, the following design choices have been made for the monitoring architecture.

Decentralised monitoring approach The proposed monitoring system leverages a decentralised approach where each of the local managers described in Section 3.2 hosts a monitoring entity, called the *monitoring module* (MM), which is responsible for gathering information within the scope of the LM. Scalability for coping with a large number of network devices and their geographical span is the main driver for selecting a distributed approach. Specifically, such approach can reduce the total amount of monitoring traffic handled by individual LMs, and can achieve low delays when reconfigurations are computed close to where the monitoring information is collected.

Modular structure of monitoring entities Each MM relies on a modular composition to maximise the system extensibility and improve the overall flexibility of the solution. The modular structure allows to decouple the logic involved in the processing of the application requirements from the one operating on the raw measurement primitives. This reduces the deployment effort when new types of requirements need to be supported, or new measurement mechanisms become available.

High-level declarative monitoring interface In the proposed architecture, management applications use a common interface, *e.g.,* a RESTful interface, offered by the MM for both injecting new monitoring requirements and receiving the corresponding measurement results. Such interface allows applications to declare their monitoring information requirements in a high-level form, so that MAs do not need to deal with dataplane details, and allows MAs to specify their preferences on the update frequency of monitoring reports. This is essential for the monitoring system to satisfy the monitoring needs of a wide range of MAs and to enable a more efficient use of the resources.

Dynamic measurement scheduling The monitoring system has been designed to dynamically configure measurement operations based on current operating conditions. In particular, the scheduling of measurements takes into account both the resource availability at the switches and recently-observed traffic patterns. In the proposed architecture, scheduling relies on an admission control to avoid indiscriminately admitting monitoring tasks that could not receive enough switch resources, and leverages an adaptive algorithm (SAM) to dynamically adjust the measurement times based on traffic dynamics.

3.3.2 Monitoring module

Figure 3.2 presents the structure of the MM, which sits between MA instances and the southbound interface. The MM components operate on two main workflows. The first one starts with the MAs registering their monitoring needs on the MM interface in the form of requirement tuples. These are parsed into low-level monitoring specifications by the *Requirement Processor*, mapped to individual measurement operations by the *Scheduler*, and lastly translated into calls to measurement primitives by the *Measurement Engine*. On the second workflow, measurement data received from the switches are processed (e.g., aggregated, compared to previous information) by the *Result Processor* and finally they are delivered to MAs on the MM interface. In addition, the MM includes a *Persistent Data Repository* for storing network topology and setup information, and a *Synchronisation Interface* for handling the exchange of monitoring data between LMs.

As an example, a Traffic Engineering MA can be considered, which requires monitoring results on the (i) utilisation of network links and on the (ii) flow throughput of source-destination pairs in the network. Initially, the Requirement Processor maps the link-utilisation requirements to "switch, port" specifications and the flow-throughput ones to "switch, flow-rule" pairs. The Scheduler then generates for each link-utilisation requirement a set of OpenFlow port-rate measurements to be executed at different times, and for each flow-throughput requirement a sequence of OpenFlow

flow-rate measurement operations. The Measurement Engine translates each port-rate (flow-rate) measurement into a call to the controller that generates an OpenFlow Port (Flow) Statistic Request. Responses received from the switches are finally parsed by the Result Processor and the retrieved port (flow) counters are compared to the previous samples to estimate the link load (flow throughput). The obtained results are exposed to the local Traffic Engineering instance or, if the information has been required by a remote instance of the MA, delivered through the Synchronisation Interface.

The proposed MM architecture is designed to operate with OpenFlow-based measurements. It is compatible with all OpenFlow versions (1.0 to 1.5) since it makes use of a minimal set of messages, common to all versions of the protocol, for the collection of individual flow statistics, aggregate flow statistics and switch port statistics. Moreover, given the modular structure of the MM, the set of supported measurement primitives can be extended to different technologies/protocols beyond OpenFlow by only acting on a subset of the MM components. Specifically, extensions of the monitoring system require (i) adding new "translation rules" to the Requirement (and Result) Processor to map MAs requirements to new, different monitoring specifications, and (ii) providing appropriate calls to the new measurement primitive(s) in the Measurement Engine. Following this approach, the proposed architecture can be extended to support SNMP (e.g., SNMP link-load measurements) or P4 [17] (*i.e.,* extracting measurement-related information from P4 switches).

The various components of the MM are described below.

3.3.2.1 Persistent Data Repository

This component maintains network information which is not updated frequently, such as the topology graph representation (*e.g.,* switches and links) and the current setup of paths between pairs of edge nodes. Such information can be represented through transactional databases and can be flexibly accessed/modified by a SQL-like querying mechanism.

3.3.2.2 Requirements Processor

The first task of this component is to parse new monitoring requirements received from applications. These are registered in a local data store (*Requirements Table*, *c.f.* Figure 3.2), with each requirement represented as a tuple:

⟨ `Req_id, MA_id, Task, HL_targets, Mon_times` ⟩

Req_id and *MA_id* are the unique identifiers of the monitoring requirement and the requesting management application. *Task* represents the overall goal of the measurements, for example the utilisation of one or a set of links. *HL_targets* is the list of targets (high-level identifiers in the application's abstract view of the network) of the monitoring task, for instance, in case of a path utilisation request, the corresponding list of paths. *Mon_times* can be a single parameter, *i.e.,* the polling period, or an explicit sequence of measurement intervals. Section 3.4 presents an adaptive polling mechanism (*SAM*) where the measurement rate is continuously adjusted based on the network traffic behaviour. When this mechanism is enabled, the MA instance should only provide, under the attribute

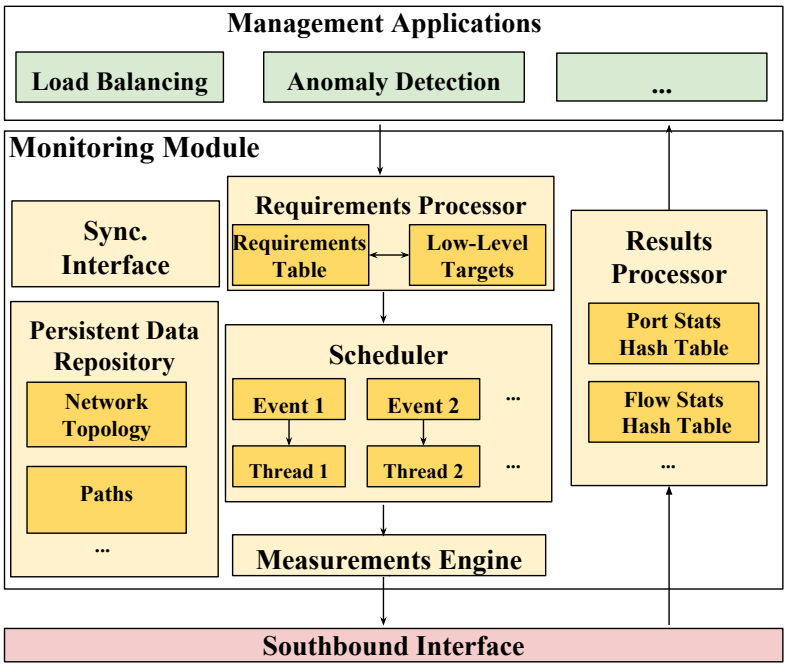


Figure 3.2: Monitoring module architecture

Mon_times, the boundaries for the measurement rate, *i.e.,* the interval of acceptable monitoring frequencies $[f_{min}, f_{max}]$.

The next procedure performed by this component is a translation routine, based on the *Task* specification, that maps each new table entry into one or more *Low-level targets*. Each low-level target (*LL_target*) can identify a specific physical resource, *e.g.,* a switch port, or map a set of flow rules, *i.e.,* a specific subset of the switch flow table. These entities are stored in the *Low-level targets* table with the following format: *Op_type* indicates what type of measurement operation should be performed, for example collecting the average traffic rate of a specific switch interface. [*Req_id*] is the list of pointers to the corresponding application requirements, used for the reverse translation. *Sched_state* is a flag indicating whether the low-level target refers to a new monitoring requirement (*i.e.,* measurement operations have to be scheduled from scratch), or to a previous task for which some adaptation is required (*i.e.,* operations have to be re-scheduled).

The acquisition of statistics from a switch poses a substantial burden on the device in terms of both processing and control bandwidth [22] [21]. As a result, for each time unit only a limited amount of statistics can be reported by the switch to the MM. Any data exceeding the limit can be lost or considerably delayed, which penalises the accuracy of MAs operations. To mitigate this issue, the proposed solution aggregates different monitoring tasks, when possible. In other words, before the insertion of a new low-level target, the table is looked up for similar entries. An existing entry is *similar* if the target is equivalent or included, *e.g.,* two targets with the same *Task* and *Mon_times* attributes are similar if one refers to the flows matching source IP address 128.40.200.1 and the other

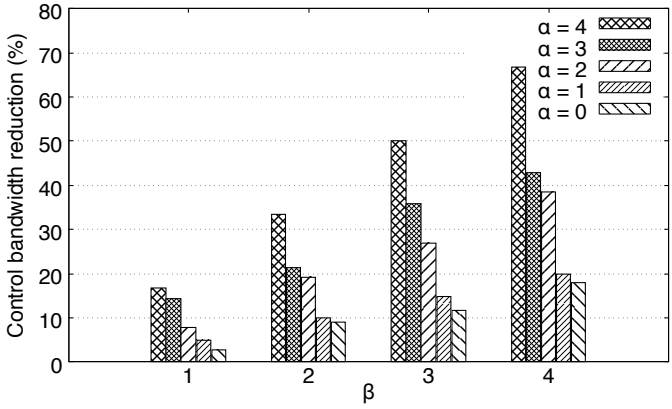


Figure 3.3: Reduction of switch control bandwidth based on aggregation.

corresponds to the flows for any source IP in the subnet 128.40.200.0/24. In such a case, the MM merges the two low-level targets and the corresponding measurement times is updated accordingly to satisfy both requests. Such a feature reduces the consumption of switch resources, especially for different MAs requiring flow measurements at similar times, and/or for similar portions of the switch flow space. The result is a more sustainable monitoring (reduced monitoring load) when measurements compete with other control plane operations, such as new flows setup, for accessing the switch resources.

Resource saving through aggregation. To show the gain that can be achieved through aggregation, a representative scenario is considered, where 3 monitoring requirements m_1, m_2, m_3 are registered at the same time and on the same MM by three different MAs. The execution of the measurements associated with each m_i results in fetching a fixed number of flow table entries k at a period $p_i \in [1,2,..7,8]$ from the same switch. Two parameters α and β are associated with each set. Parameter α represents the level of temporal concurrency of the required measurements, which ranges from 0 (lowest level of concurrency, *e.g.,* $[p_1, p_2, p_3] = [6,7,8]$) to 4 (maximum level of concurrency, *i.e.,* $p_1 = p_2 = p_3$). Parameter β represents the level of overlap in terms of similar flow entries required from the switch, and also ranges from 0 to 4. $\beta = 0$ represents the case of no overlap. For $\beta = 1$, there exists an overlap of 50% between the requirements of two MAs. For $\beta = 2$, the three MAs share 50% of requests. For $\beta = 3$, two MAs have completely overlapping requirements and share 50% with the third one. $\beta = 4$ represents an overlap of 100% between the three MAs. Figure 3.3 shows the resulting gain in terms of switch control bandwidth. The case $\beta = 0$ is not depicted as it does not result to any savings.

Significant reductions can be achieved when $\beta \geq 2$. For example, for $\beta = 2$, an average reduction close to 20% is obtained. Such savings can allow the overall measurement rate to increase, thus enhancing the resource reconfiguration reactivity. For instance, assuming a fully saturated bandwidth between the switch CPU and the local manager, aggregation can allow an increase of the monitoring rate by up to a factor of 3 (*i.e.,* $\alpha = \beta = 4$).

3.3.2.3 Scheduler

This component is in charge of generating and managing the individual measurement procedures (*e.g.,* the ones requiring a single message exchange with a switch), which are executed as threads. It is called on every insertion in the *Low-level targets* table, and on any modification of existing tuples involving the measurement times. Scheduling new measurements indiscriminately can lead to none of them getting enough switch resources. As such, once invoked, the scheduler executes an admission control routine, in which it verifies, depending on the current measurement load, whether the measurement procedures for the new low-level target can be performed. In case there are not enough resources available to accommodate the new measurement procedures, these are rejected and the corresponding MAs are notified so that monitoring requirements can be re-negotiated. The measurement load for a specific switch is defined by the expected monitoring bandwidth, which is estimated on a time-window basis given the list of the low-level targets already scheduled. This metric depends on the current measurement rates (*i.e.,* the average polling frequency) and on the number of flow (or switch port) records returned for each measurement procedure in the corresponding OpenFlow *Statistics Reply* messages. In this respect, the approach in this chapter differs from recent proposals, which focus on the limited flow table TCAM space [56] rather than the control bandwidth.

Once accepted, the new low-level target is mapped onto a set of events, each one associated with a timer to trigger the new measurement thread. The way this mapping is executed depends on the switch polling algorithm used by the MM. In case of fixed frequency measurements, the mapping is executed all at once, and all timers are recorded in the scheduling table. When the adaptive polling mechanism presented in Section 3.4(*SAM*) is enabled, the measurement events and timers are generated one by one, based on the feedback (*e.g.,* the new value of a low-level target) provided by the *Results Processor* upon receiving fresh monitoring data.

3.3.2.4 Measurements Engine

This component, called every time a new measurement thread is triggered, operates as an interface between the measurement thread under execution and the measurement mechanisms implemented on the southbound interface. It assigns individual measurement procedures to one of the available primitives offered on the interface and supported by the underlying device. Such an interface is essential to allow most of the monitoring operations to remain independent of specific implementations.

3.3.2.5 Results Processor

This component receives the raw measurement results, for example messages of type OpenFlow statistic reply. These are parsed (*e.g.,* into JSON format) and the *Low-level targets* table is looked up for the corresponding target(s). Based on the operation type specified in matching table entries, the measurements are filtered to select the required counters. These are stored in corresponding data structures (hash-tables) and used for computing the metrics of interest. Finally, the processed results

are associated to the relevant high-level targets and delivered to MAs through update messages. In case the MM adopts an adaptive monitoring scheme like the one presented in Section 3.4, the Results Processor also generates a call to the Scheduler, so that new measurement events can be adaptively scheduled based on variations of the relevant metrics over time.

3.3.2.6 Synchronization Interface

In addition to the aforementioned components, the MM offers an extensible interface for the synchronisation of monitoring information in a distributed management plane. Using the methods of this interface, instances of MAs can forward monitoring reports through signaling channels as described in [92]. The interface provides different solutions for the exchange of monitoring data. In the simplest case, this can take place periodically or every time new statistics are available locally. More advanced solutions consist in exchanging new values only when they differ substantially from the previously reported ones. These techniques can improve the synchronisation efficiency as they allow management applications to strike the right tradeoff between accuracy and overhead in monitoring data dissemination.

3.3.3 Workflow examples

As concrete examples of the MM workflow, two simple monitoring procedures are considered:

1. Average link utilization The requirements processor registers a requirement of type *link utilization*, with a fixed measurement period, for a set of links $l_1, l_2, ..l_n$. Each of these is translated into a low-level target s_x:p_y, where s_x identifies the switch and p_y the port on which the bitrate should be measured. According to the specified measurement period, for each target the scheduler periodically generates measurement threads, each calling the controller to create an OpenFlow *Port Statistics Request* and send it to s_x. The results processor receives the corresponding *Port Statistics Reply* messages from the controller, together with the measurement timestamps, and extracts the current byte counters (*tx_bytes*, *rx_bytes*). It then uses the new sample and the previous one, which is stored in a hash-table, to compute the average link utilisation, and finally returns the value to the application.

2. Average flow throughput Consider an application requiring the throughput of the flow identified by source IP y and destination IP z. The requirements processor translates this into a low-level target s_x:src_y:dst_z, where s_x identifies the switch from which the flow counters should be fetched. By default, the ingress switch for that flow is selected as the measurement target. Then, the scheduler generates periodic calls to the SDN controller to build a *Flow Statistics Request* and sends it to the target switch. As the corresponding *Flow Statistics Reply* is received by the controller, the result processor extracts the current flow *byte count* and *duration*. By comparing two consecutive samples, it computes the average flow throughput, and reports this to the application.

3.4 Self-tuning Adaptive Monitoring

Querying a network switch for updated statistics involves a tradeoff between accuracy and overhead. On one hand, continuous pulls of fresh statistics impose a considerable burden on the switch hardware, *e.g.,* on its scarce control channel bandwidth [22], and can overflow the network capacity with monitoring overhead. On the other hand infrequent measurements fail in capturing transient events, such as short-lived congestion, due to sampling and averaging bias. To reduce the monitoring load while ensuring timely and precise reports, adaptive monitoring mechanisms by which the query frequency is dynamically reconfigured are needed.

To enable efficient extraction of monitoring information, SAM (*Self-tuning Adaptive Monitoring*) is introduced, a novel adaptive monitoring method that guarantees timely and accurate reconfigurations of the switch query rate. As opposed to previous solutions in the literature (*e.g.,* [83], [23]), SAM requires almost zero tuning effort, as the algorithm parameters are automatically updated based on the evolution of the traffic shape.

Comparative performance analysis shows that SAM always provides more predictable/reliable results in terms of both monitoring precision and resource consumption with respect to the existing approaches. Moreover, it shows that SAM can always match – and in some cases even outperform – the best-case accuracy of previous methods [83] [23], *i.e.,* the one obtained with the optimal parameter settings, while using less amount of resources.

3.4.1 Limitations of current adaptive approaches

Two main techniques have been proposed in the literature to adapt the switch query rate: i) threshold-based approaches [83] and ii) prediction-based approaches [23]. The objective of these approaches is to adapt the period at which switch variables are queried based on network traffic behaviour. While in the case of threshold-based approaches, the adjustment is based on threshold conditions, a linear prediction of the evolution of the variable value is used in the case of prediction-based approaches to update the period. The advantage of these techniques is that they can easily apply to different monitoring operations – for instance, flow size and link utilisation estimation. Other methods exist but they are bound to specific measurement tasks, *e.g.,* flow autocorrelation [25].

3.4.1.1 Threshold-based adaptive monitoring

The principle of Threshold-based Adaptive Monitoring (TAM) is to adapt the monitoring period (*i.e.,* rate at which the switch is queried) based on the variation of the variable values between two consecutive measurements. If the difference is above a threshold th_1, the monitoring period is divided by a constant d, while if it is below a threshold th_2, the period is multiplied by a second constant m. In essence, the sharper the variation, the shorter the period and hence the more intense the polling.

3.4.1.2 Prediction-based adaptive monitoring

In contrast to the threshold-based approach, Prediction-based Adaptive Monitoring (PAM) uses the history of the last set of collected measurements (not only the last one) to decide how to adjust the monitoring period. More specifically, based on the previous N collected values of monitored variable x denoted as $x_1,...,x_n$, a predictor x_p of the next value of x is computed. When the actual new value x_{n+1} of x is fetched, x mean and standard deviation are updated and the *real* variation $(x_{n+1}-x_n)$ is compared to the *predicted* one (x_p-x_n). If the predicted variation is substantially higher than the real one, *i.e.,* for $x_p > mean(x)+\alpha \cdot std(x)$, the monitoring period is increased by a factor equal to d. In this case, the prediction is overestimating the change. Otherwise, if the value of x is changing much faster than predicted, *i.e.,* for $x_p < mean(x)-\alpha \cdot std(x)$, the period is divided by d. In both cases, α is a small integer constant. In all other cases, the period is unchanged.

The main issue with TAM and PAM is that, to efficiently adapt the period, they both require some complex parameter tuning, *i.e.,* thresholds th_1, th_2 and multipliers m, d for TAM, and selection of sample queue size N and factor α for PAM. This is specifically evident under periods of bursty traffic, where accurate reconfiguration of the query rate is essential. In particular, changes in the traffic burst patterns (*e.g.,* burst amplitude, duration, inter-arrivals) are likely to require new settings given that for a specific pattern, only a small subset of setups guarantees efficient statistics collection.

To illustrate this issue, the two approaches have been implemented to measure the size of a bursty flow over a period of 5 minutes and compared the obtained results with the actual flow size denoted as *ground-truth*, measured every millisecond. To emulate a bursty profile, the size of the flow is modulated between 0 and 10 Mbps by injecting traffic bursts with arrivals modelled as $Poisson(0.1)$, height (in Mbps) and duration (in seconds) as $Uniform[0,5]$. Figure 3.4 shows the performance of the two approaches for different parameter setups with $th1, th2$ in $[10\%, 20\%, ..100\%]$; m, d in $[2,4,..10]$; N in $[2,3,...30]$; α in $[1,2,3]$. For each configuration, the accuracy is quantified using the Root Mean Square Error (RMSE) between the value collected by monitoring and the ground-truth, while the resource consumption is given by the average switch query rate. The results are shown in Figure 3.4.

The main observation is that, for both approaches, different setups can produce widely different outcomes in terms of precision, as depicted in Figure 3.4.a. More surprisingly, Figure 3.4.b shows that different parameter configurations can lead to significantly different accuracy levels even when consuming exactly the same amount of resources. For instance, with a mean polling rate 0.5 Hz, the TAM approach can either produce RMSE=1.9 or RMSE=1.5. At the same time, different resource usage can result in the same monitoring precision, *e.g.,* RMSE=1.4 can be obtained with both 1 and 2 Hz average query rate with the PAM approach. In all cases, the performance with respect to the accuracy *vs.* query rate tradeoff strictly depends on how well parameters are tuned. In practice, however, determining the optimal setups is hard. This not only requires long (preliminary) periods of traffic observation but also necessitates adjusting the setups to match emerging traffic characteristics

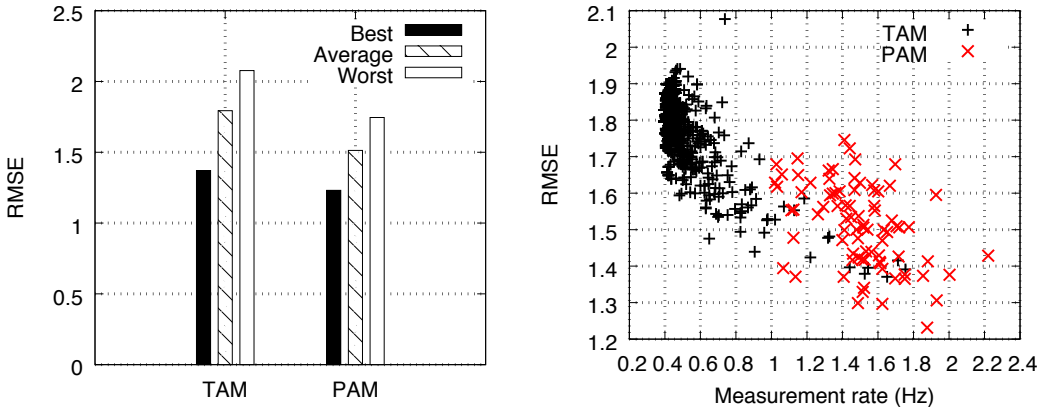


Figure 3.4: Performance of previous adaptive switch-polling techniques: monitoring precision (left) and accuracy/resources results (right)

(*e.g.,* new traffic burst patterns).

Below, a novel self-tuning adaptive monitoring approach is presented, which addresses the limitations of existing solutions. The proposed approach can produce precise reconfigurations of the measurement rate, between boundaries f_{min} and f_{max}, with minimal parameter tuning effort.

3.4.2 SAM algorithm

The proposed Self-tuning Adaptive Monitoring (SAM) approach allows to achieve the right trade-off between accuracy and consumed resources, without the need of manually performing complex parameter tuning according to the traffic characteristics. It works by automatically refining the algorithm parameters based on the evolution of the traffic shape.

More specifically, the objective of the proposed solution is to continuously adapt the timeout T, *i.e.,* the time to the next measurement. In a similar fashion to prediction-based approaches [23], SAM uses linear prediction to predict the next value x_p of a variable x based on the value of its previous measurements. When the new value x_{n+1} of x is collected, the normalised deviation D of the predicted variation $(x_p - x_n)$ from the real one $(x_{n+1} - x_n)$ is computed. When the value of D is negative, the prediction of the variation of x is underestimated. For instance $D = -0.5$ indicates that the predicted variation is 50% of the real one. In this case, the behaviour of x is more dynamic than expected and T is reduced proportionally to D, so that the larger deviation, the faster the query rate. In contrast, when D is positive, the prediction is overestimating the variation of x, *e.g.,* $D = 1.0$ corresponds to 100% overestimation. In this case, x is changing less (or less quickly) than expected (the behaviour of x tends to become more stable) and T is increased proportionally to D.

After each measurement, the linear prediction is automatically reconfigured by tuning the length N of the sample queue used to compute x_p based on an Additive Increase Multiplicative Decrease (AIMD) scheme. In particular, N is increased by one when $T_{new} > T$ and halved otherwise. Intuitively, the length is shortened when traffic becomes more dynamic (*e.g.,* when a traffic burst starts) so that the next decision on T only involves the most recent history of x. It is progressively expanded

Algorithm 1: COMPUTE NEXT QUERY TIMEOUT

Input: Current timeout T, Current sample queue length N
Output: Next timeout T_{new}, New sample queue length N_{new}

Compute prediction x_p: $x_p = x_n + \frac{t_n - t_{n-1}}{n-1} \sum_{i=1}^{N-1} \frac{x_{i+1} - x_i}{t_{i+1} - t_i}$
Retrieve value x_{n+1}
Compute deviation of $(x_p - x_n)$ from $(x_{n+1} - x_n)$: $D = \frac{(x_p - x_n) - (x_{n+1} - x_n)}{x_{n+1} - x_n} = \frac{x_p - x_{n+1}}{x_{n+1} - x_n}$
if $D < 0$ **then**
    $T_{new} = max(\frac{1}{f_{max}}, T_{old} - D \cdot T_{old})$
else
    $T_{new} = min(\frac{1}{f_{min}}, T_{old} + D \cdot T_{old})$
Add x_{n+1} to sample queue
Update sample queue $N_{new} = AIMD(N)$
return T_{new}, N_{new}

as the behaviour of x becomes more stable.

The pseudo-code of the proposed algorithm is shown in Alg. 1. It takes as input the latest timeout T and the current sample queue length N, and returns as output the time to the next measurement T_{new} and the new sample queue length N_{new}.

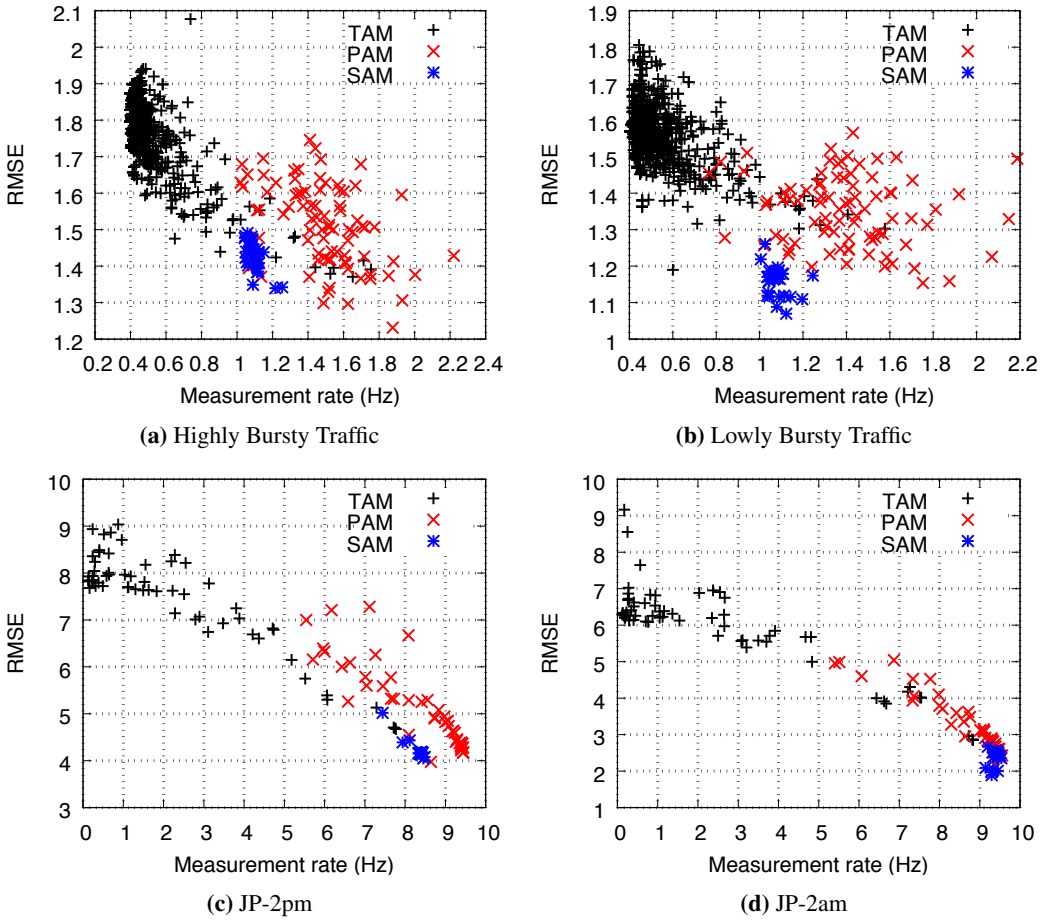
In terms of complexity, the proposed algorithm is similar to PAM, since both approaches require updating the predictor x_p and computing the deviation between the predicted variation of x and the real one. The main advantage of SAM, however, is that it requires minimal tuning effort. The only parameter needed for the algorithm setup is the initial value of N, whose impact is strictly limited to the algorithm startup phase. Differently from SAM, TAM requires the tuning of thresholds th_1, th_2 and multipliers m, d, and PAM the one of sample queue size N and factor α.

An additional advantage of SAM is that, unlike previous approaches [83] [23] for which the switch query period is only modified when large variations of x are observed, the proposed solution continuously adjusts the timeout T. Although this may lead to rescheduling measurement tasks more frequently (hence incurring thus increased burden on the monitoring module scheduler), it can prevent situations where the variations of x are undetected, for instance when the fluctuations of x are periodic and "phase-locked" [96] to the switch polling rate (*i.e.,* same frequency but out of phase).

3.4.3 Performance of SAM algorithm

The performance of SAM is evaluated by implementing it to measure the bitrate of traffic generated using both synthetic and real traffic traces and comparing the results to the ones obtained with TAM and PAM. To test the performance under different traffic conditions, two synthetic traces are used, with the same duration (5 minutes) and bitrate range (between 0 and 10 Mbps) but different levels of burstiness.

The first trace, referred to as *Highly Bursty Traffic*, emulates bursty conditions with traffic burst arrival modelled as $Poisson(0.1)$ (on average one in 10 seconds), burst height in Mbps as



(a) Highly Bursty Traffic

(b) Lowly Bursty Traffic

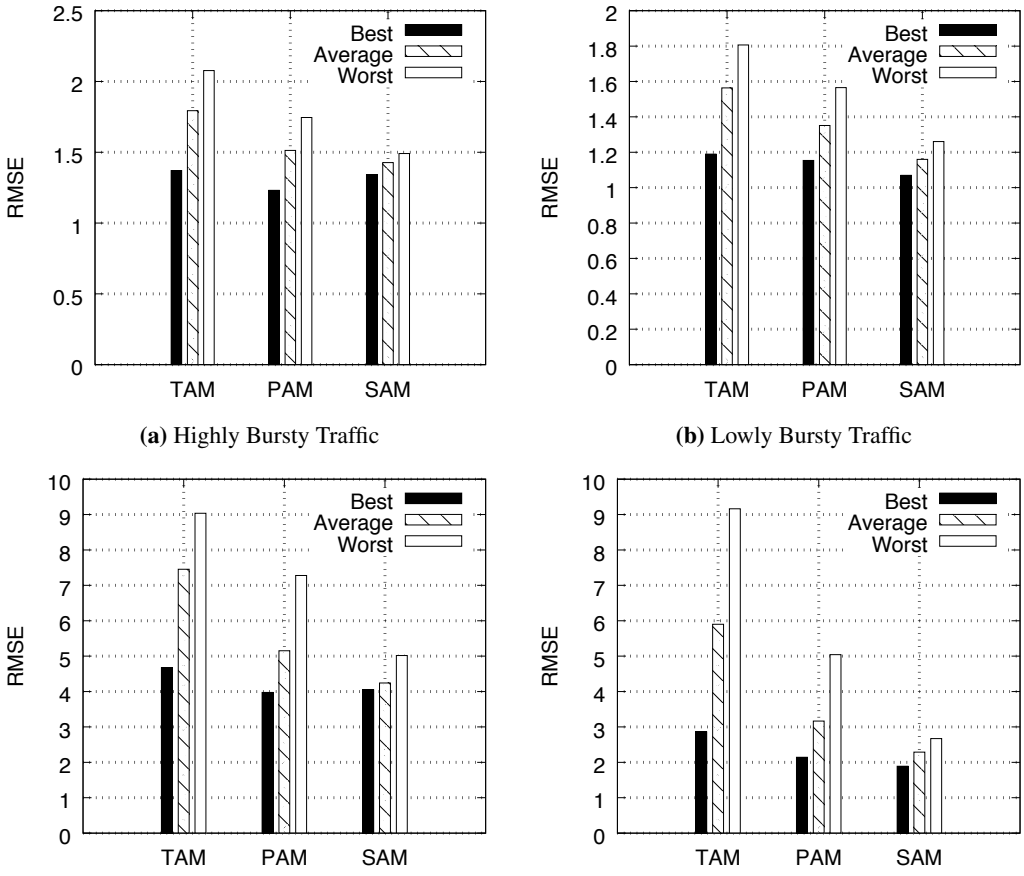
(c) JP-2pm

(d) JP-2am

Figure 3.5: Monitoring precision (RMSE) vs resource consumption (average polling frequency)

$Uniform[0,10]$ and burst duration in seconds as $Uniform[0,1]$. The second trace, referred to as *Lowly Bursty Traffic*, corresponds to more stable traffic with burst arrival modelled as $Poisson(0.02)$, burst height as $Uniform[0,5]$ and bust duration as $Uniform[1,10]$. Compared to the first profile, bursts in the second trace are less frequent, have a longer duration and exhibit smaller variation of the traffic rate. In addition to the synthetic traces, two real 15 minute traffic-packet traces from a 100 Mbps link of a Japanese operator [97] are used, representing peak time (JP-2pm) and off-peak time (JP-2am) traffic, respectively.

For all traces the performance is evaluated based on a wide range of parameter setups. In particular, for the TAM approach all combinations of thresholds $th1, th2$ in $[10\%, 20\%, ..100\%]$ are used, with $th1 > th2$, and query rate constant factors m, d in $[2, 4, ..10]$. For the PAM approach, sample queue length N in $[2, 3, ...30]$ and the constant α in $[1, 2, 3]$ are selected. Finally, for the proposed self-tuning approach the initial value of N is varied in the range $[2, 3, ...30]$. The same minimum (f_{min}) and maximum (f_{max}) query rate boundaries are applied for the three approaches, with $f_{min} = 0.1Hz$ and $f_{max} = 10Hz$. In a similar fashion to the example in Figure 3.4, the two main performance indicators used are the monitoring precision, given by the RMSE with respect to the



(a) Highly Bursty Traffic

(b) Lowly Bursty Traffic

(c) JP-2pm

(d) JP-2am

Figure 3.6: Monitoring precision comparison (Best-case, Worst-case and Average RMSE)

ground-truth traffic rate, and the resource consumption, represented by the average switch query rate over each experiment run.

The results are depicted in Fig. 3.5 and 3.6. Figure 3.5 shows that the performance of the self-tuning approach is, in general, more predictable in terms of precision and resource consumption compared to the two state-of-the-art solutions. For example, the RMSE obtained with the SAM approach only oscillates between 1.35 and 1.5 for the first trace (Fig. 3.6a), and between 1.05 and 1.25 for the second one (Fig. 3.6b) while, for PAM, the distance between the best and worst case accuracy is twice the one obtained with the SAM algorithm, and it can even be three times greater for TAM. The same applies to the average switch query rate. For example, for the first trace (Fig. 3.5a) the proposed solution queries the switch with mean rate between $1Hz$ and $1.5Hz$, while the two state-of-the-art approaches operate with an average query frequency between $0.5Hz$ and $2.5Hz$. Similar observations can be made for the experiments with real traffic profiles (Fig. 3.6c and Fig. 3.6d), where SAM approach achieves more predictable results compared to previous solutions.

Another important observation is that the proposed algorithm can achieve a good tradeoff between accuracy and resource consumption under different traffic profiles. In particular, the average

monitoring accuracy obtained by SAM generally lies close to the *best-case* precision of the PAM approach, always using a lower or, in the worst case, similar amount of resources. For the Lowly Bursty Traffic trace (Fig. 3.5b), lower values of RMSE with respect to TAM and PAM are obtained with a reduced average query rate, while for the case of the Highly Bursty Traffic trace shown in Fig. 3.5a, PAM can still achieve a slightly lower error but by consuming substantially more resources (by 50% or more). In the case of the peak time real packet trace (JP-2pm – Fig. 3.5c), the proposed algorithm obtains similar performance in terms of precision compared to the best results of PAM but by consuming approximately 10% less resources. Finally, in the case of the off-peak time real packet trace (JP-2am – Fig. 3.5d), results are in line with the most precise results obtained with PAM.

Summary of results and discussion To summarise, SAM introduces two key benefits with respect to state-of-the-art approaches. The first one is the reduced variability of the results on both monitoring precision and resource utilisation. On average, for the resource consumption (quantified by the mean switch polling rate), the range of SAM results is 6x (5x) smaller than for TAM (PAM). At the same time, the results on SAM monitoring accuracy fall in a range that is 4x (3x) smaller than for TAM (PAM). This is due to the large dependence of TAM/PAM performance on the algorithm parameter selection, which does not apply to SAM due to its self-tuning design. The second benefit of SAM, which results from the automatic algorithm re-tuning based on traffic dynamics, is the improved accuracy/resource tradeoff compared to the existing approaches. On average, SAM achieves a monitoring precision 14% higher than TAM best-case result (*i.e.,* the maximum monitoring precision, obtained from the best parameter selection) and only 2% lower than PAM best case, while consuming lower amounts of resources compared to TAM and PAM best cases. It should be noted that SAM average precision does not generally exceed PAM best-case one due to the similar, prediction-based nature of the two approaches.

As shown in Fig.3.6, SAM monitoring error (the RMSE in Mbps with respect to the ground-truth) increases with higher traffic speeds. When moving from 10 Mbps (Fig. 3.6a, 3.6b) to 100 Mbps (Fig. 3.6c, 3.6d) maximum rate, higher RMSE levels are obtained. Applying SAM on multi-Gbps line rates would unavoidably produce higher Mbps deviations between SAM monitoring results and the ground-truth. As a result, this would penalise the effectiveness of management applications that require high (e.g., Mbps-level) precision while operating on large (e.g., multi-Gbps) traffic aggregates. However, a different trend is obtained when considering the normalised error. In particular, the Normalised Root Mean Square Error is on average the 12% for 10 Mbps synthetic traffic (Highly/Lowly Bursty Traffic traces), and only the 3% for 100 Mbps real traces (JP-2am/pm traces). In other words, the percent deviation between SAM monitoring results and the ground-truth does not increase with the traffic rate, and it is higher for the synthetic traces since they stress-test SAM with bursty traffic patterns.

3.5 Use case scenario

To demonstrate the capabilities of the proposed monitoring framework, a distributed SDN environment is considered on which two different services are deployed by the network operator. The first one is a *content distribution* service, for which a set of content items is cached within the network. The network management system periodically updates (*e.g.,* in the order of hours) the content placement and the paths between user locations and content servers. Following an approach similar to the one proposed in [107], it reconfigures the routing of user requests in real-time by selecting an appropriate *path* between the user location and one of the available content servers based on the current path utilisation.

The second service provided by the operator is an on-demand gaming (*cloud gaming*) service, in a similar fashion to well-known platforms such as Gaikai [108]. The network provider offers specialised hardware resources, such as GPUs and fast memory, to support the computation required for the user game experience, which is offloaded from the end-user devices. From the network perspective, this service implies a continuous interaction between the user device and the server, where the client sends new input data, and in response it receives chunks of the video stream to be reproduced on the user device. The network management system can tune this service at run time by reconfiguring the routing of client and server traffic, which is performed by selecting a suitable path from a set of available options. While doing so, the operator objective is to avoid congestion in the network and, as such, to prevent potential Quality of Experience (QoE) degradation. Network congestion will increase the content delivery times and can lead to user dissatisfaction due to unresponsive client-server interactions in the on-demand gaming service.

When reconfigurations of the two services are generated in response to congestion episodes, the latest decisions on server and path selection are enforced using a method similar to the one proposed in [109]. The programmable (*e.g.,* OpenFlow-enabled) forwarding hardware at the edge of the network is instructed in real-time to rewrite fields of the IP packet header (*e.g.,* the destination address) in order to redirect traffic transparently to clients and servers. The enforcement of the path selection decisions is part of the header rewriting operations. For example, the path selection can be encoded in the packet ToS field.

Figure 3.7 exemplifies the use case. The forwarding nodes are partitioned in clusters, each under the control of a local manager. Each manager hosts an instance of a distributed Load Balancing (LB) application, which implements the necessary logic for reconfiguring, on a per user location, the content server from which a requested content is retrieved (for content distribution) and the path through which client/server traffic is delivered (for content distribution and cloud gaming). For simplicity, it is assumed that clients and servers to communicate on symmetric paths. For each network edge switch mapped to a user location, the application keeps a list of available setups $\langle Server, Path \rangle$ indicating how traffic should be routed. Each LB instance operates periodically on

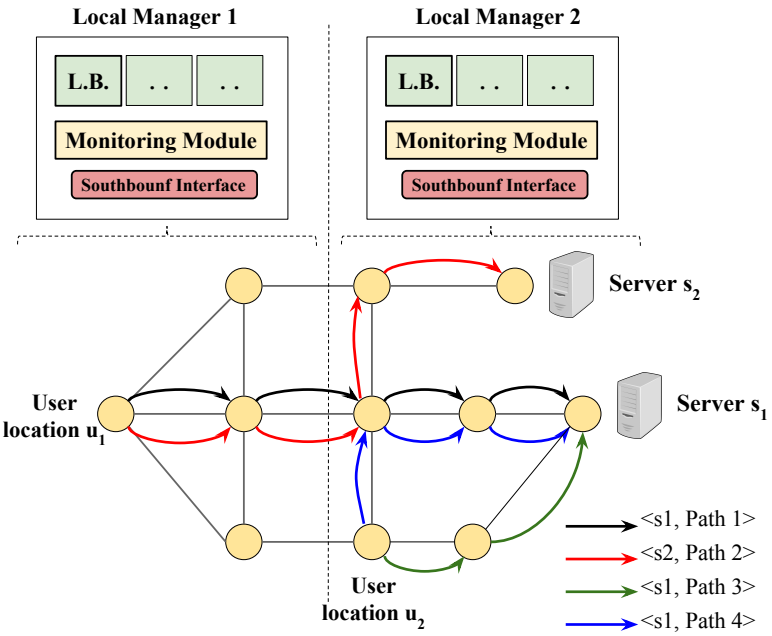


Figure 3.7: Use case scenario: distributed Load Balancing (LB) application

a short timescale, *i.e.,* every few seconds, and at each execution it obtains two statistics from the monitoring system.

The first statistic is the *average link utilisation* for the links included in the local paths, *i.e.,* the paths emanating from a client location within the scope of the relevant LM. Statistics for these links are directly collected and exposed by the underlying MM at each execution of the application. The monitoring of remote links is delegated to the LB instance operating on the corresponding network partition. This registers the relevant monitoring requirements on its MM, and periodically synchronises the results with the other LB instances.

The second statistic is the *average rate* of traffic originated by users in the local network partition. More specifically, each LB instance obtains the average throughput of all the flows matching the source IP address of one of the clients and the destination IP address of one of the servers, or vice versa. This measurement is used to determine the volume of traffic by which congested paths can be offloaded.

In case of link congestion (*e.g.,* average utilisation exceeding a predefined threshold), the LB application is responsible for offloading part of the traffic from the congested link in order to bring its utilisation below the threshold. Some flows are removed from the congested paths (paths including the congested link) and are (equally) assigned to alternative, non-congested, options represented by the 2-tuple $\langle Server, Path \rangle$. The new configurations are enforced on the ingress OpenFlow switches.

If a congested path spans multiple network partitions, the corresponding LB instances operate iteratively, as in the solution presented in [110]. The first decision is taken by the LB instance directly associated with the congested link based on the bandwidth availability on the alternative paths. The

result is then communicated to the next LB instance until the process terminates.

3.6 Evaluation

In this section, the benefits of the proposed decentralised monitoring framework are investigated. The evaluation is based on the use case described in Section 3.5 and focuses on the performance in terms of latency and traffic overhead, as well as on the impact on the two different services. Furthermore, it investigates the gain that can be achieved by applying SAM, the self-tuning adaptive monitoring solution presented in Section 3.4.

More specifically, the evaluation is conducted as follows. In Section 3.6.3 the performance of the proposed decentralised architecture is compared, in terms of monitoring latencies as well as traffic overhead, to the one obtained with a centralised solution. Sections 3.6.4 and 3.6.5 explore how the monitoring information extraction and the monitoring information distribution functions can affect the performance of the use case services. Considering monitoring information extraction, the focus is on the improved monitoring efficiency derived from the SAM algorithm presented in Section 3.4. Regarding monitoring information distribution, the possible tradeoff(s) between application performance and monitoring scalability-overhead are explored, following an approach similar to the one in [20].

3.6.1 Network testbed and Local Manager implementation

Experiments are performed on the testbed represented in Fig. 3.8. This relies on *Mininet* [111] [112] [113] to emulate the network topology, including hosts, *i.e.,* clients and content servers, and OpenFlow switches. The advantage of using Mininet is that it allows to create a realistic virtual network, running real kernel, switch and application code on a single machine or virtual machine. In the emulated Mininet network, hosts (clients and servers) are lightweight Linux containers, while network devices are instances of a virtual switch. Specifically, the widely-adopted *Open vSwitch* virtual switch [51] is selected.

The Local Manager, including the Monitoring Module and the Load Balancing application logic, is implemented as a set of Python modules. A small set of APIs from the SDN controller POX [98] is also reused to implement the southbound interface functionality, which includes OpenFlow message crafting and parsing, and switch-controller communication primitives. Furthermore, both the switch-controller channels and the local manager synchronisation interfaces are based on TCP sockets established at network initialisation.

3.6.2 Experiment setup

Experiments are performed on the two network topologies, *Topo1* and *Topo2*, summarised in Table 3.1, where each node is an OpenFlow-enabled switch, and all links have 10Mbps bandwidth. In *Topo1* the average link latency is $5ms$, while the end-to-end latencies (round-trip) fall in the range $[25ms, 70ms]$. In the case of Topo2, the link latencies are artificially tuned in a way that allows us to

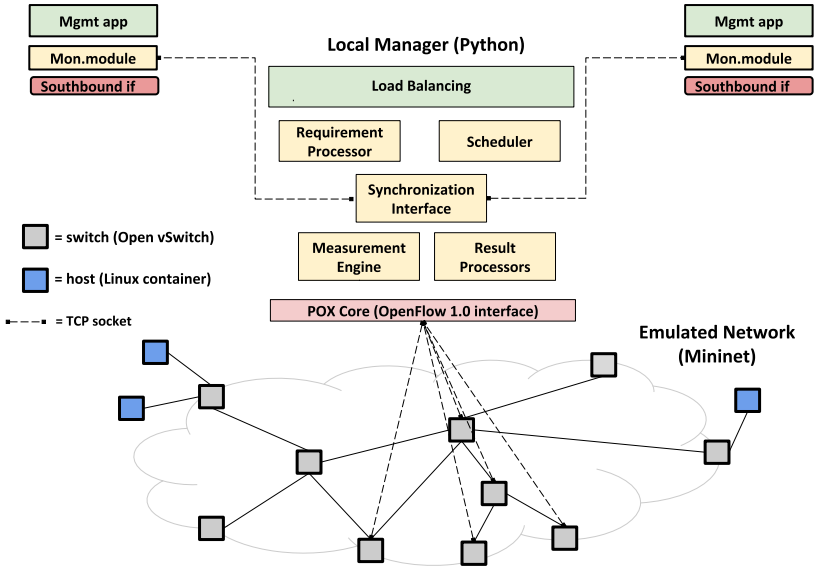


Figure 3.8: Implementation of the framework on a Mininet-based virtual network testbed

Table 3.1: Network characteristics

	Network	# Nodes	# Bidirectional Links
Topo1	Geant [99]	23	37
Topo2	Germany50 [100]	50	88

experiment with increased end-to-end delays, with round trip latencies between 100*ms* and 150*ms*.

In both topologies, clients are distributed over five user locations. Each client can reach two servers, each being accessed using three alternative paths. By default, all clients are initially assigned to the shortest path in terms of hop count. Each experiment has a duration of 5 minutes and is preceded by a short startup phase in which paths are installed and the MM and LB application initialised. The placement of LMs is provided as an input and used to compute the relevant hop-count and corresponding latencies between pairs of LMs.

For each experiment, only one of the two use case services is emulated. In the case of *content distribution*, each client generates content requests following a pattern derived from the one used in [101]. The content size is scaled down in accordance to the reduced link bandwidth. In the case of *cloud gaming*, the corresponding network traffic is directly emulated by generating separate client and server-originated packet streams. In particular, the results reported in [102] are used to configure the average packet size and packet rate for both upstream (*i.e.,* client-originated) and downstream (*i.e.,* server-originated) traffic.

The LB application reconfigures the flow routing in case of link congestion, as described in Section 3.5, based on local knowledge, including the utilisation of local links and the throughput of local traffic flows (if any), as well as information about remote links, which is accessed through periodic synchronisation. For simplicity, all LB instances run simultaneously (same frequency and

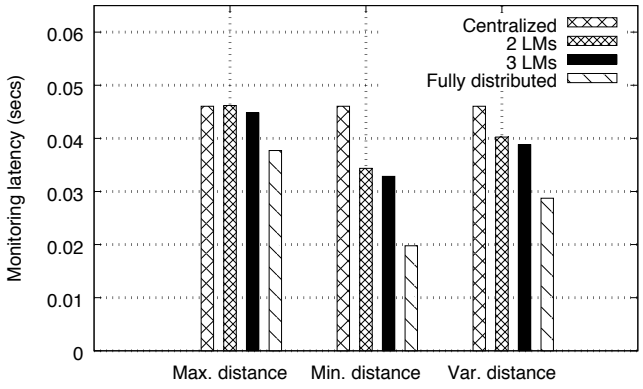
clock reference). For each experiment two parameters are configured: p_l is the period of local measurements performed by each MM, and p_s the synchronisation period of link status between the LB instances, with $p_s \geq p_l$. For both services, link congestion is generated by creating spikes of user demand, which is achieved by increasing the number of clients over the experiment time. In addition, fixed-rate UDP flows are used, generated with *Iperf* between the client and the server locations to emulate background traffic.

Another key parameter is the congestion threshold, which has a direct impact on the flow (re)scheduling operated by LB. It is set to 85% of the link capacity in accordance to [103]. Such settings allow to avoid excessive route flapping and to keep the average period of route reconfigurations at least one order of magnitude higher than the content download times. These values are in line with traditional end-user redirection practices [104].

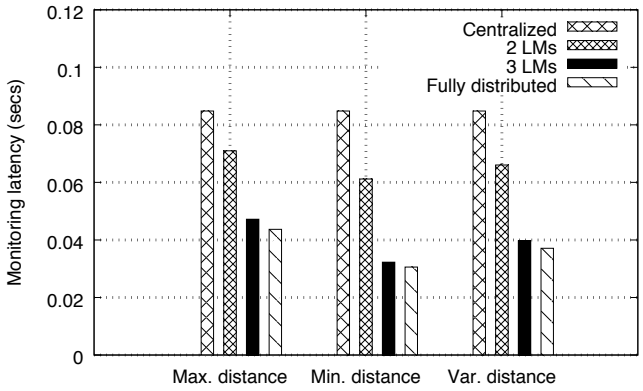
3.6.3 Performance of decentralised monitoring

In this subsection, the proposed decentralised monitoring approach is compared with a centralised solution where the full state of the network is collected by a single management entity. The initial focus is on the monitoring latency, a measure of reactivity, defined as the delay between the time the measurement starts (*e.g.,* the corresponding procedure is selected by the scheduler) and the time the requested information is made available to the LB instance performing the flow routing reconfigurations. The monitoring latency is evaluated for the link utilisation measurements in 4 different setups: *Centralised* (single manager), *2 LMs*, *3 LMs* and *Fully distributed*, in which one LM is assigned to every single switch. For the centralised, 2 LMs and 3 LMs cases, 10 experiments are performed, each with a different manager allocation, and average the results. The condition $p_s = p_l$ is fixed, *i.e.,* the link status is synchronised between LB instances with every new measurement, and each LM is configured to synchronise with every other LM in the topology.

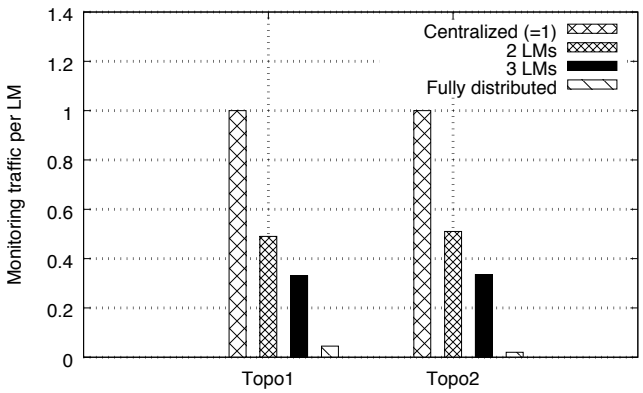
Figure 3.9 depicts the average monitoring latency for 3 cases: *i) Minimum distance*: reconfigurations are computed close to where raw statistics are extracted, *i.e.,* by the closest LM; *ii) Maximum distance*: reconfigurations are computed by the farthest LM from where the statistics are collected and *iii) Variable distance*: reconfigurations are computed with the same probability by any of the available LMs. As can be observed, the performance obtained with the *Centralised* setup (baseline scenario) is almost constant as the monitoring information is always processed at the central manager independently from where the statistics are gathered. For the decentralised setups, a significant delay reduction for *minimum distance* in comparison to the centralised scenario can be observed. The reduction is up to 57% for *Topo1*, and it is even more evident (61%) for *Topo2*, where paths generally span higher latencies and number of hops. Smaller latency reductions can be noticed for *maximum distance*, up to 17% for $Topo1$ and 40% for *Topo2*. As expected, the higher the percentage of reconfigurations computed close to where the relevant knowledge is collected, the higher the reduction in terms of control-loop delays achieved with the decentralised approach.



(a) Monitoring latency, Topo1



(b) Monitoring latency, Topo2



(c) Monitoring traffic per LM

Figure 3.9: Performance of the decentralised monitoring approach

In addition, the total amount of monitoring traffic handled by an individual LM is evaluated for the different decentralised setups. Results reported in Figure 3.9c are normalised to the ones obtained in the *Centralised* case. As can be observed, the decentralised approach can drastically reduce the burden on the single LM since the incoming monitoring traffic decreases, as expected, proportionally to the number of LMs deployed in the network.

3.6.4 Monitoring information extraction

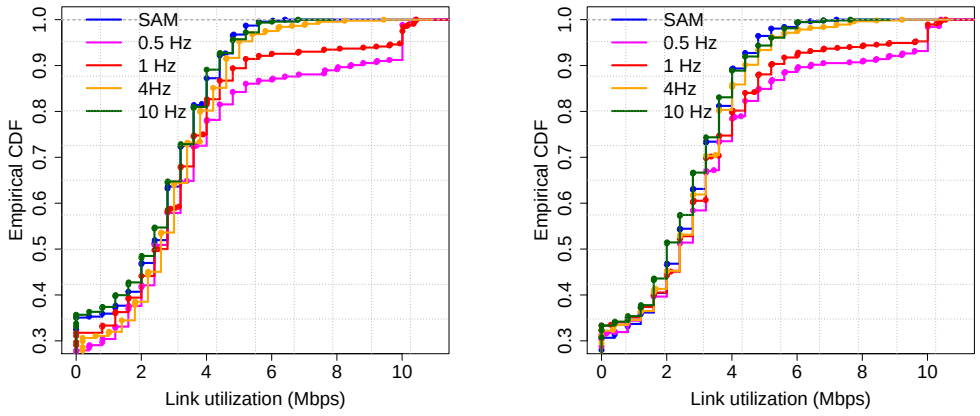
This subsection investigates how the extraction of monitoring information by the monitoring modules can affect the performance of the use case services (content distribution and cloud gaming) and the LB application presented in Section 3.5. In particular, it compares a baseline solution where the switches are queried at a fixed rate with the self-tuning adaptive monitoring algorithm (SAM) More specifically, it extends the analysis of the performance of the SAM approach by focusing on the trade off between application/service performance and monitoring overhead.

To quantify the performance of the LB application, as well as the impact on content distribution and cloud gaming, three different metrics are taken into account. The performance of the LB management application is represented by the *utilisation* of the congested link l, obtained by sampling it at rate $1/p_l$ (*i.e.,*, every 1 second) throughout the duration of the experiment. For the content distribution service, the *download speed* is collected as a measure of the user's QoE. Finally, the performance of cloud gaming is evaluated by measuring the *service latency*, *i.e.,* the response time of each individual user request, which determines how responsive the gaming service is. To guarantee acceptable performance, the service latency should not exceed $80ms$ for highly interactive games and $150ms$ for slow-paced games [105].

To represent the monitoring overhead, the *mean monitoring traffic* rate produced by individual switches is considered. This metric is in line with [22] [21] that show that only a limited amount of monitoring data can be reported by SDN-enabled switches for each time unit.

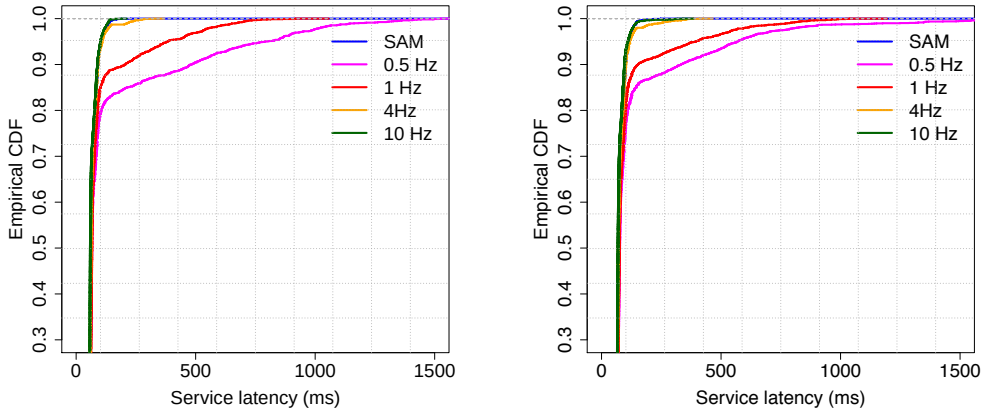
The evaluation is performed by running experiments with the *Fully Distributed* setup in which congestion episodes on a specific network link are generated. For each experiment, monitoring is configured to query the switch either at a fixed rate (with frequency $0.5Hz$, $1Hz$ or $10Hz$) or using SAM with query rate varying in the range $[0.1Hz, 10Hz]$. The condition $p_s = p_l$ is also fixed, so that measurement results are always synchronised between the LMs. For each test run, download speed and cloud gaming latency values are recorded for all clients in the network. Also, for the results of different clients not to be affected by the different path latencies, the selected users are served in each topology through paths of equal length.

Figure 3.10, 3.11, and 3.12 show the empirical CDF of *link utilisation* of the congested link for the different monitoring configurations, the cloud gaming *service latency* and the *download speed* of content distribution, respectively. In addition, the mean monitoring traffic (*i.e.,* the monitoring overhead) is reported, normalised to the one obtained with SAM. As depicted in the figure, the values of the three performance indicators improve when increasing the measurement rate as congestion episodes can be detected more reactively. Interestingly, it can be observed that the performance obtained with SAM can generally match the one obtained with fixed $10Hz$ monitoring. For example, as shown in Figure 3.11, the service latency never exceeds the $150ms$ threshold in both the SAM and fixed $10Hz$ monitoring cases. The SAM approach does however produce less than 50% of monitoring traffic compared to fixed $10Hz$ monitoring, as shown in Figure 3.13. This is because



(a) Topo1 **(b)** Topo2

Figure 3.10: Extraction of monitoring information: impact on link utilisation

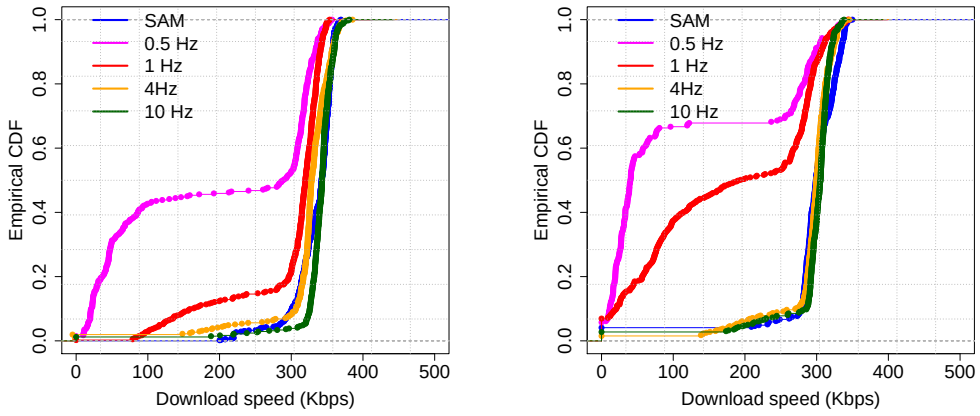


(a) Topo1 **(b)** Topo2

Figure 3.11: Extraction of monitoring information: impact on service latency in cloud gaming

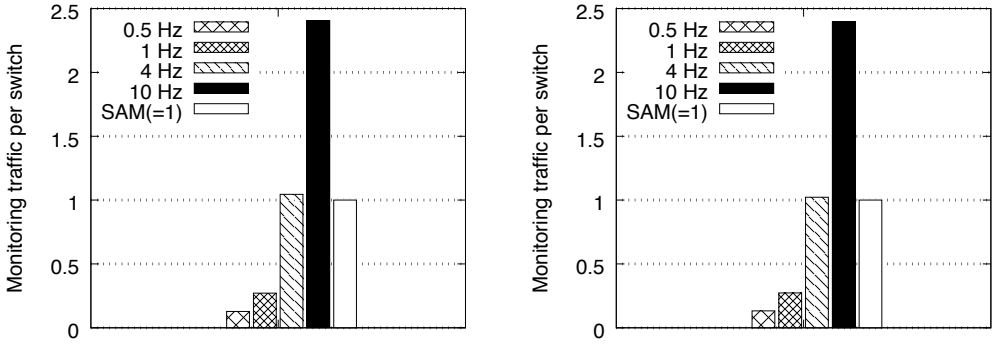
the adaptive monitoring logic of SAM increases the query rate up to $10Hz$ only when needed, *i.e.,* when congestion arises. The only case in which $10Hz$ monitoring can outperform SAM is for the download speed but yet the difference in performance is never substantial. For instance, in *Topo1*, where the difference is more noticeable, the download speed obtained with SAM is lower in only 20% of the cases, and the speed reduction never exceeds $40Kbps$, which represents less than 12% of the maximum speed. This small benefit of fixed $10Hz$ monitoring comes furthermore at a huge cost as 150% more monitoring traffic is produced.

Finally, the evaluation also considers the case where the operator adopts a fixed 4Hz measurement frequency, thus *guessing* the same average query rate of SAM (Fig. 3.13). In practice, this is unlikely to happen, as no prior knowledge on a suitable measurement rate is usually available to the operator. As shown in Fig. 3.11, 3.12 even in this case SAM is not outperformed.



(a) Topo1 **(b)** Topo2

Figure 3.12: Extraction of monitoring information: impact on download speed in content distribution



(a) Topo1 **(b)** Topo2

Figure 3.13: Extraction of monitoring information: monitoring traffic per switch with respect to SAM (=1)

3.6.5 Monitoring information distribution

In the considered decentralised management framework [1], instances of a distributed application can take decisions based on information extracted at a remote location. This subsection investigates the effects of the dissemination of monitoring information between LMs on the performance of the LB application and its impact on the content distribution and cloud gaming services. Experiments have been executed with the *Fully Distributed* setup in which congestion episodes occur on a specific link l, located under the scope of a specific LB instance (local LB), while the offloading decisions are made outside the local partition by another application instance (remote LB).

In these experiments, $p_l = 1$ second is selected while p_s varies in the range $(1, 10)$ seconds in order to increase the inconsistency between the views of the two LB instances. The performance results below are shown in the form of boxplots, with the whiskers extending from the box (first and third quartile boundaries) to the 95 percentiles.

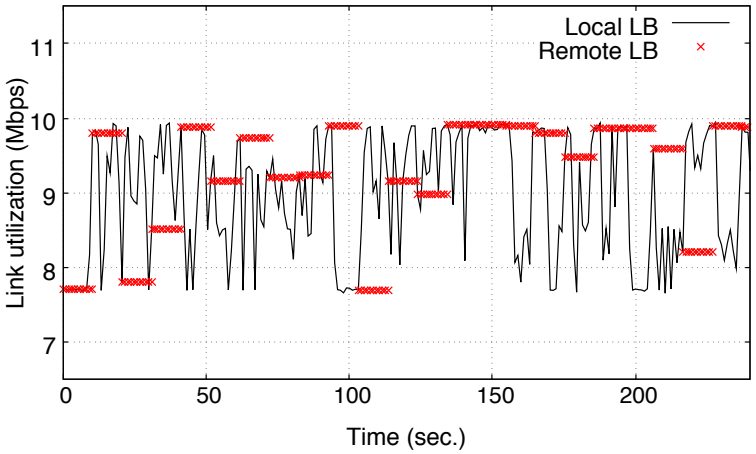


Figure 3.14: Link utilisation time-series at Local LB and Remote LB for $p_s = 10$.

Info. Distr. Period p_s(sec.)	Avg Util. Error	RMSE
2	0.44501	0.78
4	0.65137	1.0096
6	0.73237	1.0298
8	0.90245	1.1952
10	1.30	1.5448

Table 3.2: Average link utilisation error and RMSE vs synchronisation period

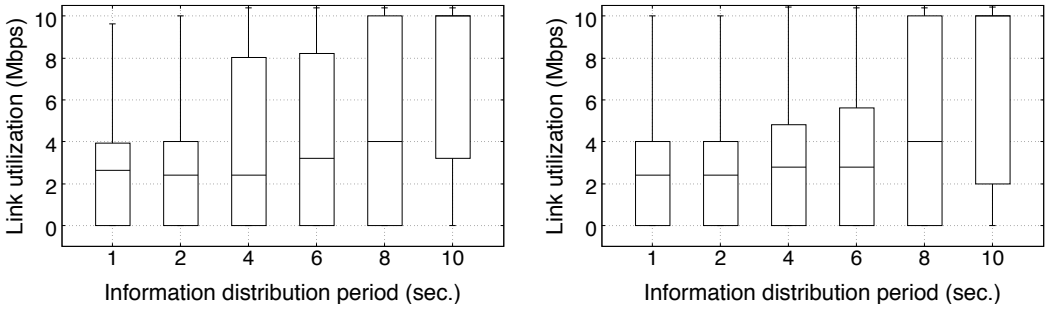
Synchronisation error Figure 3.14 illustrates the link utilisation exposed to the *local* and *remote* LB instances for the worst-case scenario, *i.e.* $p_s = 10$ seconds.

It can be observed that, with such a high synchronisation period, the *remote LB* fails to catch utilisation spikes of short duration. In this specific case, no action is performed for 53% of the congestion events. Table 3.2 reports the error between the local and remote LB views of link utilisation.

To measure the effect of relaxed synchronisation, the RMSE (root-mean-square error) is used, where a RMSE of 0 corresponds to perfect synchronization. As expected, the RMSE increases as the information dissemination period increases, which indicates an increase in terms of inconsistency between the views of the two LB instances.

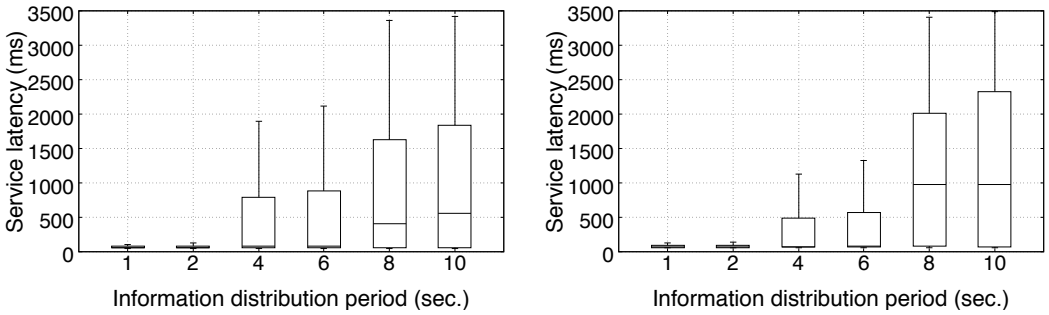
Impact of synchronisation on Link Utilisation To quantify how the network is affected by the inconsistent views of LB instances, the utilisation of link l is sampled at rate $1/p_l$ (*i.e.*, every 1 second) throughout the duration of the experiment. It can be observed from Fig. 3.15 that the utilisation of link l is significantly affected by the synchronisation period. For low values, such as $p_s = 1,2$, the utilisation is kept below the congestion threshold for more than 75% of the total experiment duration. Starting from $p_s = 4$, larger synchronisation errors (*i.e.,* higher values of RMSE in Table 3.2) lead to a noticeable increase of the utilisation for half of the collected samples. Finally, for high values of the information distribution period, *e.g.,* $p_s = 8,10$, link l is above the congestion threshold (85% of the link capacity) for more than 25% of the experiment time.

Impact of synchronisation on the application performance In the following, the effects of re-



(a) Topo1 **(b)** Topo2

Figure 3.15: Impact of relaxed synchronisation between LMs on link utilisation

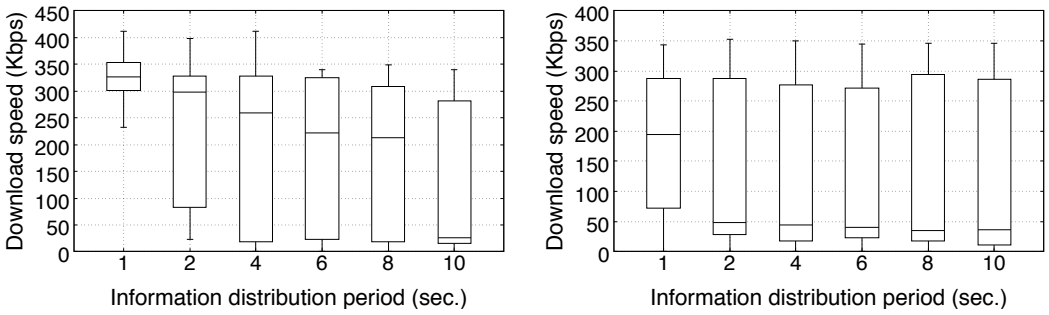


(a) Topo1 **(b)** Topo2

Figure 3.16: Impact of relaxed synchronisation between LMs on cloud gaming application performance

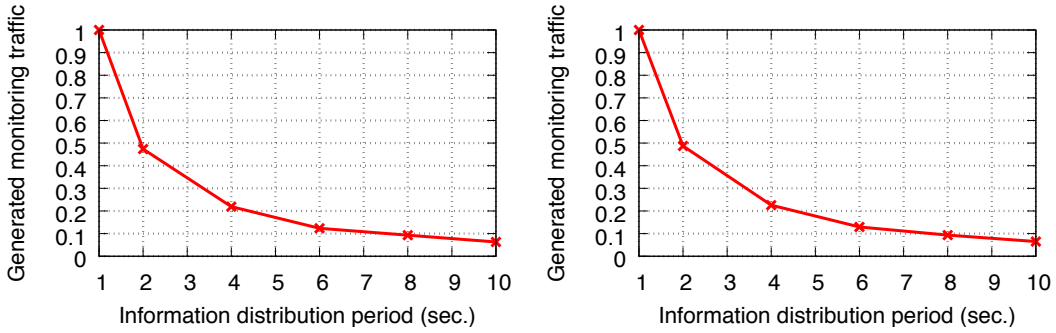
laxed synchronisation on the performance of the use case applications *content distribution* and *cloud gaming* is investigated. For the content distribution service, the performance metric is the download speed, used as a proxy for the user's perceived quality of experience. For cloud gaming, the performance is evaluated by measuring the service latency, i.e., the response time of each individual user request, which determines how responsive the service is. To guarantee acceptable performance, the service latency should not exceed 80ms for highly interactive games and 150ms for slow-paced games [105].

As generally shown by Fig. 3.16 and 3.17, monitoring information distribution clearly reflects on the user perceived quality for both use case services. In the content distribution case (Fig. 3.17), a reduction of the median download speed can be observed, which can be more or less progressive, *e.g.,* based on the duration of congestion episodes. The impact of relaxed synchronisation is particularly evident in *Topo2*, where the download rates are lower than in *Topo1* due to the higher end-to-end latencies in this network. For instance, increasing p_s from 1 second to 2 seconds leads to a substantial reduction of the median throughput that drastically decreases from 300 to 50 Kbps. In the cloud gaming case (Fig. 3.16), the effect of infrequent synchronisation can be even more disruptive. In particular, while an acceptable gaming experience can be guaranteed with $p_s = 1,2$ throughout all the experiment time (the service latency rarely exceeds $100ms$), under $p_s = 8,10$ the latency is above $500ms$ in *Topo1* and above $1sec$ in *Topo2* for approximately half of the cases. This



(a) Topo1 **(b)** Topo2

Figure 3.17: Impact of relaxed synchronisation between LMs on content distribution application performance



(a) Topo2 **(b)** Topo2

Figure 3.18: Monitoring information distribution overhead (generated monitoring traffic)

would in practice make the gaming service inaccessible to clients.

Synchronisation overhead In addition, the monitoring overhead is evaluated, which is defined for each experiment as the generated monitoring traffic, *e.g.,* sum of the size of each packet multiplied by the path length (number of hops). The overhead is the sum of two components: i) the *measurements* overhead, *i.e.,* the traffic incurred by the collection of the raw statistics from the physical devices, and ii) the traffic incurred by the distribution of the link status between the LB instances that linearly increases with the frequency of the information distribution. Only the latter is plotted in Fig. 3.18a and 3.18b since the generated traffic is in general dominated by the dissemination of monitoring information. As expected, increasing the synchronisation period can lead to substantial reductions of the overhead that decreases proportionally to p_s.

It can be finally observed that significant overhead reductions can be obtained by trading off service performance a little bit. For instance, the overhead can be approximately halved by choosing $p_s = 2$ instead of $p_s = 1$, while incurring negligible disruption on the cloud gaming performance, as shown in Fig. 3.16a and 3.16b.

3.7 Limitations

In this section, the main limitations of the proposed SDN monitoring framework are discussed. These relate to the polling-based nature of the monitoring approach presented, the cost incurred by SAM when applied to large numbers of variables (e.g., for measuring massive amounts of different flows), and the dissemination of monitoring information.

Polling-based monitoring approach The design of the proposed architecture is geared towards highly-configurable monitoring in terms of measurement frequency and granularity, which is important for satisfying the requirements of different management applications. One limitation, however, is that the key architectural components (Scheduler and Measurements Engine in particular) and also the SAM algorithm require the monitoring system to operate in *polling* mode, *i.e.,* with pulls of monitoring information whose execution is driven by network managers/controllers. This is the case of measurement approaches based on OpenFlow or SNMP (e.g., SNMP link-load measurements), or extracting measurement data temporary stored in P4 switches [54]. Such approaches are currently supported by the proposed architecture or can be supported with limited extension of the Monitoring Module components. On the contrary, approaches based on streaming (e.g., using OpenConfig [94] or sFlow [58]), where the monitoring data is continuously pushed by the switches, cannot be supported by the proposed framework. For example, one essential conditions for SAM is the full control over the times at which monitoring information is extracted from the switches, which does not apply to the case of streaming-based monitoring.

Cost of SAM To adapt the switch-polling period when measuring a network variable such as a flow rate or a link utilisation, SAM needs to compute a linear prediction and to decide on increments/decrements of the polling period. The computational cost of SAM can become a problem when large numbers of measurements (of different variables) need to be adaptively scheduled at the same time. In such a case, additional resources would be needed at the Local Manager to effectively cope with the monitoring demand in terms of CPU (to perform many SAM runs in short times) and memory (to keep the state of adaptive monitoring, e.g., sample queue used by the prediction and current polling period, for many different variables). However, it should be noted that the cost incurred by SAM is in line with PAM state-of-the-art approach [23] due to the similar, prediction-based nature of the algorithms.

Dissemination of monitoring information The impact of monitoring information synchronisation on the use-case service performance and on the monitoring-related overhead has been evaluated using different fixed synchronisation periods. Although the results showed that good performance/overhead tradeoffs can be achieved by selecting an appropriate information distribution period, any periodic synchronisation might still be outperformed by more advanced and adaptive approaches, for instance exchanging the new monitoring results only when they are substantially different from the ones previously synchronised. Despite the proposed architecture includes a Syn-

chronisation Interface enabling the development of advanced synchronisation solutions, the adaptive monitoring capability of the framework (*i.e.,* SAM) is currently restricted to the extraction of measurement data from the switches and does not involve the synchronisation of monitoring results.

3.8 State of the art overview

SDN monitoring frameworks The advent of SDN has empowered network monitoring with new measurement enablers as OpenFlow switches can keep track of active flows in the network and update per flow counters. A number of proposals have recently exploited this feature to provide direct and precise flow measurements without resorting to packet sampling. In OpenTM [60], the SDN controller pulls, at fixed intervals, the switch counters collected by explicitly polling the switches in order to periodically generate traffic matrices. In [27], the authors propose FlowSense, an approach where the network utilisation is measured using a different, push-based, approach. This uses the messages generated during the setup and eviction of flows from the switch flow table. Compared to the technique used in this chapter that leverages explicit switch polling, the solution in [27] can reduce the measurement overhead but suffers from limited flexibility since it only works with short-lived flows.

While most of the recent proposals have focused on specific measurements or on a very limited set of measurement tasks, the approaches presented in [34] and [83] provide a measurement API for supporting a wide range of tasks. OpenSketch [34] relies on a clean-slate approach where a novel processing pipeline is used on the switch to support many different measurement tasks. In addition, a library is developed for the control-plane to reconfigure the pipeline. Payless [83] resembles more the approach adopted in this chapter as it provides an API to serve different monitoring requests, all executed through pull-based measurements.

Distributed SDN solutions In contrast to the work presented in this chapter, all the aforementioned proposals are mainly tailored to early SDN solutions that rely on a physically centralised control infrastructure. This assumption has been questioned in [24] and [88] where distributed control planes have been proposed to overcome scalability issues such as processing bottlenecks at the central controller and large control latencies. However, the main focus of these papers is on how distributed controllers can unify their local views of the network, paying little attention to measurement issues. In [88] Tootoonchian *et al.* present a controller-to-controller communication mechanism based on a pub/sub paradigm. In [24], Koponen *et al.* propose a distributed database for the dissemination of slowly changing network state and a distributed hash table for exchanging volatile information. Another important work is [20], which investigates the main issues posed by state distribution in a logically centralised, physically distributed SDN architecture. One of these issues, *i.e.,* the tradeoff between performance optimality and state distribution overhead, has been considered in Section 3.6.5, which has evaluated how the timeliness of synchronised monitoring information impacts the MA performance and the overhead in terms of additional monitoring traffic. In less recent work

[91], the authors introduce a model, called A-Gap, for adaptive reduction of the traffic overhead in distributed monitoring based on filtering. However, this technique addresses a different, hierarchical, monitoring architecture where the information is aggregated and transmitted on a spanning tree toward a central management station.

Adaptive SDN monitoring Adaptive monitoring in SDN has recently attracted several research efforts. In Payless [83], monitoring adaptations are performed based on fixed thresholds. In [25], a model for dynamically updating the switch query timeout is presented, but it is exclusively tailored to flow covariance measurements. In a similar fashion to SAM, the approach in [23] reconfigures the query rate based on the outcome of a linear prediction. It does however require some complex parameter tuning, an issue that the proposed scheme overcomes through automatic reconfigurations of the algorithm parameters.

3.9 Summary

This chapter has presented a novel monitoring framework for software-defined networks that can provide heterogeneous management applications with frequent and consistent network state updates, thus enabling fast and effective resource reconfigurations. The proposed solution relies on a decentralised architecture satisfying the requirements of networks with a large number of geographically dispersed devices. To reduce the consumption of the switch control bandwidth, it performs frequent adaptations of the switch query rate using SAM novel self-adaptive monitoring method. As opposed to existing approaches for which complex parameter tuning is needed under highly dynamic network traffic, the proposed algorithm can automatically reconfigure itself without any intervention from the operator.

SAM has been shown to provide more predictable performance in terms of precision and resource consumption compared to state-of-the-art solutions and to achieve better accuracy/resource tradeoffs. Such results have been obtained with both synthetic and real traffic traces. Despite the limited number of traces used, it has been shown that SAM algorithm outperforms existing adaptive techniques under a range of dynamic traffic conditions in terms of burst inter-arrivals, durations and heights, including episodes of short-lived (e.g., sub-second) and multi-Mbps traffic spikes. Another key benefit of SAM algorithm is its broad applicability. Indeed, SAM can be integrated with a range of different monitoring designs (beyond the SDN monitoring framework presented in this chapter) that use switch polling for extracting measurement data, including for example SNMP-based monitoring solutions (e.g., SNMP link-load measurements [39]), since it does not depend on specific protocols (e.g., OpenFlow) nor specific network control/management architectures.

The benefits of the proposed decentralised monitoring framework have been investigated based on realistic topologies and demanding use case services. The evaluation has shown that the framework can improve the reconfiguration reactivity by significantly reducing the control-loop delays, in particular when a large portion of reconfiguration decisions are taken close to where the relevant

statistics are collected. In addition, it has demonstrated that service performance can significantly improve by enabling SAM, without incurring additional switch resource consumption. Finally, it has shown that although decentralising the monitoring functionality involves additional communication overhead, this can be mitigated by slightly relaxing the synchronisation of monitoring data, while maintaining acceptable service performance.

Chapter 4

Adaptive and Accuracy-Aware Monitoring for Software Dataplanes

4.1 Overview

The previous chapter has addressed key issues related to (i) the efficient extraction of measurement data from the dataplane (for the case of SDN-enabled switches) and (ii) the timely delivery of monitoring knowledge to management applications, targeting the demanding requirements of large-scale networks and management applications operating on short timescales. However, further challenges concerning the efficient and accurate monitoring of network traffic should be considered that relate to how concurrent measurement tasks can be collectively executed in the dataplane, especially under adverse operating conditions. These challenges are explored and addressed in this chapter.

Specifically, the case of a software dataplane is considered here, where traffic measurements are performed using *commodity hardware* and integrated with software packet-processing pipelines. This approach has been gaining increasing attention over the last years [8] [67] due to its improved flexibility and reduced cost. In particular, it enables the execution of sophisticated per-packet monitoring in real-time, *i.e.,* in line with high-speed traffic streams, thus enabling timely reports of fine granularity [8]

Monitoring systems adopting this approach are required to perform elaborate measurement operations in the dataplane on a per-packet basis, while processing all packets in time (*lossless* packet processing), which is an extremely challenging task. On one hand, monitoring systems need to cope with increasing data rates supported by network cards (10+ Gbps), which squeeze the admissible packet processing times to a few tens of nanoseconds. On the other hand, they should satisfy the operator's requirement of assigning limited resources to the monitoring process (*e.g.*, 1 processor core per 10 Gbps [8]) while performing advanced measurement tasks on a per-packet basis. Recent packet capture engines [15][16], hardware technologies such as Receive-Side Scaling (RSS)[71], and multi-core packet scheduling architectures [68][114] allow to cope well with packet capture at wire-speed. However, short-lived bottlenecks can still arise in the monitoring process, leading

to potential loss of packets in the input buffer(s). This occurs when unbalanced packet rate spikes, affecting one or a subset of CPU cores, compress the available per-packet time, or when variations of traffic skew [28], or concurrent access to shared server resources [115] inflate the packet-processing latencies.

A possible approach to limit the risk of packet loss is *overprovisioning*, *i.e.,* to allocate additional processor cores to traffic monitoring, so that each core would face a lower-speed packet stream, hence a large number of CPU cycles is reserved to each packet. An alternative solution is to restrict the available measurement-related operations to a minimal set such as byte and packet count only. These solutions however result to *inefficient use of the dataplane resources*.

These limitations are addressed in this chapter by introducing MONA, an adaptive traffic monitoring framework based on software packet-processing. MONA solves two key problems under increasing workload conditions in the monitoring pipeline. On the one hand, it guarantees resilience to bottlenecks by timely reconfiguring the monitoring operations under dynamic operating conditions. To this end, MONA performs frequent estimations of the available processing time, coupled with extensive offline analysis of the different per-packet latencies involved in the monitoring process, it timely reduces the monitoring operation sets for portions of the active flows' population. At the same time, MONA preserves the accuracy of monitoring reports, based on user-specified accuracy thresholds.

Jointly achieving zero packet loss (*i.e., all packets processed in time*) and accurate monitoring reports is a hard problem. In particular, it is difficult to know the impact of monitoring reconfigurations on the reports' accuracy *a-priori* [86][56], as this depends on traffic characteristics and on the monitoring operations logic. MONA overcomes these issues by decoupling the *adaptation* functionality (in face of bottlenecks) from the *accuracy control* one. The latter progressively redistributes subsets of the active traffic flows between the *measurement tasks* running in the system, so that monitoring reports can be generated at the desired accuracy. To quantify accuracy degradations, MONA estimates at run time how many events (*e.g.,* heavy hitters, traffic bursts) remain undetected *after* the measurements sets have been reduced in part of the flows. This is obtained through a novel, task-independent, estimation technique which ensures high levels of confidence by computing estimates according to recently observed traffic characteristics.

MONA is not the first design where monitoring is dynamically configured to achieve accuracy goals, but it is the first that is fully tailored to a software dataplane. Existing solutions rely on approximate measurement techniques such as sketches [86][116][117] and top-k counting [118][79]. However, these techniques are mainly geared towards reducing memory usage, while for software packet-processing the stringent constraint is on CPU-time. In contrast to these approaches, MONA operates with simple hash-tables [8][28][119], with enough space to store (error-free) results for all active flows. This not only shifts the focus of the design from memory to CPU-time consumption, but also allows for more heterogeneous traffic analysis, *e.g.,* beyond volume and connectivity-based

results of sketches [82].

To investigate the benefits of MONA, the proposed framework has been implemented based on a generic and widely-used [8][28] traffic monitoring pipeline relying on a hash table. In addition, a set of widely-used measurement tasks has been enabled in the monitoring pipeline for evaluation purposes, which adopt the same reporting period used in [8] (10*ms*).

The experimental results show that MONA can significantly reduce the risk of packet loss under various events such as multi-Gbps traffic rate spikes, increasing processor concurrency and changes in traffic skew. Furthermore, the evaluation shows a general improvement on the accuracy level of monitoring reports for all measurement tasks. In particular, MONA enhances the monitoring task accuracy – in terms of the task *satisfaction* metric proposed in [56] – by a factor of 2 compared to the traditional *static* monitoring approach. Lastly, the evaluation demonstrates that MONA can operate in short time-scales while incurring only a small CPU-time overhead ($\approx$1-2%).

The remainder of this chapter is organised as follows. Section 4.2 provides background information on software-based traffic monitoring, presents representative measurement tasks used in this chapter, and analyses potential bottlenecks in the monitoring process. In Section 4.3 and overview of MONA and the related design is presented. Section 4.4 describes the adaptive functionalities of MONA in the face of bottlenecks, while Section 4.5 presents the accuracy-aware functionalities of MONA. Section 4.6 evaluates the performance of MONA. Related work specific to traffic monitoring adaptations and software dataplanes is discussed in Section 4.8, and final remarks are presented in in Section 4.9.

4.2 Traffic monitoring in software dataplanes

A number of research approaches have recently embraced the use of commodity hardware to realise a wide range of network functions, as this entails improved flexibility and reduced costs. Traffic monitoring, in particular, is a good candidate for such an implementation, as it can benefit from the processing capability of powerful servers in order to perform complex measurement tasks at the granularity of a single packet, without the need to employ sampling techniques. As such, network operators have started developing monitoring solutions that are part of the software packet-processing pipeline, *e.g.*, in a software switch.

Compared to monitoring operations in hardware switches where memory availability is the main shortage, traffic measurements on commodity hardware are constrained by the CPU-time and the *working set*, *i.e.*, the data most frequently accessed for the measurements [28]. This clearly reflects on the design choices for such monitoring tools. Instead of using approximate measurement techniques like *heap-based* [79] solutions and *sketches* [120][48][34][82], which reduce the total memory usage, traffic monitoring for software dataplanes can just rely on simple hash tables for storing the traffic flow statistics, as they guarantee the best performance in terms of CPU-time and working set [28][8]. Hence, the monitoring process consists of hashing the header of each packet,

e.g., on the 5-tuple, and storing a set of statistics in the hash table.

4.2.1 Measurement tasks

Well-known measurement tasks are adopted in this chapter, which monitor the packet stream and collect traffic statistics over short time intervals (*e.g.,* 10*ms*), called *time-windows*. Each task analyses the packets and collects per-flow results in the flow (hash) table. Flows are defined based on the packet's *5-tuple*[1]. At the end of a window, each task generates a report containing all the *events* found in the flows it processed. In the following, representative examples of measurement tasks extensively studied in the literature [56] [86] [82] [9] [54] [8] are presented. They will be used throughout this chapter.

Heavy Hitter detection (**HH**): discovers traffic aggregates exceeding a bytes threshold, where each aggregate is the sum of all flows with same source IP.

Bursty flow detection (**Bursty**): checks if at least *x%* of the packets of a flow arrived in bursts, *i.e.,* with an interarrival time below *y* milliseconds. If so, the flow is tagged as *bursty*.

Latency Change detection (**LatChange**): checks if the current round-trip-time (RTT) of a flow falls outside the interval $[mean(rtt) - \beta \cdot stddev(rtt), mean(rtt) + \beta \cdot stddev(rtt)]$, where β is a small integer value. If so, it increases the count of latency changes for that flow.

ReTransmission detection (**RTx**): counts for each flow the number of retransmissions (repeated acknowledgment/sequence numbers).

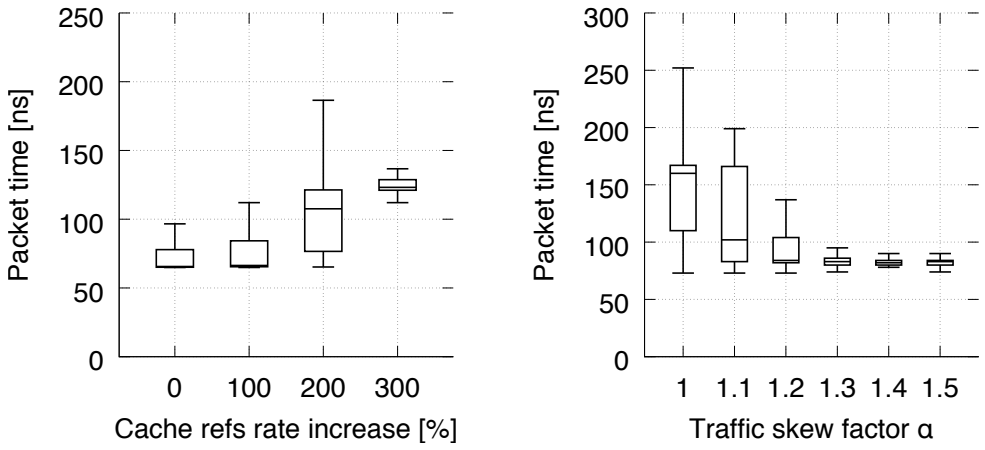
A wide range of analyses can be performed with the statistics collected by these tasks. Heavy hitters are used for anomaly detection and for supporting load balancing decisions. Bursty flows serve the diagnosis of congestion, while retransmissions can reveal reachability problems between two virtual machines (VM) or servers. The results can also be combined to detect network misbehaviours or to guide network management decisions, for traffic engineering or VM migration, for instance. An operator can identify TCP flows with significant loss (high retransmissions) and correlate them with heavy hitters in order to detect short-lived congestions [32]. Alternatively, LatChange can be used to track unresponsive servers and Bursty results are analysed to check if latency changes are due to bursts (*e.g.,* spikes of requests).

4.2.2 Analysis of potential bottlenecks

To satisfy the operator's requirements, traffic monitoring on commodity hardware has to combine three main features: handling high traffic rates at a limited cost (*e.g.*, 1 core for 10 Gbps [8]), achieving zero packet loss, and supporting diverse, sophisticated forms of analysis. Collectively meeting these requirements is not a trivial task.

In order to sustain high throughputs (10+ Gbps), the monitoring process should ensure total packet processing times in the order of few tens of nanoseconds, *e.g.*, no more than 70 ns for 10 Gbps of 64-byte packets. Current state-of-the-art practices, such as RSS (Receive Side Scaling,

[1] 5-tuple fields include source and destination IP, source and destination port, and protocol



(a) *vs.* cache references rate **(b)** *vs.* traffic skew

Figure 4.1: Per-packet processing time (nanoseconds per packet)

a technology enabling the distribution of network receive processing across multiple CPU cores) and capture engines (frameworks like DPDK and Netmap) provide essential support by ensuring packet capture at wire-speed. While these techniques can get packets from the network card to the monitoring process at a high rate, they only solve half the challenge since monitoring bottlenecks can emerge after packets have been captured. These are described below.

Traffic rate variations High-speed packet processing servers use multiple cores and RSS to deal with multi-Gbps traffic. However, these setups are still prone to performance degradation. If the amount of resources devoted to monitoring is limited (*e.g.*, in small-scale deployments), a single core can still face unsustainable workloads at high traffic rates. In addition, events such as fast variation of user demand [121], sub-second congestion [32], or DoS attacks [8] can result to traffic rate spikes affecting one or more cores, even for deployments with multi-queue packet capture (such as RSS). Variations of the input packet rate have a clear impact on the monitoring process. Intuitively, if the rate increases by $x\%$, the available time for processing each packet will drop by $x\%$ or more if additional overheads are included for packet acquisition.

Shared resource contention A monitoring process usually coexists with other tasks on the same machine, often on the same processor, including other monitoring processes running on different cores. Resulting hardware resource contention [115] involves caches, the memory controller, and buses. Among these, the L3 cache, shared by multiple cores in modern platforms, accounts for most of the performance degradation in traffic monitoring, since measurement tasks are particularly aggressive in terms of L3 references per second. Assuming that on a n core processor the monitoring process executes on core 1, variation in data access patterns of other processes running on cores 2 to n can affect the monitoring time per packet due to cache entry replacements, resulting in a higher miss ratio. As depicted in Fig.4.1a, the increase of cache reference rate on cores 2 to n[2] (initially

[2] A 4-cores CPU with 8MB shared L3 cache is used

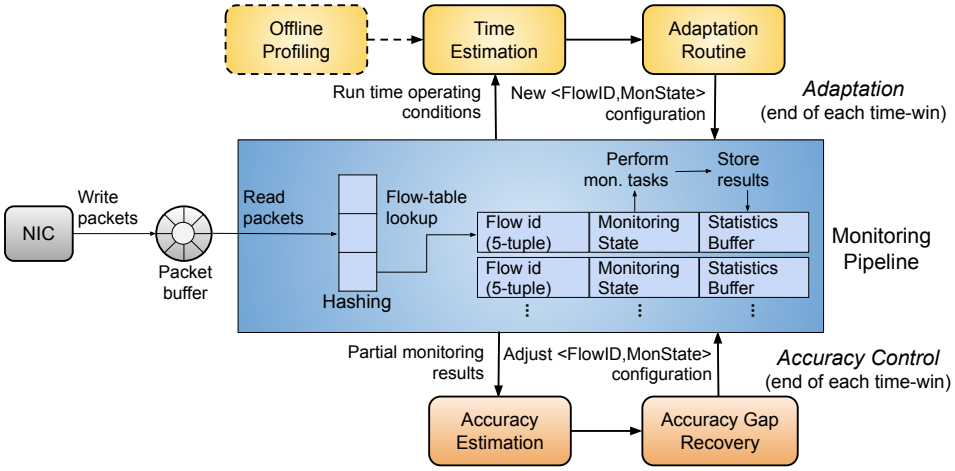


Figure 4.2: Overview of MONA

≈ 0) can double the per-packet latency.

Change of traffic skew The skewness of the traffic distribution plays a key role at run time as it defines the monitoring working set. For traffic with lower skew, a higher fraction of packets cannot be served from the processor caches, resulting in higher packet processing latencies. As an example, the packet completion time is measured for a simple monitoring process that updates packet and byte counts using packet traces with different skewness.[3] As shown in Fig.4.1b, reductions of the skew factor α can double the per-packet latency and generally make it less predictable.

4.3 MONA

Bottlenecks in the monitoring process, resulting from the aforementioned conditions, translate into longer queues in the packet capture stack which lead to higher chances of packet loss. This is an important problem given also the reduced size of RSS queues (no more than 4K packets) and packet I/O rings [70], enough to absorb only less than one millisecond of traffic at 10 Gbps. To ensure resilience to potential bottlenecks, the operator can either count on resource overprovisioning, or restrict the available monitoring tasks to a minimal set, *e.g.*, packet and byte counting only. However, the former approach violates the requirement of only devoting a limited amount of resources for monitoring and the latter penalises the granularity and expressivity of monitoring reports.

To overcome these limitations, this chapter introduces MONA, a traffic monitoring framework for software dataplanes, that dynamically configures the measurement operations sets based on emerging conditions. From a logical view, MONA is the combination of two functions, namely *Adaptation* and *Accuracy Control*. The first function, referred to as *Adaptation*, is aimed at *lossless* traffic monitoring, *i.e.,* it adjusts the monitoring process so that all packets are processed in time. This is achieved through (i) online detection of changes in the operating conditions, and (ii) timely reconfigurations (in $10ms$) of the monitoring operations on part of the input packet stream to obtain

[3] A Zipf distribution and 10^5 flows are used

more *light-weight* processing under strict time constraints. In contrast to existing approaches that use multi-core packet scheduling [68][114], MONA tackles the challenge of lossless traffic monitoring from a different angle, by realising adaptations in the monitoring process itself, at the level of a single CPU core.

Reductions of the packet-processing time are achieved by preventing certain measurement tasks to execute for subsets of the active flow population. For example, by moving the active flows from a configuration where all tasks presented in Sec.4.2.1 are executed (approx. $200ns$ per packet consumption), to one where only *HH* and *RTx* are active (approx. $100ns$ per packet consumption), MONA can handle a packet rate increase from 5 to $10Mpps$. Adaptations of the monitoring operations can however result in missed *events*, *e.g.* undetected bursty flows, which penalises the accuracy of the monitoring reports. To deal with accuracy degradations, MONA executes the second function, referred to as *Accuracy Control*, which re-adjusts the flow allocation after adaptations have been executed to ensure that a user-specified level of accuracy is satisfied for all tasks. This is achieved by (i) tracking the monitoring report accuracy at run-time by estimating the number of events *missing* for each task, and (ii) recovering the identified accuracy gaps by iteratively re-allocating flows so as to meet the desired accuracy objective for all tasks.

4.3.1 MONA design

An overview of MONA is shown in Fig.4.2. The traffic monitoring process is modelled as a packet-processing *pipeline* based on a single hash table, where incoming packets read by the monitoring process from a buffer, e.g., a DPDK ring, are hashed on their 5-tuple to match a corresponding *flow-entry*. Flow-entries contain an identifier, called the Monitoring State, that indicates which measurement tasks to perform for each packet of the flow. The reason for aggregating tasks in *monitoring states* is two-fold. It simplifies the design of applications where transitions between monitoring configurations are decided within the dataplane, *e.g,* based on the outcome of recent measurements and using a finite state machine [122][123]. It also enables settings in which multiple tasks share part of their data to save CPU cycles.

The Adaptation and Accuracy Control functions operate together with the monitoring pipeline, as part of the same process, to avoid additional resource usage (*i.e.*, more cores) as well as synchronisation overheads. As shown in Fig.4.2, the two are however decoupled. This design choice is a consequence of a well-known problem in software-defined measurements [56][86]: determining *a priori* the effects of the set of flows processed by a task on the accuracy of its report is hard. If such relationship could have been easily characterised, an optimisation-based approach would have been used to process all packets in time while minimising accuracy losses.

This difficulty is further amplified by the fact that the impact of adaptation decisions on the report accuracy differ between tasks as this depends on the task-to-monitoring state mapping. Even in the case where the same decision is applied to all tasks, they can each experience different degrees

of accuracy degradations. Take the example of an adaptation that prevents both tasks *Bursty* and *HH* to run on a flow subset containing small but very bursty flows. While this causes a large fraction of bursty flows to be *missed* (low accuracy for *Bursty*), it results in only few missed heavy hitters (high accuracy for *HH*). These differences are not only due to traffic characteristics but also depend on the thresholds used by measurement tasks to trigger new events, *e.g.,* the bytes threshold for a heavy hitter.

In MONA, Adaptation and Accuracy Control are executed separately on the same time-window basis. At the end of each time-window, the former estimates the available packet-processing time and determines the new assignment of flows to Monitoring States, while the latter estimates accuracy degradations and decides how to best recover them. Reconfigurations, if any, are enforced over the following window(s) based on the joint outcome of the two functions.

The design of MONA addresses three main challenges.

1. It ensures limited run time overhead in terms of CPU consumption. All procedures involved in MONA time-share the CPU with packet processing since they operate on the same core, hence their time consumption can directly affect the monitoring pipeline by decreasing the sustainable packet-rate and incurring additional loss. As such, all operations in MONA are designed so that (i) they generate limited CPU-time overhead ($\approx 1ms$), and (ii) they all run to completion in short times (no more than $10\mu s$) to avoid starvation in the packet capture queue.

2. It enables the operational conditions to be accurately estimated by only employing light-weight tools that provide high levels of confidence.

3. It supports reactive monitoring reconfigurations by making sure that each component of the proposed solution works with time-windows as small as $10ms$.

4.4 Monitoring adaptation

As shown in Fig.4.2, the Adaptation function relies on a three-phase procedure. The first phase, *Offline Profiling*, runs at the initialization of the monitoring pipeline before the start of an incoming packet stream. Its role is to profile the various processing times involved in the monitoring pipeline. The other two, namely *Online Estimation* and *Adaptation Routine*, execute at run time.

In each time-window, the Online Estimation procedure extracts the current run time conditions of the monitoring pipeline using limited additional measurements and a few key results from the Offline Profiling phase. At the end of each time-window and based on the extracted knowledge, an estimate of the available monitoring time *per-packet* is generated, which is set as the target for the next window. The *Adaptation Routine* takes this value as input and, if the actual monitoring configuration exceeds the target time, it adjusts the set of measurement tasks for subsets of the flow-table entries to ensure that all packets can be processed on time.

4.4.0.1 Expected time per packet

The total expected time associated with each packet in the monitoring pipeline depends on whether the packet belongs to a new flow, for which no entries exist in the flow-table, or to an existing flow. In the first case, this represents the total processing time for a new-flow packet (including the new flow-entry insertion), denoted here as T_i. In the second case, the time can be decomposed down to two main components: the retrieval time T_r, *i.e.*, the time for retrieving the measurement data for the packet, including hashing and accessing the matching flow-table entry, and the processing time T_p, *i.e.*, the time needed to perform the operations in the current Monitoring State of the matching flow-entry (*i.e.,* the associated measurement tasks). The total expected packet time T_{pkt} can then be estimated based on the following equation:

$$T_{pkt} = (1-\lambda_f)(T_r + T_p) + \lambda_f T_i \tag{4.1}$$

where λ_f represents the ratio of packets belonging to new flows over the total number of packets processed in the current time-window. Based on the findings reported in [115][28], which show that the probability of retrieving data from L3 processor cache is by far the dominant factor affecting the retrieval time, T_r can be further decomposed as:

$$T_r = T_r^H \cdot P + T_r^M \cdot (1-P) \tag{4.2}$$

where T_r^H and T_r^M represent the retrieval time in case the data is accessed from the processor cache and from memory, respectively. P is the probability of cache hit (*i.e.*, the matching flow-table entry is retrieved from the cache), and $(1-P)$ is the probability of cache miss (*i.e.*, access to memory). Combining equations (4.1) and (4.2), the time per packet T_{pkt} is given by:

$$T_{pkt} = (1-\lambda_f)[T_r^H \cdot P + T_r^M \cdot (1-P) + T_p] + \lambda_f T_i \tag{4.3}$$

As observed from equation (4.3), T_{pkt} can be obtained based on the estimation of six variables. To keep the run time adaptation cost as low as possible, the best approach to determine these values is to perform the estimation offline, for example based on benchmarking. While this works well for T_p, T_r^H, T_r^M and T_i, it cannot apply to P and λ_f given that both variables strongly depend on the run time conditions. In this case, an online procedure is required.

4.4.1 Offline profiling

The objective of the Offline Profiling phase is to characterise the resource utilisation of traffic monitoring by analysing the execution times T_p, T_r^H, T_r^M and T_i through a set of benchmarks. While resource consumption can be easily derived online in the case of monitoring solutions dedicated to hardware switches (each monitored flow strictly maps to a single flow-entry in TCAM), it is a much harder task to achieve in software deployments where the focus is on CPU time rather than

Operation	5th Quantile Time (ns)	95th Quantile Time (ns)	Std dev.
Timing	13	15	0.9
T_p: RTx & HH	67	74	2.62
T_p: Bursty & LatChange	74	79	2.13
T_r^H	80	128	8.87
T_r^M	149	272	39.20

Table 4.1: Statistics for T_r and representative T_p datasets

memory usage. Not only do the processing times depend on the server hardware (*e.g.*, clock rate), they also vary based on what monitoring operation must be performed on a packet. Offline Profiling overcomes this limitation by building the knowledge with which resource utilisation can be tracked at run time with a limited cost.

4.4.1.1 Estimating the processing and retrieval times

To determine the value of the processing and retrieval times, the Offline Profiling leverages the observation that in practice the processing time T_p is much more predictable than the retrieval times T_r^H and T_r^M [4]. Intuitively, the processing time for each Monitoring State is proportional to the number of boolean/arithmetic operations executed in that state. In contrast, T_r^H and T_r^M, which are dominated by the flow-entry retrieval time, can be affected by possible hash collisions, the use of different processor caches in the available hierarchy, or unwanted episodes regarding memory access, *e.g.,* TLB (Translation Lookaside Buffer) misses, whose impact also depends on the server hardware and kernel configuration (*e.g.*, memory page size).

To illustrate these effects, Table 4.1 shows the statistics of the execution times distribution for data retrieval (T_r) as well as various monitoring operations (T_p). These tests have been conducted on a 2.7 GHz CPU with 3MB L3 cache. A timing operation is also included in the table, which measures the duration of the different operations using a high definition timer. As observed, while T_r^H and T_r^M exhibit high statistical dispersion, the values of T_p for the two considered Monitoring States are characterised by low standard deviation that can partly be attributed to the bias introduced by the timer. Based on these results, two different strategies have been developed to estimate the values of T_p and T_r^H/T_r^M, respectively.

4.4.1.2 Processing time estimation.

Given the predictability of T_p, the processing time required for each Monitoring State s_i ($T_p^{s_i}$) can be estimated by collecting samples of $T_p^{s_i}$ over a large packet trace and setting the value of $T_p^{s_i}$ to the sample mean.

4.4.1.3 Retrieval time estimation.

Due to the sensitivity of the retrieval times, a different approach is proposed based on statistical model fitting to estimate the value of T_r^H and T_r^M. The proposed approach works in two steps as

[4] Extensive analysis performed during this study confirmed this behaviour.

	Normal (baseline)	Weibull (BIC to baseline)	Gumbel (BIC to baseline)
T_r^H	0%	+1.86%	-3%
T_r^M	0%	+8.1%	-7%

Table 4.2: Model selection for T_r^H and T_r^M

described below.

The objective of the first step is to collect two datasets of time samples, one for T_r^H and one for T_r^M. When collecting the relevant datasets, it is essential to ensure that flow entries reside in either the fast processor caches (for T_r^H) or in memory (for T_r^M). This can be achieved by modulating the size of the monitoring working set used by the input packet stream (*i.e.*, number of unique flows in each trace). Given the L3 processor cache size S, with N_H and N_M denoting the number of different flows in each trace (*i.e.*, for T_r^H and T_r^M, respectively), and the size of the flow-table entry F, the size of the monitoring working set should be so that the following conditions are satisfied: $N_H.S < L3$ (for T_r^H to force cache hit) and $N_M.S \gg L3$ (for T_r^M to ensure cache miss).

Using a standard fitting strategy, the second step selects the most appropriate statistical model to represent the distribution of each variable. Although different distributions can be taken into account, the process has been simplified by restricting the choice to three representative cases capturing well various degrees of sample asymmetry. In particular, the Normal distribution is selected as the baseline, as well as two distributions characterised by a heavy tail, namely the Weibull and the Gumbel distributions. The Bayesian Information Criterion (BIC) is used for the model selection as it shows more consistent results compared to the maximum likelihood estimation for very large sample sizes. It is based on $-2\log(likelihood)$, hence the lower its value the better the fit.

Table 4.2 shows the results of the model fitting strategy based on the considered distributions for the setup of Table I. As observed, the best fit is obtained with the Gumbel model for both T_r^H and T_r^M.

4.4.1.4 Estimating the flow-insertion time

The objective of this procedure is to extract the total execution time for packets of new flows T_i. The estimate of T_i is taken as the average of all T_i values collected from a large packet trace (*e.g.*, approximately $210ns$ for the setup in Table I) under the assumption that new-flow packets form a small subset of the total traffic (*e.g.*, no more than 10%). In cases like SYN attacks, where T_i becomes dominant, existing resilience mechanisms [8], that are out of the scope of this chapter, could be used in conjunction with our solution.

4.4.2 Online estimation

The objective of the Online Estimation phase is to determine at each time-window the value of λ_f and P, as well as the packet arrival rate at the capture engine queue λ_{pkt}. The result is an estimate of the available per-packet time for the next time-window, which is based on the run time conditions of

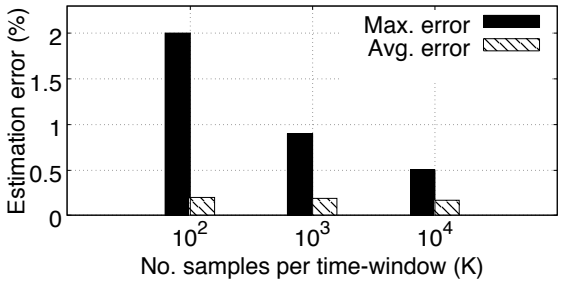


Figure 4.3: Precision of P (probability of flow-entry retrieval from cache) estimation

traffic monitoring and the value of T_p, T_r and T_i computed during the Offline Profiling phase.

To estimate λ_f, a simple strategy incurring negligible overhead is used, which counts the flow-table insertions and divides this value by the total number of processed packets during each time window. The packet capture rate λ_{pkt} is derived by periodically updating the count of packets that have been written in the queue, *i.e.*, each time a new packet burst is loaded by the packet acquisition library. In DPDK [15], for instance, this information can be retrieved using the counts in the rte_eth_stats and rte_ring API.

Several methods can be considered for estimating the value of the variable P. One possibility is to use an analytical model to predict the cache miss rate as proposed in [124]. However, this does not apply well to our solution as: (i) it requires that the temporal behaviour of the application in terms of reuse of addresses has a single profile, which does not apply in our case; and (ii) it does not consider the effect of co-runner processes on the cache hit ratio. Other approaches involving online Miss Rate Curve generation generally incur substantial overheads (*e.g.*, an additional 230 ms is reported in [125]), while faster techniques, like the one presented in [126], rely on cache-related hardware counters that are restricted in current hardware [127].

This chapter proposes a simpler approach which is based on T_r sampling and uses the models of T_r^H and T_r^M obtained from the Offline Profiling phase. In each time-window, the proposed approach periodically samples the flow retrieval time with a high precision timer. Denoting K as the number of samples to collect, the sampling period can be approximated by λ_{pkt}/K. For each sample t_r^i, the approach first computes $Prob(T_r > t_r^i|hit)$, *i.e.*, the probability for the retrieval time to be greater than t_r^i assuming that the retrieval was from a hit, and $Prob(T_r \leq t_r^i|miss)$, *i.e.*, the probability for the retrieval time to be lower than t_r^i under a miss. Given the model of T_r^H and T_r^M computed through profiling, the value of $Prob(T_r > t_r^i|hit)$ is obtained by $1 - CDF_{T_r^H}(t_r^i)$ and $Prob(T_r \leq t_r^i|miss)$ is obtained by $CDF_{T_r^M}(t_r^i)$. Let r denote the vector of results with each element $r(i)$ equal to:

$$r(i) = \begin{cases} 1 & 1 - CDF_{T_r^H}(t_r^i) \geq CDF_{T_r^M}(t_r^i) \\ 0 & otherwise \end{cases}$$

The value of P is approximated as the percentage of non-zero values in the vector r.

To illustrate the performance of the proposed approach in terms of estimation accuracy, this has been used to classify 10^3 different ground-truth traces (for which P is known - P_{truth}) that has been obtained by modulating the traffic skew as explained in Sec.4.2. The estimation error is measured as the difference in percentage between the value of P_{truth} and the value of P computed by the algorithm. As depicted in Fig.4.3, our method achieves very high accuracy on average. Its performance is also compared to the one obtained using a naive classification where each $r(i)$ is set to 1 or 0 by simply using the distance of each t_r^i from the sample mean of T_r^M and T_r^H. For $K = 10^3$ the error is around 5%, whereas our method achieves $< 1\%$.

Available processing time Given the values of λ_f, λ_{pkt} and P, and the variables T_p, T_r^H, T_r^M and T_i, the Online Estimation procedure finally extracts the average processing time for the next time-window T_p^{target} by setting:

$$(1-\lambda_f)[T_r^H P + T_r^M (1-P) + T_p^{target}] + \lambda_f T_i = 1/\lambda_{pkt} \tag{4.4}$$

4.4.3 Adaptation routine

In case a monitoring configuration exceeds the target time T_p^{target}, the role of the Adaptation Routine is to decide which measurement task(s) should be avoided in the next time window, and for which flows. To this end, available knowledge on the monitoring pipeline can be exploited, such as the current average processing time and the time estimations described in Sec.4.4. Based on this information, new monitoring configurations that satisfy the available time constraint can be derived and enforced in the next time window.

Ideally, a new configuration should be generated by a convex optimisation that assigns each flow entry to a specific monitoring state, so as to maximise some objective in terms of monitoring accuracy. In practice, this is not viable for two reasons. The first is the overhead associated with solving such a problem per flow entry, and the second is that the impact of reconfigurations on the monitoring accuracy is not known *a priori*.

To overcome these limitations, the proposed Adaptation Routine takes a different path. Instead of dealing with individual 5-tuples or packets, it operates at the granularity of Monitoring States, as each one maps to a different subset of the flow-table. In addition, it follows a probabilistic approach, where random portions of the flow-table are re-allocated to more light-weight states such that T_p^{target} is achieved. These two features allow our solution to generate a new monitoring configuration within a few microseconds.

The pseudocode of the proposed approach is shown in Alg.2. When invoked at the end of a time-window, it takes as input the count of packets processed at each Monitoring State s_i in the last time-window, and a graph mapping each s_i to a less time-consuming state s_{i-1} is generated. An example of such a graph is shown in Fig.4.4. The way this can be derived depends on the logic of the monitoring process. For example, in scenarios like [123][122], each Monitoring State incorporates

Algorithm 2: Adaptation Routine

1: **procedure** COMPUTENEWCONFIGURATION (T_p^{target},$\{n_i\}$,$\{s_{i-1}\}$)
2: $j = 0$
3: Compute avg processing time: $T^j = \sum_{i=1}^{k} n_i T_p^{s_i} / \sum_{i=1}^{k} n_i$
4: **while** ($T^j > T_p^{target}$) **do:**
5: **for each** $i \in (1,..,k)$ **do:** Shift all flows in s_i to s_{i-1}
6: j++
7: Update avg processing time: $T^j = \sum_{i=1}^{k} n_i T_p^{s_{i-1}} / \sum_{i=1}^{k} n_i$
8: Compute residual shift x: $x = (T^{j-1} - T_p^{target})/(T^{j-1} - T^j)$
9: **return** adaptation decision (j,x)

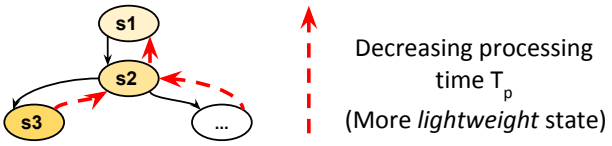


Figure 4.4: Representative monitoring state graph

the necessary logic in its code for transitions to other states once specific conditions on flow-entry statistics are met. For these cases, s_{i-1} is obtained for each s_i by backtracking in the state machine until a more light-weight monitoring state is found.

Our algorithm initially calculates the current average processing time $T^0 = (\sum_{i=1}^{k} m_i T_p^{s_i}) / \sum_{i=1}^{k} m_i$, where m_i is the number of packets in state s_i during the last time-window. Then, it iteratively re-allocates flow-entries to more light-weight Monitoring States. At each iteration j it computes the average processing time if all flows were forced to their previous Monitoring State, *i.e.*, from s_i to s_{i-1}. If this value, T^j, exceeds T_p^{target}, a new iteration is executed. Otherwise, the procedure takes the monitoring configuration for T^{j-1} and forces only a portion x of the flow-table to an additional step-back in the Monitoring State set, where $x = (T^{j-1} - T_p^{target})/(T^{j-1} - T^j)$. In practice, x is the ratio of flows to be further shifted so that the average processing time can match T_p^{target}.

When the next time-window starts, the new configuration is applied using the hash of the first *new* packet for each flow-entry, which ensures that x is a (pseudo)random portion of the flow-table. By comparing the hash with x (using a modulo operation), it decides to update the flow Monitoring State to j or $j-1$ steps back.

4.5 Monitoring accuracy control

While timely adaptations prevent packets from being dropped in the monitoring pipeline, they can penalise the accuracy of the reports computed by each measurement task, as explained in Section 4.3. This section presents a Monitoring Accuracy Control function that aims at re-adjusting the flow allocation *after* adaptations have been executed, so that a global accuracy objective (maintain accuracy above a threshold) can be satisfied for all tasks. Its design consists of two key procedures. The first one quantifies the effect of adaptations on the monitoring report accuracy. Given the lack

of ground-truth information for the monitoring results, this is achieved using a *task-generic* solution that generates accuracy estimates at run-time; prior work *e.g.* [56][86] is limited to employing ad-hoc estimation techniques for each task. The second procedure performs accuracy gap recovery. Based on the output of the online estimation, it re-adjusts the flow allocation in order to alleviate accuracy degradation.

4.5.1 Online accuracy estimation

The objective of the Online Accuracy Estimation procedure is to generate accuracy estimates at run-time based on partial monitoring results. The accuracy of the reports computed by a measurement task is quantified according to the value of the *Recall* [120][118][56][86], *i.e.,* the ratio between the number of events identified by the task (*e.g.,* number of heavy hitters, bursty-flows, *etc.*) and the total number of events in the traffic. This metric is well-aligned with the logic of the adaptations: the accuracy degradation is in terms of *false negatives* (*e.g.*, missed heavy-hitters) rather than *false positives*. The recall of monitoring task i at time-window w is defined as $Recall_{iw}$:

$$\text{Recall}_{iw} = N_{iw}^{Found} / (N_{iw}^{Found} + N_{iw}^{Miss})$$

where N_{iw}^{Found} is the number of events identified by task i, while N_{iw}^{Miss} is the number of events *missing* with respect to the input traffic (*i.e.,* ground-truth), which is unknown. N_{iw}^{Miss} can be further expanded as:

$$N_{iw}^{Miss} = F_{iw}^{Miss} \cdot E[X_{iw}^{Miss}]$$

where F_{iw}^{M} is the number of *missing* flows for task i at time-window w (*i.e.*, flows for which task i has been dropped), which is measured from the output of the *Adaptation Routine*, E indicates the expected value, and X_{iw}^{Miss} is the number of events for a missing flow at w, which is unknown, *e.g.,* the number of retransmissions of a flow not processed by RTx at time-window w. The objective is to determine a reliable estimation of X_{iw}^{Miss}.

Formally, the estimation problem can be modelled using a decision-theoretic approach. In this chapter, the decisions are expressed as risk-taking decisions, where the risk is in generating poor estimations of X_{iw}^{Miss}. Let $\hat{x}^{Miss}$ represent a generic estimator for X_{iw}^{Miss}. A *Risk* function $R(\hat{x}^{Miss})$ is defined, associated with the choice of the estimator $\hat{x}^{Miss}$, as:

$$R(\hat{x}^{Miss}) = \sum_{l=0}^{\infty} L(x_l^{Miss}, \hat{x}^{Miss}) Prob(X_{iw}^{Miss} = x_l^{Miss}) \tag{4.5}$$

where the *Loss* function $L(X_{iw}^{Miss}, \hat{x}^{Miss})$ represents the loss incurred by replacing X_{iw}^{Miss} with $\hat{x}^{Miss}$. The best estimator of X_{iw}^{Miss} is the one with which $R(\hat{x}^{Miss})$ is minimised, *i.e.,*:

$$\hat{x}_{Best}^{Miss} = \underset{\hat{x}^{Miss}}{\operatorname{argmin}} R(\hat{x}^{Miss}, L) \tag{4.6}$$

As can be observed, $\hat{x}_{Best}^{Miss}$ depends on the choice of the loss function L. In practice, different estimators can be used for X_{iw}^{Miss}. A possible approach is to replace X_{iw}^{Miss} with its worst-case value [56]. However, this solution is prone to significantly underestimating the accuracy if F_{iw}^{Miss} is large. In addition, it is not suitable for some tasks, *e.g.,* the worst-case number of missing flow retransmissions cannot be known.

This chapter takes a different approach and proposes to estimate X_{iw}^{Miss} based on the monitoring results observed over the most recent q time-windows. More specifically, let $X_{iw} = (x_{iw1}, x_{iw2}, ..., x_{iwm})$ represent the vector of the results x_{iwj} of task i, for each flow j, at time-window w. x_{iwj} is equal to the number of events detected by task i for flow j at time-window w, and undetermined if j is a missing flow for task i. The temporal dependence of X_{iw} over the most recent q time-windows is modelled as a moving average $MA(q)$ with order q [5]:

$$X_{iw} = \mu_i + z_w + \theta_1 z_{w-1} + ... + \theta_q z_{w-q}$$

where μ_i and θ_i are the mean and parameters of the model, respectively, and z_w ... z_{w-q} is Gaussian noise so that $E[X_{iw}] = \mu_i$.

The value of μ_i can easily be determined by noting that $E[X_{iw}] = E[\overline{X}_{iw}] = \mu_i$, where $\overline{X}_{iw}$ represents the mean of vector X_{iw}. In other words, to characterise the model, it is sufficient to track the values $\overline{X}_{i(w-q)}$ ... $\overline{X}_{iw}$ in the last q time-windows. Given that $\overline{X}_{iw}$ are unknown by definition, their value is estimate using the observed average monitoring results (*i.e.,* extracted from flows processed by task i), denoted as $\overline{x}_{i(w-q)}^{Obs}$ ... $\overline{x}_{iw}^{Obs}$. μ_i is then obtained by taking the weighted average of values $\overline{x}_{i(w-q)}^{Obs}$ ... $\overline{x}_{iw}^{Obs}$ as follows:

$$\mu_i = \frac{\sqrt{\frac{n_{iw}}{\sigma_{iw}^2}}\overline{x}_{iw}^{Obs} + \sqrt{\frac{n_{i(w-1)}}{\sigma_{i(w-1)}^2}}\overline{x}_{i(w-1)}^{Obs} + ... + \sqrt{\frac{n_{i(w-q)}}{\sigma_{i(w-q)}^2}}\overline{x}_{i(w-q)}^{Obs}}{\sqrt{\frac{n_{iw}}{\sigma_{iw}^2}} + \sqrt{\frac{n_{i(w-1)}}{\sigma_{i(w-1)}^2}} + ... + \sqrt{\frac{n_{i(w-q)}}{\sigma_{i(w-q)}^2}}}$$

where n_{iw} is the number of available results for task i at time-window w (*i.e.,* number of flows processed by task i) and σ_{iw}^2 their sample variance. In practice, the more the results available at time w and the less their variance, the higher the contribution of time-window w.

Using the average values $\overline{x}_{iw}^{Obs}$ to compute the model parameters offers several advantages. The state that needs to be maintained for each monitoring task is drastically reduced and the computational overhead of the estimation is decreased.

We take into account the model of the evolution of $\overline{X}_{iw}$ values over the last q time-windows to set the loss function L, so as to make the accuracy estimation more or less conservative according

[5] By default results from the last 10 time windows are used, so $q = 10$

to run-time conditions. In particular, these are assessed based on the probability of large accuracy estimation errors. If the probability is small, L is replaced with the *quadratic loss function*, widely-used in testing (*e.g.*, *least squares* techniques), that penalises the large errors more. In contrast, if the probability is large, L is defined so that only large deviations from X_{iw}^{Miss} are penalised. More specifically, the probability of large estimation errors depends on the variability of the distribution of the results of task i. This is captured by the coefficient of variation σ_i/μ_i of the mean values $\overline{X}_{iw}$ over the last q time-windows, where σ_i is the standard deviation of $\overline{X}_{iw}$ (measured based on $\overline{x}_{iw}^{Obs}$ values)[6]. The ratio $\sigma_i/\mu_i = 1$ is used as the threshold of high variability[7] and the loss function L is defined as follows:

$$L = \begin{cases} (X_{iw}^{Miss} - \hat{x}^{Miss})^2 & \frac{\sigma_i}{\mu_i} \leq 1 \\ \mathbb{I}[|X_{iw}^{Miss} - \hat{x}^{Miss}| > c] & \frac{\sigma_i}{\mu_i} > 1 \end{cases} \tag{4.7}$$

where c is an arbitrary value to decide when to penalise estimation errors in case $\frac{\sigma_i}{\mu_i} > 1$.

Based on (4.7), the best estimator $\hat{x}^{Miss}$ of X_{iw}^{Miss} is determined depending on the value of σ_i/μ_i, as follows:

$$\hat{x}^{Miss} = \begin{cases} \mu_i & \frac{\sigma_i}{\mu_i} \leq 1 \\ \underset{\hat{x}^{Miss}}{\operatorname{argmin}}\ Prob(|X_{iw}^{Miss} - \hat{x}^{Miss}| > c) & \frac{\sigma_i}{\mu_i} > 1 \end{cases} \tag{4.8}$$

In the following the proof of equation (4.8) for *case 1* (*i.e.,* $\frac{\sigma_i}{\mu_i} \leq 1$) is reported, as well as the justification of equation (4.8) for *case 2* (*i.e.,* $\frac{\sigma_i}{\mu_i} > 1$).

Proof of (4.8) - case 1 Replacing L with the Quadratic Loss Function $(x_l^{Miss} - \hat{x}^{Miss})^2$ in the expression of $R(\hat{x}^{Miss})$ we obtain:

$$\begin{aligned} R(\hat{x}^{Miss}) &= \sum_{l=0}^{\infty} (x_l^{Miss} - \hat{x}^{Miss})^2 Prob(X_{iw}^{Miss} = x_l^{Miss} | X_{iw}^{Obs} = x_{iw}^{Obs}) \\ &= \sum_{l=0}^{\infty} (x_l^{Miss})^2 Prob(X_{iw}^{Miss} = x_l^{Miss} | X_{iw}^{Obs} = x_{iw}^{Obs}) \\ &- 2\hat{x}^{Miss} \sum_{l=0}^{\infty} x_l^{Miss} Prob(X_{iw}^{Miss} = x_l^{Miss} | X_{iw}^{Obs} = x_{iw}^{Obs}) \\ &+ (\hat{x}^{Miss})^2 \sum_{l=0}^{\infty} Prob(X_{iw}^{Miss} = x_l^{Miss} | X_{iw}^{Obs} = x_{iw}^{Obs}) \end{aligned}$$

To minimise the Risk function we set $\frac{dR(\hat{x}^{Miss})}{d\hat{X}^{Miss}} = 0$:

[6] σ_i^2 is the variance of the sample mean of X_{iw}

[7] This is a standard choice, based on the comparison with the exponential distribution

$$\frac{dR(\hat{x}^{Miss})}{dX^{Miss}} = 2\sum_{l=0}^{\infty} x_l^{Miss} Prob(X_{iw}^{Miss} = x_l^{Miss} | X_{iw}^{Obs} = x_{iw}^{Obs})$$
$$- 2\hat{x}^{Miss} \sum_{l=0}^{\infty} Prob(X_{iw}^{Miss} = x_l^{Miss} | X_{iw}^{Obs} = x_{iw}^{Obs}) + 0 = 0$$

from which we obtain:

$$\hat{x}^{Miss} = \sum_{l=0}^{\infty} x_l^{Miss} Prob(X_{iw}^{Miss} = x_l^{Miss} | X_{iw}^{Obs} = x_{iw}^{Obs}) = E[X^{Miss}] = \mu_i$$

Hence, the risk is minimised choosing $\hat{x}^{Miss} = \mu_i$.

Justification of (4.8) - case 2 In this case the loss function L is replaced by:

$$L = \mathbb{I}[|X_{iw}^{Miss} - \hat{x}^{Miss}| > c] = \begin{cases} 1 & |X_{iw}^{Miss} - \hat{x}^{Miss}| > c \\ 0 & |X_{iw}^{Miss} - \hat{x}^{Miss}| \leq c \end{cases}$$

and the expression of the Risk function becomes:

$$\begin{aligned} R(\hat{x}^{Miss}) &= \sum_{l=0}^{\infty} \mathbb{I}[|x_l^{Miss} - \hat{x}^{Miss}| > c] Prob(X_{iw}^{Miss} = x_l^{Miss} | X_{iw}^{Obs} = x_{iw}^{Obs}) \\ &= Prob(|X_{iw}^{Miss} - x_l^{Miss}| > c | X_{iw}^{Obs} = x_{iw}^{Obs}) \end{aligned} \tag{4.9}$$

As such, minimising the Risk function corresponds to minimising the probability of estimation errors larger than c. ■

As shown in (4.9), in case $\frac{\sigma_i}{\mu_i} > 1$, the equation 4.8 cannot be solved if no assumption is made on X_{iw}^{Miss}. As such, for this case, instead of defining a solution for (4.8), we provide an approximate bound for the Risk function. To this end, we apply the Chebyshev's inequality to the probability of large estimation errors $Prob(|X_{iw}^{Miss} - \hat{x}^{Miss}| > c)$, *i.e.,*

$$Prob(|\overline{X}_{iw} - \mu_i| \geq \lambda\sigma_i) \leq \frac{1}{\lambda^2}$$

from which we can further derive:

$$Prob(\overline{X}_{iw} \geq \mu_i + \lambda\sigma_i) \leq \frac{1}{\lambda^2}$$

This gives an estimator of X_{iw}^{Miss} equal to $\mu_i + \lambda\sigma_i$ in the case of large probability of large estimation errors, with guarantees that the Risk is no larger than $1/\lambda^2$. For instance, with $\lambda = 3$, the *Risk* of

poor estimation is bounded to 10%.

Summary: The estimation procedure for a generic task i operates as follows. At the end of time-window w, the number of available results n_{iw} and their sample variance σ^2_{iw} are updated. These values, together with those obtained in the $q-1$ previous time windows, are then used to compute μ_i. The sample mean of the available results $\overline{x}^{Obs}_{iw}$ are also extracted and used to update the coefficient of variation of $\overline{X}_{iw}$, *i.e.,* $\frac{\sigma_i}{\mu_i}$. Based on the value of $\frac{\sigma_i}{\mu_i}$, the loss function L is selected according to (4.7). The estimator is finally chosen as μ_i if $\frac{\sigma_i}{\mu_i} \leq 1$, and as $\mu_i + \lambda\sigma_i$ otherwise.

4.5.1.1 Performance evaluation and discussion

Fig.4.5 illustrates the performance of the proposed online accuracy estimation procedure in terms of *Accuracy Estimation Error* calculated as the absolute distance between the Estimated_Recall and Real_Recall (extracted from the ground-truth) normalised by the Real_Recall. Experiments are conducted using 100 ground-truth packet traces derived from [55], with dynamic rate in $[0Gbps, 10Gbps]$, and the four different monitoring tasks presented in Sec.4.2.1. To show the gain achieved by adapting $\hat{x}^{Miss}$ to the run-time conditions, the performance is also compared against two baseline approaches, one with an estimator always given by the mean (*mean*), and one only using the upper bound $\mu_i + \lambda\sigma_i$ (*u-bound*).

As can be observed, the estimation error is generally low, overall 8% on average, and never exceeds 20%. The highest error (around 17%) is obtained for *LatChange*, which is the task producing the least predictable monitoring results in the experiments. Compared to the two baseline cases, the proposed approach generally achieves lower error, with substantial gain in particular for *RTx* (2x reduction compared to *u-bound*) and *LatChange* (>5x reduction compared to *mean*).

In addition to performance gains, a key advantage of the proposed method is its task-generic nature as opposed to recent solutions for TCAM-based [56] and sketch-based [86] measurements, which define ad-hoc accuracy estimation procedures for each monitoring task. In practice, the proposed approach can be applied to all monitoring tasks whose output is a count (*e.g., how many retransmissions for flow x?*) or a classification (*e.g., is aggregate y a heavy-hitter?*). As shown in [8][56][86], tasks detecting network episodes or anomalies, or serving network management decisions, fall well in these categories. In addition, the proposed solution can run on a limited time budget as it involves operations that can be executed at a low cost: all distributions are obtained by sampling and fast techniques can be used to compute mean and variance values, *e.g.,* [128].

4.5.2 Accuracy gaps recovery

An estimate, based on the value of the *Recall*, is generated by the Online Accuracy Estimation procedure for each monitoring task at the end of each time-window w. The estimates are further used by an Accuracy Gaps Recovery procedure to re-adjust the flow allocation so that accuracy degradation resulting from the *Adaption Routine* can be alleviated.

Let A_{iw} denote the accuracy estimate for task i at time-window w. A task is tagged as *poor*

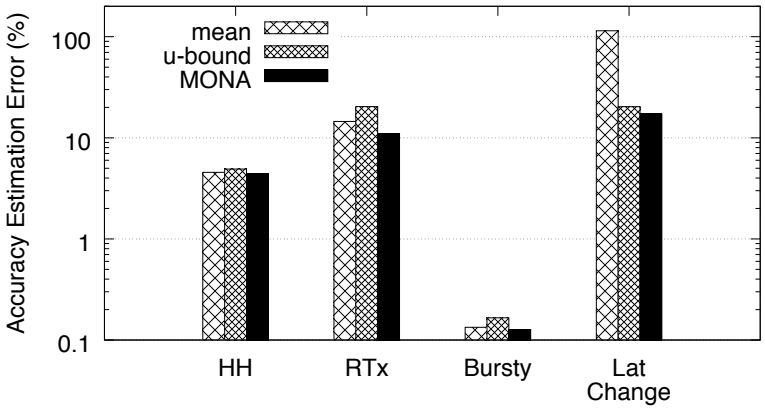


Figure 4.5: Performance of Accuracy Control: accuracy estimation error

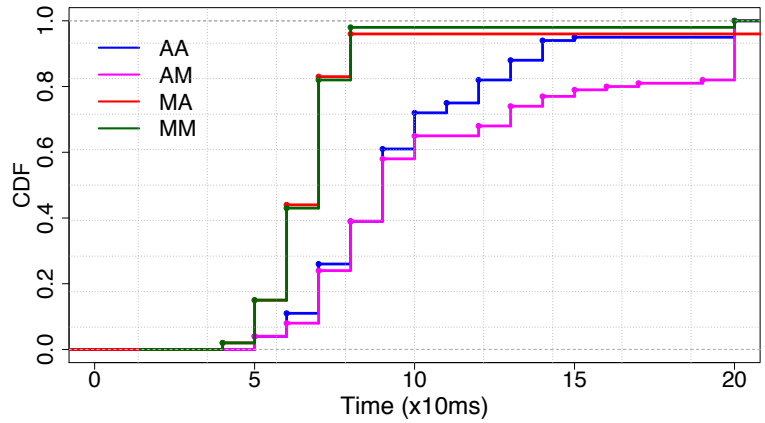


Figure 4.6: Performance of Accuracy Control: recovery convergence time (multiples of 10ms time window)

at time w if A_{iw} is below a threshold and as *rich* otherwise.[8] The objective of the Accuracy Gaps Recovery procedure is to recover the accuracy gap of a poor task by applying it to a larger set of flows and compensating the additional CPU time needed with controlled degradation of rich tasks. This is achieved using a *rebalancing* approach that reallocates subsets of flows to different *monitoring states* in an iterative fashion, with one iteration per time-window. Ideally, all accuracy gaps should be recovered within one time-window, but this is difficult to achieve in practice. Not only is the number of flows needed to meet an accuracy target not known *a priori*, it is also not possible to determine the maximum number of flows a task can *donate* while keeping the accuracy of its report above the desired threshold.

More specifically, at each iteration the algorithm presented in Alg.3 is executed. Let $A_w = (A_{1w}, A_{2w}, ..., A_{kw})$ denote the vector of estimates for each monitoring task $1, ..., k$ and M a binary $n \cdot k$ matrix, where n is the number of monitoring states and k the number of tasks, with the task/state *mapping* so that $M_{ij} = 1$ if state i includes task j, and 0 otherwise. In addition, let B be a $n \cdot n$ matrix

[8] For simplicity, the same accuracy threshold was used for all tasks.

indicating the transitions of flows between monitoring states due to previous adaptation decisions, as recorded in the current time-window. Each value B_{ij} is the ratio of flows that were previously moved by the *Adaptation Routine* from state i to j (with j being more light-weight than i) to save time.

Alg.3 initially selects, from the global set of monitoring states, a subset of *poor* states, containing one or more poor task(s), and a subset of *rich* states, in which all tasks have an accuracy higher than the desired threshold with a small headroom ε. The headroom is used to prevent rich states becoming poor after a single iteration. For each pair of states (s_{rich}, s_{poor}) a re-balancing action is taken based on a *step-size* parameter S. This involves shifting Δ^- flows from s_{rich} to the default state (*e.g.,* s_1 in Fig.4.4) and using the resulting gain in time to revert the monitoring adaptation for Δ^+ flows moved from s_{poor}. The value Δ^- is set to S normalised by the number of poor states, while Δ^+ is obtained from Δ^- by imposing the equilibrium condition *"constant CPU time consumption"*:

$$\Delta^-(t_s^{rich} - t_s^{default}) = \Delta^+(t_s^{poor} - E[\widetilde{t_s^{poor}}]) \tag{4.10}$$

where the values t_s are the state execution times, and $E[\widetilde{t_s^{poor}}]$ is the expected execution time for flows moved away from s_{poor} by the *Adaptation Routine*, as indicated by B:

$$E[\widetilde{t_s^{poor}}] = \frac{\sum_j B_{poor,j} t_s^j}{\sum_j B_{poor,j}}$$

The choice of the step-size S involves a trade-off between stability and convergence time, which is a well-known challenge of any resource allocation algorithm. While using small steps may lead to high convergence times, big step sizes can make measurement tasks oscillate between *rich* and *poor* over consecutive time-windows. The best practice for setting the value of S is to use an increase/decrease policy [56]. In this work, we relate the change of S to the accuracy evolution of each *rich* state s_{rich}, with the objective to converge rapidly while preventing s_{rich} from dropping below the desired accuracy threshold. After each iteration, *i.e.,* at time-window *w+1*, the algorithm computes the decrease of s_{rich} accuracy $D = A_{rich,w+1} - A_{rich,w}$, as well as its residual accuracy $H = threshold - A_{rich,w+1}$. S is increased if $H > D$ and decreased otherwise.

The impact of different standard increase-decrease policies is evaluated on the convergence times of Alg.3. More specifically, we experiment with different combinations of the *additive* (A) and *multiplicative* (M) policies. The results are depicted in Fig.4.6 for the same input traffic used in Fig.4.5 and initial S values in $[0.25\% - 10\%]$ of the initial active flow set. The best performance in this scenario is achieved by the multiplicative increase-decrease policy (MM).

4.6 Evaluation

This section extensively evaluates MONA framework. To this end, MONA is implement based on a generic and widely-used [8][28] traffic monitoring pipeline relying on a single hash table. In addition, to quantify the gain introduced by MONA Accuracy Control functionality, the monitoring

Algorithm 3: Recover Accuracy Gaps

function UPDATESTEPSIZE(x, S_x)
  Compute accuracy decrease $D = A_{x,w-1} - A_{x,w}$
  Update residual accuracy $H = A_{x,w} - threshold$
  if $D > H$ **then return** INCREASE(S_x)
  else return DECREASE(S_x)
function REBALANCEBYSTEP(s_{Rich}, s_{Poor}, S)
  Compute $\Delta^- = S/n_p$, where n_p number of poor states
  Retrieve $E[\widetilde{t_s^{Poor}}]$ from $T_{Poor,j}$, $j \in 1,..,n$
  Compute Δ^+ from equilibrium condition (4.10)
  return Δ^-, Δ^+
procedure RECOVERYGAPS(A_w, M, T_w)
  Find set of rich, poor states $\{s_{Rich}\}$, $\{s_{Poor}\}$ using A_w
  if $\{s_{Poor}\} == \emptyset$ or $\{s_{Rich}\} == \emptyset$ **then return**
  for each x in $\{s_{Rich}\}$ **do**:
    S_x = UPDATESTEPSIZE(x, S_x)
  for each (x, y) with $x \in \{s_{Rich}\}, y \in \{s_{Poor}\}$ **do**:
    REBALANCEBYSTEP(x, y, S)

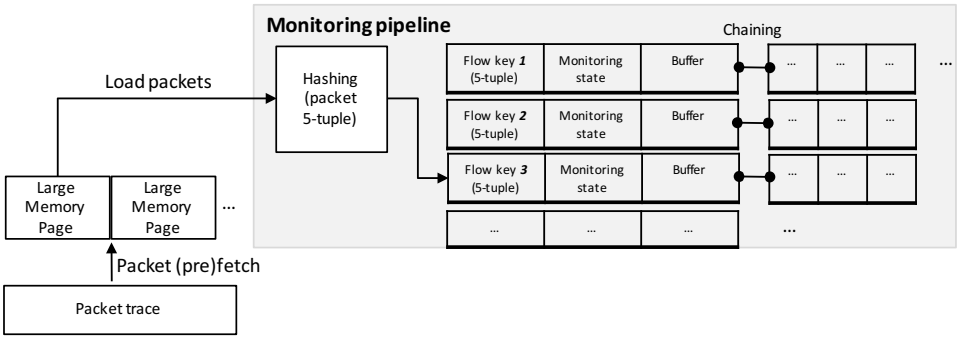


Figure 4.7: MONA monitoring pipeline implementation

pipeline is enabled to perform a set of representative, widely-used, measurement tasks, for which the associated monitoring reports' accuracy is analysed.

The evaluation is conducted in three main steps. First, Section 4.6.3 tests the performance of adaptive traffic monitoring in terms of packet loss risk and adaptation responsiveness. Then, Section 4.6.4 investigates the impact of monitoring adaptations and accuracy control on the measurement tasks. Finally, in Section 4.6.5 the overhead of the proposed framework is evaluated. This includes the completion times of the main procedures and the additional run time overhead incurred.

All experiments have been performed on an Intel i7-4790 CPU with 4 physical cores at 3.6 GHz and shared L3 cache of 8 MB.

4.6.1 Implementation of MONA

MONA is implemented in the C language as part of a generic monitoring pipeline based on a single flow-table. An overview of MONA monitoring pipeline is depicted in Fig. 4.7. A hash table is used to realise the flow-table where collisions are handled by chaining – a table size of 2^{20} entries was

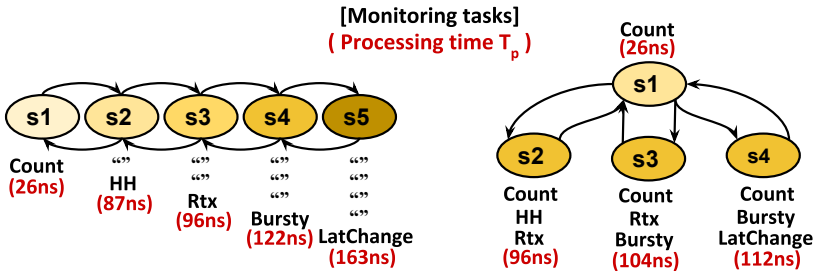


Figure 4.8: Monitoring states Config1 (left) and Config2 (right)

chosen to limit the risk of hash collisions. The flow-entry size is set to 64 bytes such that it can fit in a single cache line. The input packet streams to the monitoring pipeline is generated based on the following approach. Since the focus is on the bottlenecks arising in the monitoring process, for each experiment a packet trace is built and pre-loaded in memory. Packets are then fetched with small bursts at run time so as to isolate the monitoring pipeline from the packet capture stack. One second of traffic is pre-loaded, which corresponds to approximately 1GB allocated memory for 10Gbps of traffic.

4.6.2 Experiment setup

Packet trace The packet trace used includes only TCP flows and is derived from recently published results on flow statistics in data centers [55]. The main characteristics of the trace in terms of packet size, 5-tuple flow size, and 5-tuple flow duration are listed in Table 4.3. In addition, packet retransmissions are injected in the trace using the Gilbert-Elliot model [130][74].

Table 4.3: Packet trace characteristics

	5th %ile	25th %ile	Median	75th %ile	95th %ile
Packet size [B]	60	60	150	200	1000
Flow size [KB]	0.1	0.5	1	5	250
Flow duration [ms]	1	10	750	$5 \cdot 10^4$	10^5

Measurement tasks setup The monitoring pipeline adopts the four measurement tasks introduced in Chapter 4.2.1. All tasks are configured to report measurement results based on the same period used in [8] ($10ms$). Table 4.4 summarises the measurement tasks used and reports their setup in terms of the thresholds used.

Monitoring states setup To experiment with the monitoring pipeline, we define two different monitoring state configurations, namely *Config.1* and *Config.2*. These are presented in Fig.4.8, along with their (estimated) processing times T_p. Both configurations include a default state s_1 where only simple packet counting (*Count*) is performed. In Config.1, additional tasks are progressively incorporated in each step of the monitoring states chain. The process initially detects large traffic aggregates (s_2) and then starts monitoring the packet loss of these aggregates (s_3). For flows with a high loss, it incorporates burst detection in s_4 and finally looks for RTT changes in s_5. In the case of

Table 4.4: Representative measurement tasks

Task	Description	Setup
Heavy Hitter detection(**HH**)	Find traffic aggregates exceeding a bytes threshold B, where each aggregate is the sum of all flows with same source IP	$B = 5$MB
Bursty flow detection (**Bursty**)	Find if at least $x\%$ of the packets of a flow arrived in bursts, *i.e.,* with an interarrival time below y milliseconds. If so, the flow is tagged as *bursty*	$x = 10$, $y = 1$
Latency Change detection (**LatChange**)	Find if the current round-trip-time (RTT) of a flow falls outside the interval $[mean(rtt) - \beta \cdot stddev(rtt), mean(rtt) + \beta \cdot stddev(rtt)]$	$\beta = 1$
ReTransmission detection (**RTx**)	Count for each flow the number of retransmissions (repeated acknowledgment / sequence numbers)	

Config.2, which follows a hierarchy, flows can be processed according to three possible monitoring states, inspired by the use-cases considered in [8]. The goal of s_2 is to identify the root cause of congestion by correlating lossy TCP flows with heavy hitters. State s_3 identifies loss as a result of bursty traffic, and s_4 detects server imbalance by collectively tracking latency changes and bursty flows.

4.6.3 Lossless traffic monitoring

We compare the proposed approach (*Adapt*) with a more traditional setup where monitoring operations are not dynamically reconfigured (*No-Adapt*). We focus on two metrics that represent the risk of packet loss as a result of bottlenecks in the monitoring process. The first is *packet balance*, which indicates whether the system can process the number of packets in the trace during each time-window – a negative value signifies that the system cannot cope with the packet rate. The second metric is the *expected packet loss*, which quantifies the loss given the size of the input buffer. We compute this using a queue model for two buffer sizes: 4096 packets (saturation of a RSS queue) and 1 MB (maximum size range for circular buffers in packet acquisition libraries like DPDK and Netmap) [70].

For these experiments we use *Config.1* and we initially assign $1/5$ of the flows to each state. The monitoring adaptation time-window is set to $10ms$. To study the performance of our solution under the emergence of bottlenecks, we perform three types of experiment which reproduce the three main conditions described in Section 4.2.

Traffic rate variations In these experiments the proposed solution is tested against multi-Mpps variations of the input rate, which are realised by tuning the rate of exponential packet inter-arrivals in the trace. We increase the rate linearly from 0 to 14.8 Mpps (10 Gbps of small packets) during a 1 second interval. As shown in Fig.4.9a, the proposed solution achieves a significantly higher

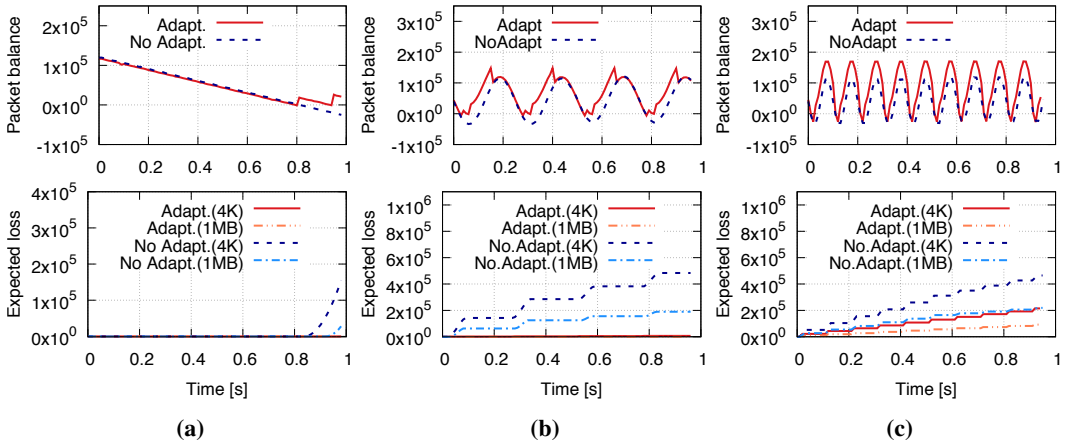


Figure 4.9: Packet balance and expected packet loss: traffic rate variations

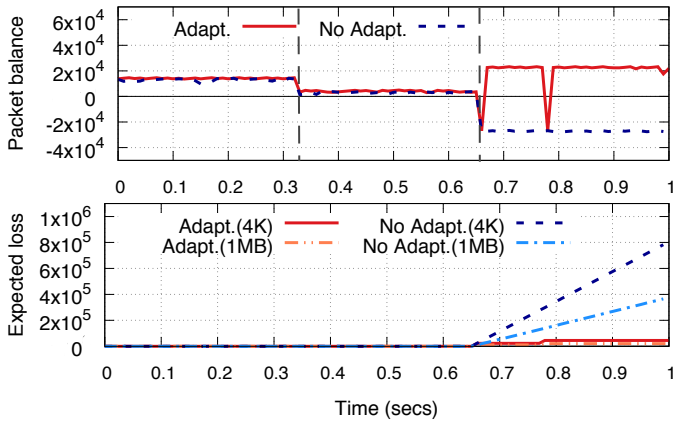


Figure 4.10: Packet balance and expected packet loss: shared resource contention

packet-rate compared to the non-adaptive approach (by 2 Gbps). In the *No-Adapt* case losses are observed after $t \approx 0.8s$, as soon as the rate becomes unsustainable, and a larger buffer (1MB) can only postpone losses by a few tens of milliseconds. To avoid the loss a second core would need to be allocated for monitoring the same input packet stream. In contrast, the two adaptations performed by the proposed approach at $t \approx 0.8s$ and $t \approx 0.95s$ allow to sustain a maximum rate of 14.8 Mpps on a single core without any packet loss.

In addition, the responsiveness of monitoring adaptations in the case of short rate spikes is evaluated. To emulate the spikes, we generate packet-rate oscillation between 0 and 14.8 Mpps based on the function $sin(t/T)$, where T is the oscillation period. Two representative cases, $T = 250ms$ and $T = 100ms$ are depicted in Fig.4.9b and 4.9c, respectively. For $T = 250ms$, monitoring adaptations always provide a response to arising bottlenecks in time, before packet loss occurs. In the case of $T = 100ms$, huge packet rate variations, up to the equivalent of 3Gbps for 64-byte packets, are generated in the time-span of a single monitoring adaptation time-window ($10ms$). As

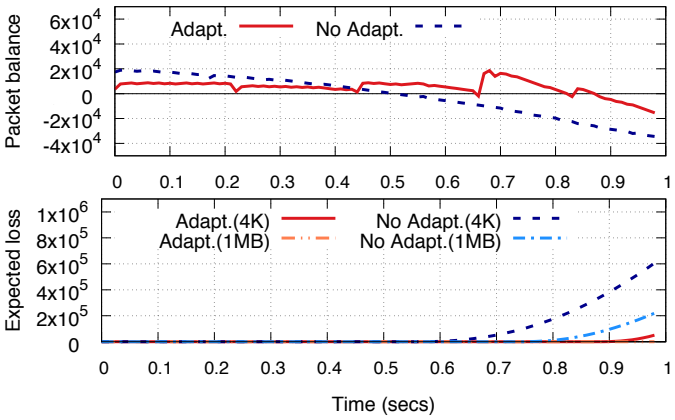


Figure 4.11: Packet balance and expected packet loss: traffic skew variation

such, some losses can occur before the new monitoring configuration is applied, *i.e.*, in the $10ms$ time-window preceding the adaptation. However, even for such intense and short-lived spikes the proposed approach significantly outperforms *No-Adapt* in terms of loss, with a reduction of more than 50% for both buffer sizes.

Shared resource contention The objective of the next experiments is to assess how monitoring adaptations handle variations of the operating conditions in terms of concurrent access to shared resources. To emulate concurrency we use the approach in [115], *i.e.,* we run our solution on core 1, and *co-run* other processes on cores 2, 3 and 4. Each co-runner is defined as a special monitoring process that only retrieves flow-entries, so as to maximise the L3 cache references per second. As depicted in Fig.4.10, we split the 1-second experiment into three intervals of length $1/3s$ each. In the first interval we execute 1 co-runner (on core 2), in the second interval we execute 2 co-runners (on cores 2 and 3), and in the last interval we execute 3 co-runners (on cores 2, 3 and 4).

As shown in Fig.4.10, increasing levels of concurrency lead to performance degradation in terms of packets processed per time-window, and considerable loss for the *No-Adapt* setup with 3 active co-runners, regardless of the input buffer size. This is due to the inflation of retrieval times T_r as a result of increasing L3 cache misses. On the contrary, the proposed solution achieves minimal loss since concurrency variations are detected at run time through P estimation (Section 4.4.2) and a new monitoring configuration is provided within $10ms$.

Change of traffic skew Lastly, the performance of the proposed solution is evaluated under variations of traffic skew. To reproduce these variations we split the input packet trace into smaller intervals of $20ms$ and for each interval we assign packets to flows (5-tuples) based on a Zipf distribution with parameter α (flow population size of $2.5 \cdot 10^5$). We start with $\alpha = 1.5$ (high skew) at $t = 0s$ and we gradually decrease the skew factor until $\alpha = 1$ at $t = 1s$ to obtain more *uniform* traffic. The packet rate has been fixed at 10 Mpps. As expected, smaller values of α lead to a significant performance drop since less packets are served with flow-entries from the L3 cache. As shown in Fig.4.11,

our solution can sustain 10 Mpps on a single core under considerable skew variations, and prevents losses at much lower skew factors, $\alpha \approx 1.1$ ($t = 0.9$), compared to the *No-Adapt* case, which starts starving packets in the input buffers for $\alpha \approx 1.25$.

4.6.4 Monitoring report accuracy

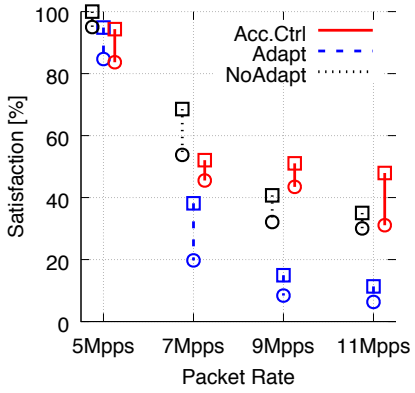
The impact of the proposed solution on the monitoring report accuracy is now evaluated to investigate how the report quality is preserved under bottleneck conditions. To this end, we measure the real accuracy, in terms of *Recall*, of each task running in the system. This is obtained by comparing the monitoring results of the system against the respective *ground-truth*, which is extracted from offline analysis of the traces.

To control the accuracy we use a threshold of 75% for all tasks, which is inline with the one used in [56]. For simplicity, traffic flows are assigned randomly to any available monitoring state with equal probabilities. Finally, to generate bottlenecks and thus trigger adaptations, we use a dynamic input packet rate in the range $[5, 11]$ Mpps, which is obtained by tuning packet inter-arrivals in the trace.

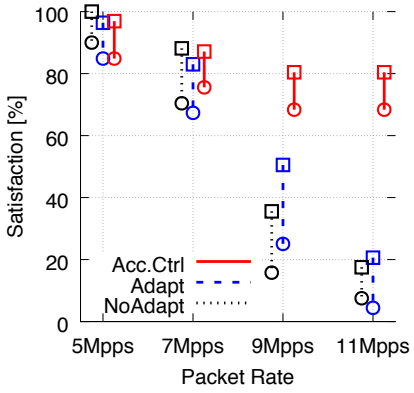
Fig.4.12 presents the monitoring accuracy results in terms of a *Satisfaction* metric similar to the one used in [86] and [56]. This represents the fraction of time a task has its *Recall* is above the threshold. Intuitively, 100% satisfaction for all tasks means that the global accuracy goal is fully achieved. The minimum and median satisfaction is shown from a set of 100 experiments, for the two monitoring state configurations (Config.1, Config.2). As depicted in Fig.4.12, we compare the performance obtained with accuracy control (*Acc.Ctrl*) against two baselines: *No-Adapt* which is the standard setup without dynamic adaptations and accuracy control; and *Adapt* which executes adaptations without accuracy recovery in place.

As shown in Fig.4.12, both *Adapt* and *No-Adapt* incur serious accuracy degradations, with the satisfaction dropping even below 20% for some tasks. For the *Adapt* case, this is due to accuracy-unaware adaptations while, in the *No-Adapt* case, the reduced satisfaction depends on the amount of packets never entering the monitoring pipeline as they are starved in the buffers under increasing packet rates. In contrast, *Acc.Ctrl* drastically improves the report accuracy, raising the overall median satisfaction to 75%; 2x and 3x improvement compared to *No-Adapt* (38%) and *Adapt* (23%), respectively.

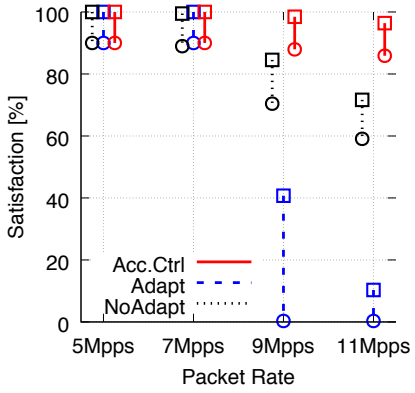
The satisfaction generally demonstrates similar trends for the two configurations. When comparing different measurement tasks, although *Acc.Ctrl* always achieves a considerable gain, more noticeable differences arise in terms of satisfaction ranges. For instance, in the case of Config.2 we observe the median satisfaction at the maximum packet rate oscillating between 60% and 100%. The reason for these differences is two-fold. Firstly, they depend on the precision of monitoring accuracy estimations, *e.g.,* overestimations usually prevent the accuracy control from fully recovering the real accuracy gaps. Considering *RTx* and *LatChange*, with accuracy estimation errors $\approx 10\%$ and



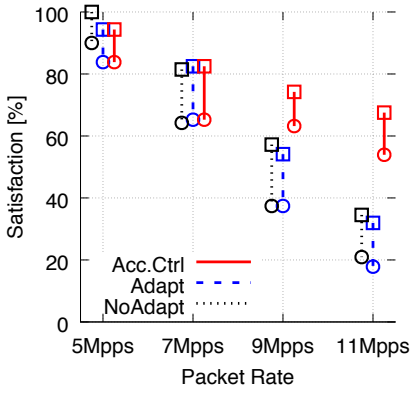
(a) Config.1, *HH*



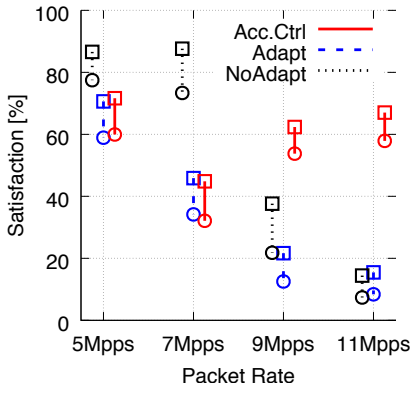
(b) Config.1, *RTx*



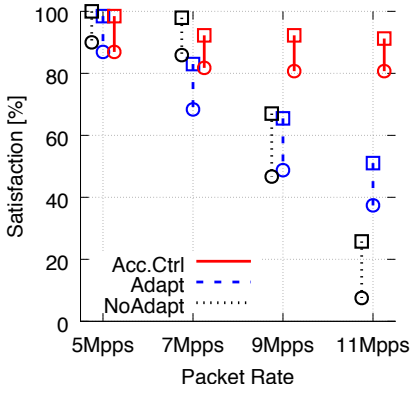
(c) Config.1, *Bursty*



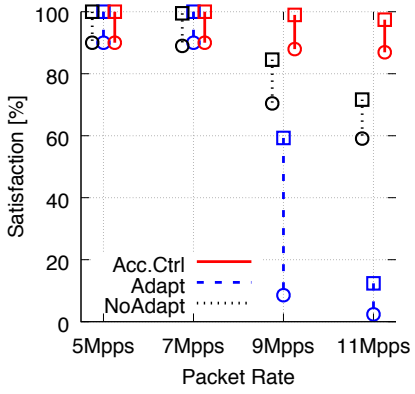
(d) Config.1, *LatChange*



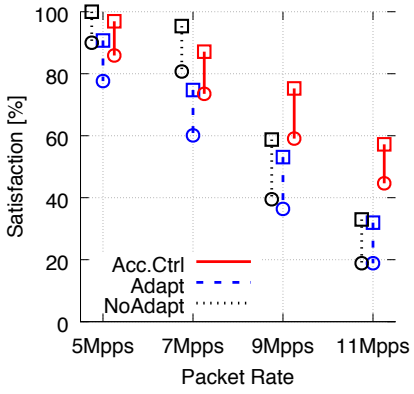
(e) Config.2, *HH*



(f) Config.2, *RTx*



(g) Config.2, *Bursty*



(h) Config.2, *LatChange*

Figure 4.12: Monitoring accuracy in terms of task satisfaction (Min ○, Median □)

$\approx 20\%$, respectively (see Fig.4.5), we obtain a satisfaction of 80/95% for the former and 70/60% for the latter. It is also evident that the very low estimation error for *Bursty* (less than 1% in Fig.4.5) relates to its very high median satisfaction. Secondly, the different satisfaction ranges reflect the different resource/accuracy trade-offs of different tasks. This explains the different behaviour, with lower satisfaction, of *HH*. In this case the concavity of the curve is such that few missing flows can translate to high ratio of undetected heavy hitters (*i.e.,* low Recall). Intuitively, this is because heavy hitters are evaluated based on the total size of multiple flows, and using a high threshold (5Mb for 10ms).

General validity of accuracy results The satisfaction results in Fig.4.12 have been obtained for a small set of monitoring tasks. However, these tasks are representative of a wide range of different monitoring objectives, which includes not only simple flow size or cardinality measurements (*e.g.*, *HH*), but also packet interarrival and traffic burstiness analysis (*e.g.*, *Bursty*) as well as TCP diagnosis (*e.g.*, *LatChange*, *RTx*). For all these different cases, MONA has been shown to achieve significant improvements of the satisfaction levels compared to both Adapt and No-Adapt approaches. Among the findings from Fig.4.12, the following three observations have general validity and can be leveraged for any monitoring task that is compatible with MONA (*i.e.,* whose output is a count or a classification, as mentioned in Section 4.5.1):

1. MONA outperforms the No-Adapt approach in terms of satisfaction levels and the gain achieved by MONA generally increases with the traffic speed, since increasing traffic rates correspond to additional packet loss in the No-Adapt case, which results in additional loss of monitoring information.

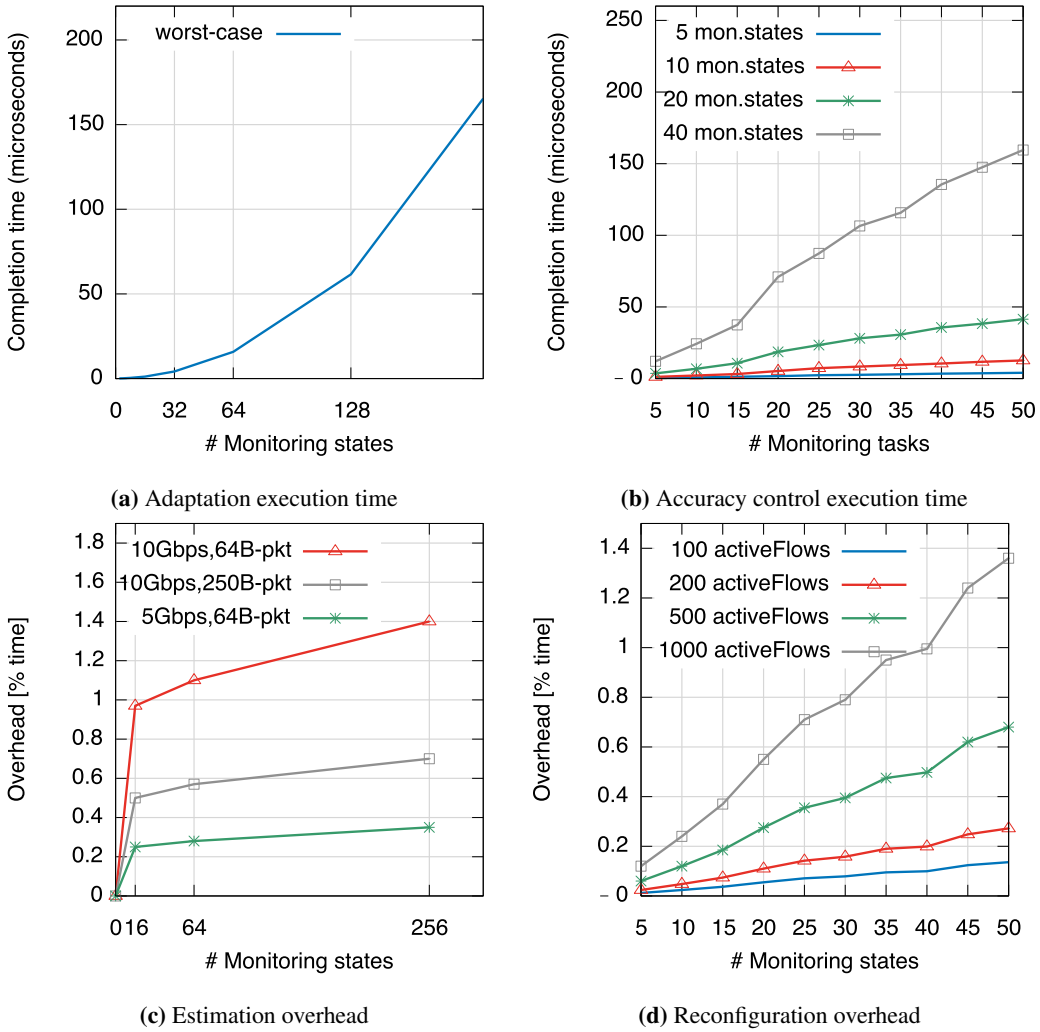
2. Higher accuracy estimation errors correspond to lower task satisfaction levels, since estimation errors result in incorrect inputs to the Accuracy Gaps Recovery routine.

3. The satisfaction level of a task depends on its resource/accuracy relation (*i.e.,* the task-specific curve of diminishing returns [56]). This curve relates the amount of monitoring information lost due to unobserved flows to the Recall reductions on which the satisfaction gaps depend.

Based on these observations, indications on the satisfaction level of a new task can be obtained by looking at the accuracy estimation errors and its curve of diminishing returns. The main problem, however, is that such information can hardly be known *a priori*. When this knowledge is not available, an experimental evaluation of the task satisfaction using ground-truth monitoring results is required as done for Fig.4.12.

4.6.5 Monitoring adaptation overhead

The cost of the proposed solution in terms of run time overhead is evaluated. To this end, we consider (i) the execution time of the main procedures running at the end of a time window, and (ii) the additional CPU time consumed throughout a time-window for the estimations presented



(a) Adaptation execution time

(b) Accuracy control execution time

(c) Estimation overhead

(d) Reconfiguration overhead

Figure 4.13: Evaluation of the main run-time overhead components in MONA

in Section 4.4.2 and 4.5.1, and for applying adaptation and accuracy control decisions. The latter implies a reduction in the sustainable packet-rate, while the former also translates to spikes of packets waiting at the input buffers.

Adaptation execution time This time corresponds to the adaptation routine completion. As depicted in Fig.4.13a, the worst-case (*i.e.,* adaptation minimises per-packet processing time) execution time in $O(k^2)$, with k being the number of monitoring states. However, even for a large value of k, *e.g.,* $k = 64$, the routine can still run to completion within a short period, in the range of $10\mu s$, resulting to only 100-150 packets being temporarily held at the input queue for 10 Gbps (much below the packet capture buffer size).

Accuracy control execution time This time is dominated by the completion of the accuracy recovery procedure. As shown in Fig.4.13b, this increases linearly with the number of measurement tasks k ($O(k)$), with the slope depending on the number of monitoring states defined. For a large number of tasks (*e,g.,* 50) and using 10 different monitoring states, this time is kept within the range of $10\mu s$.

Table 4.5: MONA feasibility based on total run-time overhead in % of CPU time

#Tasks	#States	#A.Flows	Traffic (bps/pktSize)	*Overhead* $< x$ $x = 1\%$	$x = 2\%$
5	5	1000	10G/64B	✓	✓
10	10	1000	10G/64B	✓	✓
20	10	1000	10G/64B	✓	✓
20	20	1000	10G/250B	✓	✓
40	20	1000	10G/250B	✗	✓
40	40	1000	10G/250B	✗	✗
40	40	500	10G/250B	✗	✓

Estimation overhead This metric groups different overhead components: (i) the time for counting the number of packets processed according to each monitoring state; (ii) the online estimation of P (probability of fast flow-entry retrieval) and (iii) the time consumed to compute μ_i and the loss function L. Results are shown in Fig.4.13c, where the consumed CPU time is expressed as a percentage of the 10ms time-window. While an increasing trend can be observed, the overhead is generally low and it exceeds 1% only for a large number of monitoring states (*e.g.,* 64) with constant 10Gbps traffic of 64B packets.

Reconfiguration overhead Fig.4.13d shows the additional time required to apply reconfigurations of the monitoring pipeline due to the adaptation routine or the accuracy recovery procedure. This overhead linearly depends on the number of monitoring states. Also, it relates to the number of active flows in the 10ms time window since the reconfigurations are enforced only once per flow. We can observe that this overhead exceeds 1% only for the maximum level of flow concurrency (1000), and for a large number of monitoring states (> 40). It should be noted that the range considered here for the number of active flows matches the statistics reported in [131] and [55].

Overall, the total overhead is a function of the number of measurement tasks and monitoring states, the packet rate and size, and the flow concurrency. Table 4.5 summarises the feasibility range of our solution given two maximum overhead constraints, 1% and 2% of the CPU time. As shown in the table, the total overhead exceeds 1% only under very large numbers of tasks and monitoring states, e.g., 40-40, and maximum traffic intensity. However, the requirement can be still met by relaxing the overhead constraint from 1% to 2%, which is acceptable, or by slightly reducing the hypothesis on traffic characteristics.

4.7 Limitations

The main limitations of MONA are discussed here. These relate to side effects of monitoring accuracy estimation, problems under attacks to the monitoring system, and lack of support for some categories of monitoring tasks.

Side effects of monitoring accuracy estimation In MONA, monitoring tasks whose results show higher coefficients of variation are prone to incur higher accuracy estimation errors (*i.e.,* LatChange

and RTx in Fig. 4.5). This effect can be imputed to how MONA accuracy estimation operates. More specifically, the estimation procedure is designed such that a conservative estimator, *i.e.,* the one providing an upper bound for the Risk function, is selected when the monitoring task results manifest a high degree of statistical dispersion (high coefficient of variation). This naturally favours under-estimation over over-estimation, which is essential to satisfy the global accuracy objective. However, in particular cases accuracy under-estimations can penalise the fairness of the Accuracy Gap Recovery process. Specifically, this issue arises when the monitoring accuracy is under-estimated for rich tasks. If this happens, rich tasks are less prone to redistribute their resources to the poor ones. This could prevent accuracy gaps to be fully recovered, thus leading to lower satisfaction levels.

Resilience to DoS attacks As discussed in Section VI.A, MONA can cope with much lower traffic skew compared to a non adaptive monitoring design. However, there are cases where traffic shows a "uniform" distribution (*i.e.,* with extremely low skewness), especially when denial-of-service attacks want to exhaust the resources of the monitoring system [8]. Uniform traffic can produce bottlenecks in MONA monitoring pipeline due to high sparsity in data accesses ($P \approx 0$) and, more important, to increasing flow insertion rates. This is because in MONA the formulation of the available per-packet time assumes no more than 10% of packets trigging a new flow-insertion (Sec. 4.4.1). Despite this, the effects of uniform traffic can be handled by applying the resilience mechanism proposed in Trumpet [8] in conjunction with MONA. Such mechanism consists in matching packets against an additional *filtering table* in ingress to the monitoring pipeline, and allowing the processing of a flow only if its size is above a DoS threshold. However, this comes at the cost of sacrificing a fraction of MONA packet-processing throughput due to the additional overhead introduced by the filtering table.

Support for monitoring tasks Given the approach used for estimating the monitoring report accuracy, MONA can support measurement tasks whose output is a count (*e.g., how many retransmissions for flow x?*) or a classification (*e.g., is aggregate y a heavy-hitter?*). A majority of the tasks studied in the recent literature [8][56][86], which are aimed at detecting network anomalies and/or used as input for network management decisions, satisfy this condition. However, there are still tasks that fall outside these premises, representative examples being *Flowlet size histogram* (*i.e.,* a histogram over the lengths of flowlets) and *EWMA over latencies* (*i.e.,* a moving average over packet latencies per flow) in [54].

4.8 State of the art overview

Adaptive traffic monitoring frameworks Generating accurate and fine-grained monitoring information at a reduced cost is a critical task in today's networks, especially in resource constrained environments. A number of recent proposals [23][56][86] have focused on the development of adaptive monitoring frameworks with the objective of supporting measurements under dynamic traffic patterns and resource availability. A novel adaptive flow counting approach was introduced in [23]

to enable anomaly detection with low overhead. Dynamic resource allocation solutions for traffic monitoring have been proposed in [56] and [86] based on OpenFlow counters and sketch-based measurements, respectively. Our approach also reconfigures monitoring parameters at run time to achieve efficient resource usage, but in contrast to the aforementioned solutions it focuses on software deployments.

Scalable packet capture and scheduling in software With the advent of packet processing on commodity hardware, previous efforts such as [68][114] investigated how to take advantage of multi-core architectures to minimize the packet processing times for sophisticated measurement tasks. While [68] relies on RSS and parallel threads to analyze multi-Gbps traffic, [114] uses multiple cores with the support of GPUs to perform complex intrusion detection operations. In addition, the recent packet-rate increase at the NIC raises new challenges, especially with respect to zero-loss guarantees under jitter in packet processing and unbalanced or unexpected traffic bursts. In contrast to our solution, existing approaches mainly address these challenges by enhancing either the packet capture or the packet scheduling. In [70], the authors propose to temporary store traffic in large buffers (1GB), which improves resilience at the cost of additional resource usage. The approach in [115] uses adaptive scheduling to mitigate performance drops due to resource contention.

Adaptive traffic monitoring in software dataplanes Orthogonal to scheduling and packet capture enhancements, recent solutions [82][117][28][8] address the problem of sustainable packet-processing in software by directly reconfiguring the monitoring process, as in the case of MONA. However, [82] and [117] focus on sketch-based measurements instead of hash tables where the main goal is to process all packets with fixed-size memory. The proposal in [82] augments the dataplane with a separate *fast path* that provides fast but slightly less accurate measurements under high traffic load, and recovers missing information via compressive sensing. The work in [117] learns the statistical distribution of the sketch and uses it to separate large and small flows so as to reduce the impact of hash collisions. Instead of focusing on sketches, MONA relies on simple hash tables for various reasons. One is the increased flexibility, as it enables more heterogeneous measurement tasks [8] and facilitates the design of stateful monitoring applications. Also, this approach has been shown to be more efficient than sketches [28] in terms of consumed time per packet, which is the main resource constraint in software packet-processing.

In a similar fashion to our solution, the methods presented in [28] and [8] also address the case of monitoring systems using simple hash tables to store flow statistics. In [28] the authors propose to adjust the size of the monitoring data-structure according to changes in the traffic properties. This can, however, incur significant time overhead for large flow-tables (structure size). The adaptive approach in [8] monitors only flows whose size exceeds a dynamic threshold, so as to handle denial of service attacks. While in [8] adaptations affect only new flows, reconfigurations in our work are applied to all operations in the monitoring process, responding thus to a wider range of emerging conditions.

4.9 Summary

This chapter has presented an adaptive and accuracy-aware traffic monitoring framework, MONA, tailored to software packet-processing pipelines. Current traffic monitoring solutions in software dataplanes can starve and lose packets at the packet capture buffers when changes in the operating conditions create bottlenecks. On the contrary, MONA guarantees efficient per-packet monitoring by jointly achieving zero packet loss and accurate monitoring reports.

To ensure lossless traffic monitoring, MONA timely reconfigures the operations of the monitoring process in the face of bottlenecks, based on accurate time estimations. To recover potential degradations of the monitoring reports, MONA quantifies the monitoring accuracy reductions at run time using a lightweight and widely-applicable estimation technique, and reconfigures the measurement operation sets so that a user-specified accuracy level can be met. As shown for representative measurement tasks, MONA produces reliable monitoring accuracy estimations, with errors below 8% on average, and can quickly recover potential accuracy gaps in less than 100ms.

The evaluation results have demonstrated that MONA functionalities can significantly enhance the traffic monitoring efficiency in face of bottlenecks and under dynamic traffic conditions. In particular, the MONA Adaptation function drastically reduces the risk of packet loss under various bottleneck conditions. It empowers the monitoring pipeline with the ability to handle traffic volume spikes up to 10Gbps in tens of milliseconds, without starving packets in the packet capture buffers. Similar resilience is guaranteed under 3x increase in concurrent accesses to shared server resources (shared processor cache), and for low levels of traffic skew. Furthermore, the evaluation has shown that MONA significantly enhances the monitoring accuracy levels, and that it is able to compute new monitoring configurations every 10 ms without the need of additional processor core(s). This is achieved with only minimal CPU time overhead (1-2%), even for 10 Gbps traffic of small packets, and 1000 active flows in 10 ms intervals.

Chapter 5

Classification-assisted Monitoring Query Processing

5.1 Overview

The focus of the previous chapters has been on the efficient collection of measurement data from the network traffic, their effective retrieval from the dataplane and their timely delivery to the management entities deciding on network reconfigurations. In particular, Chapter 4 has investigated how dataplane resources can be efficiently used to collect from traffic streams all the raw measurement data required for detecting a variety of network events and emerging conditions. Chapter 3 has instead addressed how to retrieve accurate measurement data from the dataplane at a low cost, as well as how to reduce data delivery latencies so that reconfigurations can more responsively be triggered by network controllers/managers. However, to meet the goal of an efficient monitoring functionality, another key challenge should be considered which concerns the *processing* of measurement data, *i.e.,* those operations required to build elaborate monitoring reports on traffic patterns and network episodes from granular (e.g., per-flow or per-packet) information sets.

Efficient measurement data processing is a key issue in today's monitoring systems, which are required to manipulate and analyse large amounts of heterogeneous data in short times. These stringent requirements can be imputed to two main conditions. The first one is that responsive network management and security applications require real-time monitoring updates, generated while processing the traffic streams, as opposed to offline trace analysis. Real time analysis is important to ensure minimal network reconfiguration latencies (e.g., for quickly addressing attacks), and it also allows monitoring systems to automatically and reactively drill down based on current conditions [8]. The second condition is the improved capability of monitoring systems to report on wide ranges of different network events [9] [8] [54], some of which (e.g., short-lived congestions) require to collectively analyse different types of measurement data [8].

Coping with massive amounts of data to be processed in real time can pose prohibitive costs in terms of computational resources. To mitigate such costs, one standard solution is to trade off

expressiveness for scalability [9], *i.e.,* to restrict the sets of supported monitoring results to ensure that measurement data can be processed using limited resources. As opposed to this, a different approach is taken in this chapter by investigating how the operations involved in the monitoring information processing can be performed in a more lightweight, cost-efficient manner. Improving data processing efficiency is key to enable timely and rich monitoring reports while handling massive traffic volumes.

More specifically, this chapter investigates the efficient processing of measurement data for the demanding case of modern *network telemetry* systems [67] [9] [10] [8], *i.e.,* monitoring systems generating sophisticated, real time reports in response to high-level monitoring commands issued by operators or network management applications. Network telemetry is generally designed to support tasks requiring continuous real-time measurements and analysis, from anomaly detection [132] [133] to traffic engineering [8]. Overall, these systems follow the principles of *software-defined measurements* [134] [34] [56]. They provide a high-level, declarative interface that hides measurement details to register monitoring requirements, and automatically configure measurement operations to satisfy hardware constraints. Specifically, they operate as *query processors* that receive declarative monitoring commands or *queries*, extract raw measurement data from the traffic streams, and process the data in real-time to identify a variety of network events and report them in query responses.

The increasing programmability of the dataplane has empowered network telemetry with the right tools to extract configurable sets of raw measurement data from packet streams at line rate. Hashing techniques and *sketch-based* approaches [78] [120] [34] [82] can now be deployed at the dataplane to collect measurement data with a limited computation and memory footprint. While these advances have contributed to the development of efficient data collection at the dataplane, the task of *processing* such amounts of data remains a costly operation. As networks move to higher speed and scale, it becomes crucial to reduce this cost in order to support large numbers of queries and massive traffic volumes.

The main methods proposed in the literature for improving data processing efficiency have mostly been focusing on ad-hoc design optimisations, specific to systems/implementations [8] [9] [54]. In contrast to previous work, this chapter presents a novel generalised approach to reduce the cost of query processing by intelligently filtering the raw measurement data collected through the monitoring pipeline. A new methodology is introduced, based on fast classifications to infer query results from small measurement subsets, and to take decisions on the portions of data that can be "proactively" filtered prior to the execution of standard data processing operations. In particular, the proposed approach uses lightweight classifiers that *learn* traffic properties based on recent measurements. To protect query responses from classification errors, the classifiers are automatically configured to discard decisions with low confidence levels, according to user requirements on query results accuracy.

To demonstrate the capabilities of the proposed approach, it has been integrated to a state-of-

the-art software packet-processing pipeline [8] [135] and tested with representative query examples and real traffic traces. The evaluation results, obtained through processing 30-minute traffic and using a short, 10-secs only, *training* set, have shown that large fractions of the extracted measurement data – more than 50% and even up to 90% in some cases – can be filtered out while satisfying accuracy requirements above 98%, leading to processing time reduction by up to a factor of 10.

The remainder of this chapter is organised as follows. Section 5.2 provides background information on monitoring queries, introduces the representative query examples used in this chapter, and presents the problem of reduced-cost query processing. Section 5.3 describes the proposed classification-assisted approach in detail, focusing on the different phases involved in the methodology. Section 5.4 evaluates the approach and Section 5.6 concludes the chapter.

5.2 Query-based network telemetry

Network telemetry systems must satisfy two fundamental requirements: (i) provide a generic, *declarative* interface based on the definition of monitoring *queries*, and (ii) cope with the complexity of query processing, from compilation to the final push of monitoring reports, while satisfying stringent monitoring accuracy goals.

5.2.1 Monitoring queries

Monitoring queries are declarative commands issued to identify a variety of events related to network performance and security. From a logical perspective, they entail combinations of three building blocks. The first is the *match & extract* component, which selects the portion of the network traffic being in the scope of the query, and extracts raw information based on packet size, header fields, or timestamp. The second one, *evaluate*, corresponds to the set of logical and arithmetic operations that check extracted information against a set of conditions (query predicates). The last component, *aggregate*, is the state aggregation function (e.g., sum, mean, count, stddev) applied to monitoring data for evaluation or reporting purposes.

These components can be used to build complex query-processing workflows, and *aggregate-evaluate* functions can be iterated in order to check different predicates at different levels of data aggregation. Assume, for example, an operator wants to detect hosts generating *bursty* TCP / UDP flows [8]. At the lower level of aggregation, this usually requires collectively considering the packet inter-arrivals of each 5-tuple flow (*aggregate*-1) and checking if more than *x%* of packets come in bursts (*evaluate*-1). Once all 5-tuple flows have been analysed, these are grouped by IP address (*aggregate*-2) and it is checked if at least *y%* of flows are bursty (*evaluate*-2).

Table 5.1 reports representative examples of monitoring queries well studied in the literature. Raw measurement data is extracted and stored for each 5-tuple flow as in [8]. Such data is periodically processed by *aggregate-evaluate* functions to produce query responses, based on a short (milliseconds or tens of milliseconds) query reporting period. These monitoring queries are further

Table 5.1: Representative monitoring queries

Query	Description	Processing workflow
Heavy hitters [8]	A traffic aggregate (srcIP) exceeding volume B_{hh}	**extract** 5-tuple bytes, srcIP; **aggregate** on srcIP (*sum* bytes); **evaluate** sum $> B_{hh}$
DDoS attack [34]	A host (dstIP) reached by more than K_{ddos} unique sources	**extract** 5-tuple srcIP, dstIP; **aggregate** on dstIP (*count distinct* srcIPs); **evaluate** count $> K_{ddos}$
Slowloris attack [9]	A host (srcIP) opening more than K_{sl} connections, with average rate below B_{sl}	**extract** 5-tuple bytes, srcIP; **aggregate** on srcIP (*mean* bytes, *count*); **evaluate** mean $< B_{sl}$ & count $> K_{sl}$
Bursty flow source [8]	A host (srcIP) generating more than $X\%$ *bursty* connections, *i.e.,* connections with more than $Y\%$ packets coming in bursts	**extract** 5-tuple, #pkts, #pkts-in-burst, srcIP; **aggregate** on srcIP (*isBursty()*); **evaluate** $\%(\text{isBursty} == 1) > x\%$

detailed below.

Heavy hitters (**HH**). This query returns all hosts (*i.e.,* source IP addresses) whose generated traffic exceeds a bytes threshold K_{hh} during the last query reporting interval. The results on heavy hitters are widely used by network management, e.g., for the detection of anomalies, to decide on traffic engineering configurations, to unveil server load *inbalance* in data centers.

DDoS attack (**DDoS**). This query returns DDoS victims [34], *i.e.,* those hosts (destination IP addresses) that have been contacted by more than K_{ddos} other hosts (source IP addresses).

Slowloris attack (**Slowloris**). Slowloris belongs to the category of "slow" denials of service. This attack consists in sending data over a large number of connections all with a very slow rate, without hitting the idle connection timeout value on the server. The query returns Slowloris-attack sources, *i.e.,* those source IP addresses generating more than K_{sl} connections, all with byte rate below B_{sl}.

Bursty flow source (**Bursty**) A TCP / UDP flow is *bursty* if more than $Y\%$ of its packets come in a burst [8], *i.e.,* with short interarrival times. This query returns bursty flow sources, *i.e.,* hosts (source IP addresses) for which more than $X\%$ of generated flows are bursty. The information about bursty flow sources can serve several network management tasks, for example the analysis of Incast [32] congestion (by correlating such information with packet loss).

5.2.2 Query processing

With networks evolving to larger scale and speed, the challenge for telemetry systems is being able to handle massive amounts of queries (e.g., 4K on a 10ms time window [8]), while facing both higher numbers of concurrent flows (1000+ on 10ms intervals [55]) and increasing packet rates (10Gbps+ on a single processor core in software-packet processing platforms [8] [82]). As a result, a vast amount of information needs to be processed to build query responses, especially for stateful monitoring [54] [8] (e.g., *groupby*-like state aggregation), with significant computational resources

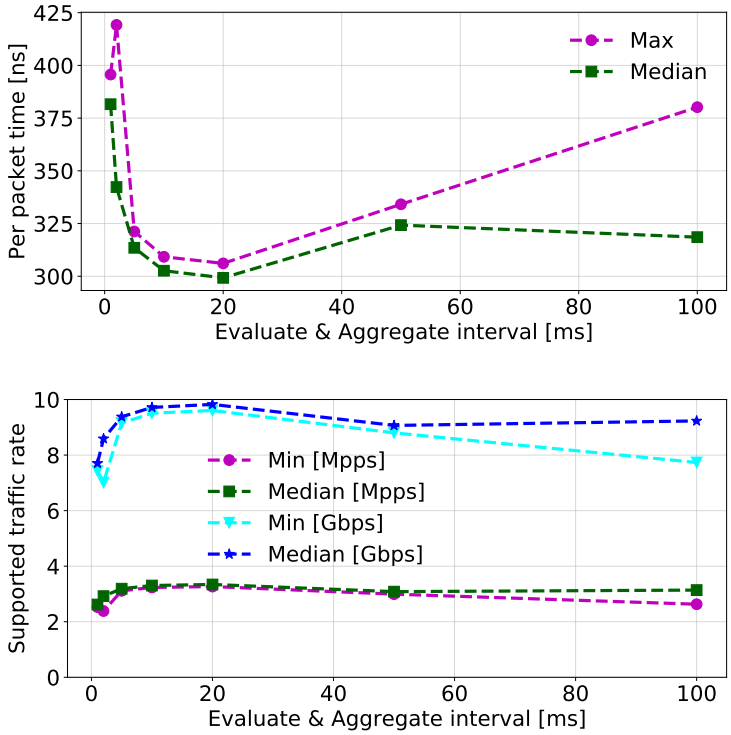


Figure 5.1: Effects of tweaking the reporting frequency on the per-packet time (top) and the supported traffic rate (bottom)

consumed for *aggregate* and *evaluate* execution.

Recent research has investigated how to efficiently match-on-traffic and extract raw data at line rate and with reduced memory footprints, using *sketch*-based techniques [78] [120] [34] [86] [82] [117], *heap*-based solutions (e.g., top-k [79]), or simple hash tables [8] [28]. These have emerged as standard solutions for the lightweight extraction of measurement data from packet streams. However, it is still unclear how to curb the cost of query processing when looking at the *aggregation* and *evaluation* of such amounts of measurement data. Ad-hoc design optimisations have been proposed to improve efficiency, e.g., [9] [54] [8], but these are mainly tailored to specific systems / implementations. Sonata [9] splits *aggregate-evaluate* workload between programmable switches and telemetry streaming aggregators at servers. Marple [54] relies on a memory backend on commodity hardware to facilitate stateful *group-by* operations. Trumpet [8], which is based on software packet-processing, adopts double-buffering of measurement data to interleave the *aggregate-evaluate* steps on previous data with *match & extract* on new traffic.

Generally speaking, different approaches can be explored to reduce the cost of query processing [136]. These can involve some relaxation on the *time*, *i.e.,* by tweaking the frequency at which raw data is evaluated and aggregated for the reporting of query results, or on the information *space*, *i.e.,* by filtering the original (raw) information sets considered in the *evaluate* and *aggregate* steps. While processing data at a reduced frequency is expected to cut the overall cost, in practice, the

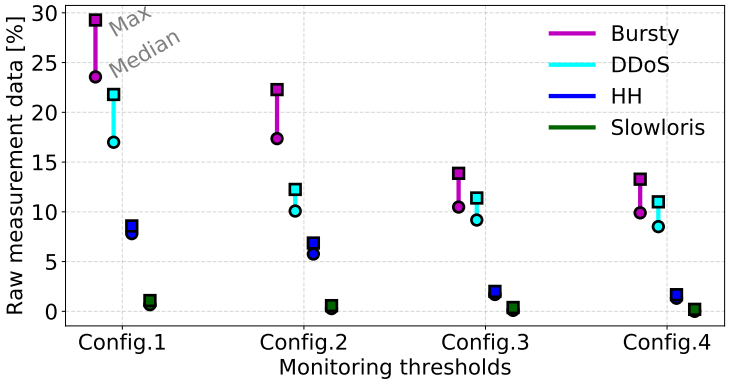


Figure 5.2: Quantification of the portion of raw data *retained* in query responses

potential benefit is undermined by the increasing accumulation of raw monitoring information over longer periods of time. In Fig. 5.1, the impact of relaxed query-results reporting is tested on a software packet-processing pipeline based on Trumpet [135], using a single CPU core[1]. As input, a ten-minute CAIDA trace from 2018 [137] is used, and the queries of Table 5.1 are applied. The cost of query processing is quantified by analysing how it affects the monitoring performance in terms of per-packet time and supported traffic rate. As shown in Fig. 5.1, increasing the reporting period from $1ms$ to $10ms$ (Trumpet default) and $20ms$ improves the maximum traffic rate by approx. 20% and 25%, respectively. Going beyond $20ms$, the performance level descends as more raw data accumulates prior to processing – more CPU cycles for iteration and memory access.

The second approach, *i.e.,* reducing the volume of monitoring data processed, is promising when only fractions of the raw measurement data are *retained* to craft query responses. In fact, this applies to all queries issued to detect events concerning unusual behaviours or specific traffic patterns. Fig. 5.2 shows the percentage of raw measurement data being used for query responses on the same setup of Fig. 5.1. To make query results more or less selective with respect to the total extracted measurement data, the thresholds of Table 5.1 are varied. More specifically, 4 threshold configurations are selected between the ranges $K_{hh} \in [1KB, 100KB]$, $K_{ddos} \in [10, 100]$, $X_{bursty} \in [10, 100]$, $K_{sl} \in [5, 50]$. As observed, despite differences between queries, large fractions of raw measurement data (65%+) could always be scrapped without influencing the query responses, with peaks above 90% for the most selective configurations.

As the example shows, there is potential for reducing the cost of query processing by filtering the raw measurement data before the *aggregate* and *evaluate* steps are executed. However, to exploit this, a few key challenges need to be addressed. First, a generalised, query-independent, approach should be provided to *anticipate* part of the query results, thus enabling intelligent filtering of the raw data. Second, such approach should be lightweight enough to guarantee processing cost reductions with respect to standard *aggregate* and *evaluate*. Finally, this should not compromise the validity of

[1] A 3GHz CPU with 8MB shared L3 cache is used

query responses in terms of query results accuracy. These challenges are addressed by introducing a novel, *classification-assisted*, query processing approach.

5.3 Classification-assisted query processing

To achieve intelligent filtering on the raw measurement data, the standard query processing workflow is enhanced with a *fast classification* functionality, whose goal is to reduce the volume of data used as input to the *aggregate* and *evaluate* steps. The proposed approach is based on the ability to derive classifiers for each query type in order to predict query results from subsets of the raw measurement data. In particular, classifiers are constructed by *learning* traffic properties based on recent measurements using a machine-learning approach.

The main reason for adopting a machine-learning approach is the high flexibility towards heterogeneous measurement data and monitoring queries. This is due to "generalisation", *i.e.,* the ability of machine-learning classifiers to perform accurately on new, unseen examples after having experienced a *learning* dataset. Moreover, this approach does not require any particular knowledge about the learning datasets. Even when the data come from unknown probability distributions, general models are automatically built to produce accurate predictions. Given these characteristics, a machine learning approach can adapt to any type of monitoring query without prior knowledge on it. While several techniques for predicting monitoring results can be found in the literature, these are generally tailored to specific monitoring queries. For example, *ProgME* [134] provides a sequential analysis method for fast Heavy Hitter identification based on the testing of probability ratios. In a similar fashion, the work in [35] defines an approach called *Threshold Random Walk* specific to the prediction of *Port Scan* attacks. In contrast with these solutions, the proposed ML-based approach can support a wide range of queries, and it can be extended to include new ones with minimal effort and without changes in the algorithm design.

5.3.1 Architecture

An overview of the approach is shown in Fig. 5.3. The core of the proposed solution is a set of classifiers, one for each query, which take samples of the raw measurement data as input and generate "decisions" on query-related events. For example, for a *heavy hitter* query, the classifier takes a sample containing the byte counts of few 5-tuple flows with the same srcIP x, and provides a probabilistic answer to the predicate"is x a heavy hitter?". This is achieved by performing *training* based on recent monitoring results, *i.e.,* using labelled samples (for which ground-truth is known) extracted from recent measurement data. The output of classifiers is the key to exclude portions of the raw measurement data from standard *aggregate* and *evaluate* processing. For instance, answering predicate "is x a heavy hitter?" allows raw counters matching srcIP=x to be discarded (*i.e.,* to filter data out), and in case of positive output to "proactively" add x to the query response.

The design of the classifiers follows two main principles. First, classifiers are built using

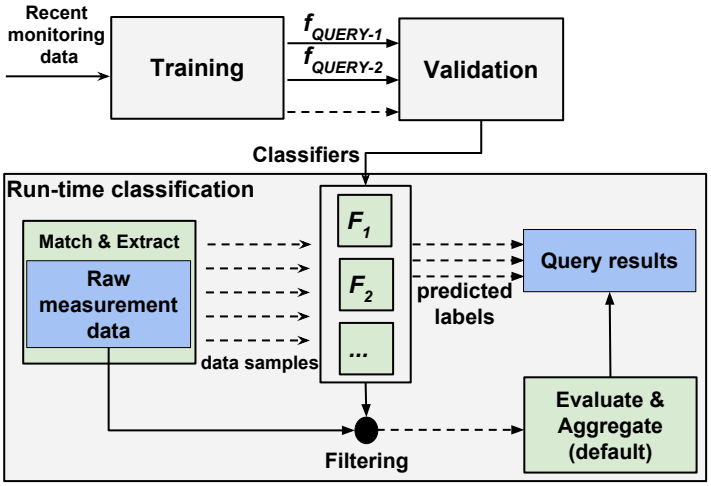


Figure 5.3: Overview of classification-assisted query processing

lightweight, computationally inexpensive, classification functions, and operate exclusively on small-sized samples. This ensures that the benefits obtained by reducing the data volume are not undermined by additional complexity when performing classifications. Second, classifiers are automatically configured to match user-specified requirements, specific to each query, based on the accuracy of query results. This operation is referred to here as *validation*. Since classification functions using samples are not error-free by definition, the validation ensures that query responses are protected from erroneous results. Based on such configuration, some classifier outputs can be disregarded when the estimated error probability is not negligible. More specifically, a classifier output falls under three cases: (i) the sample belongs to the query-related event (e.g., x is a heavy hitter); (ii) the sample is unrelated to the event, which corresponds to a negative result (e.g., x is not a heavy hitter); (iii) a decision cannot be made with a high level of *confidence* (possible error). Only in case (iii) portions of measurement data are directed to standard *aggregate-evaluate*.

Workflow As depicted in Fig 5.3, the proposed workflow includes three phases: *training* phase, where the classification functions are built; *validation* phase, where classifiers are configured to preserve query result accuracy; and *run-time classification* phase, where classifiers are run in the wild to take decisions on incoming traffic. These are detailed below.

5.3.2 Training

The training phase is divided in two steps: (i) *sampling*, which extracts training information (training sets) from recent measurement data, and (ii) construction of classification functions based on the obtained training sets.

Sampling Training is performed using recent sets of raw measurements, *i.e.,* from previous query reporting intervals, where all query results are computed through standard *aggregate-evaluate*. In particular, the training set is created by sampling recent sets of raw measurements and by associating each sample with its ground-truth label represented as a binary indicator, e.g., the sample

corresponds to a heavy hitter (1) or not (0)?

To make classifiers more efficient, sampling should ideally be a query-dependent operation. This is because the size of the samples and the way they are extracted (e.g., transformation of the raw measurement set) depend on specific query characteristics, such as its execution workflow or how selective it is. In this chapter, two sampling methods are proposed which cover a wide range of queries.

The first method is used when *aggregate-evaluate* contain a *cardinality*-based predicate, e.g., for *DDoS* in Table 5.1. In this case, a random subset of the raw measurements set is selected based on sampling factor k. The extracted values are then grouped on the *aggregate* key, e.g., the DstIP in DDoS detection, and each sample in the training set is a tuple containing the cardinality of a group and the associated ground-truth label. The second method covers cases where query predicates include aggregation functions such as *mean*, *sum*, *stddev* e.g., for *HH* in Table 5.1. In this case, the initial set is first grouped on the *aggregate* key, e.g., the srcIP for *HH*. For each group, K values are then randomly selected [2].

Classification functions Once formed, the training set is used to build the classification functions. These are modelled following the *sigmoid* function used for logistic regression. For each input sample they return: (i) the predicted label $l_{predicted}$ (1 if the sample participates to a query-related event, 0 otherwise), and (ii) the probability estimates p_0, p_1 associated with each label (with $p_0 + p_1 = 1$). Values of p_1 (p_0 respectively) indicate how likely a sample is to be labelled as 1 (0, respectively) in the ground-truth, and are used to quantify the *confidence* for individual classification decisions.

5.3.3 Validation

The goal of the *validation* phase is to configure the constructed classifiers so that accuracy requirements can be met on the query results. Since classification functions are not error-free, accepting all $l_{predicted}$ labels in the output may result in query result errors. This happens especially when the confidence on the classification is low, e.g., $(p_0, p_1) = (0.55, 0.45)$.

To configure the classifiers, we need to determine the right ranges of probability estimates p_0, p_1 under which the predicted labels $l_{predicted}$ in output from the classification functions can be "safely" accepted. More specifically, to express configurations, a probability threshold p_{thresh} is used, such that the classification result is accepted when $max(p_0, p_1) \geq p_{thresh}$. $p_{thresh} = 0.5$ corresponds to the baseline (all $l_{predicted}$ labels accepted). Intuitively, this setup maximises the filtering, *i.e.,* the amount of raw data excluded from *aggregate-evaluate* but, at the same time, it also maximises the risk of introducing errors in the query results.

The objective of the validation phase consists in finding appropriate p_{thresh} operating coordinates. This is achieved by conducting a small-scale sensitivity analysis for each classification func-

[2] In the experiments K is small, never exceeding 5

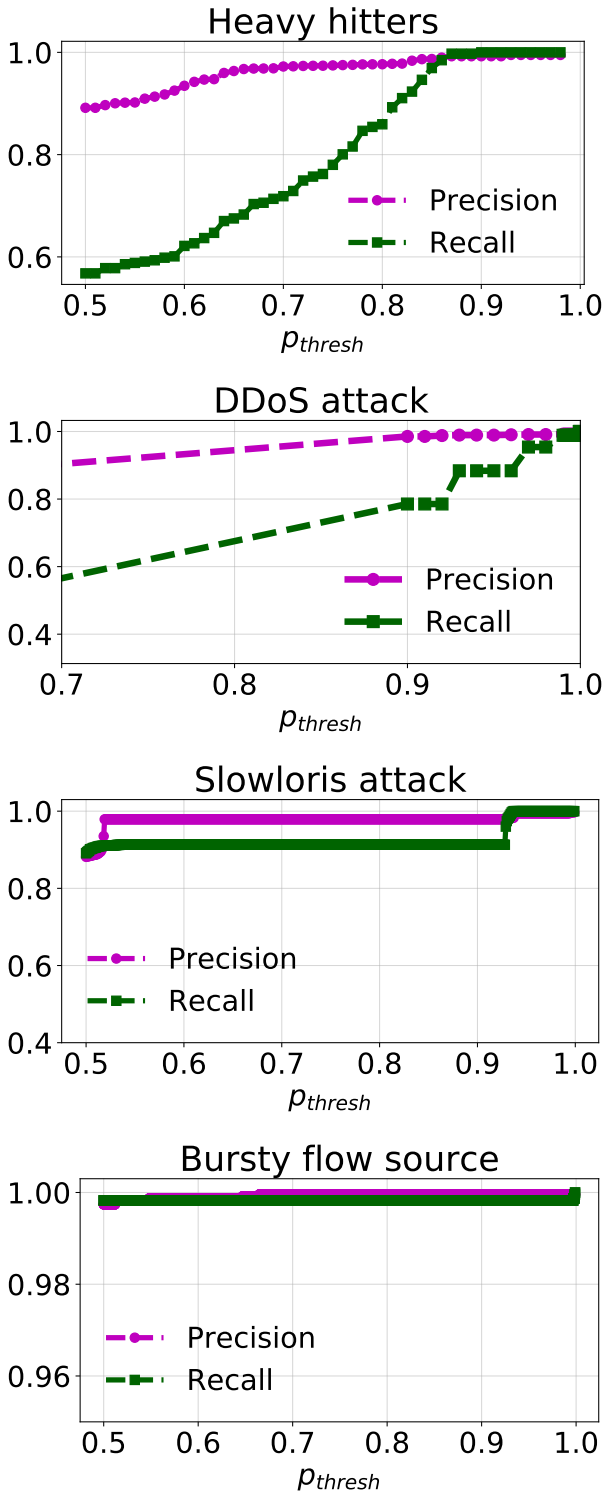


Figure 5.4: Precision and Recall analysis performed in *validation*

tion, using additional sets of labelled samples (*validation* sets) whose total volume is below 25% of the training set [3]. The accuracy of query results is quantified with the *Precision*, *i.e.,* the ratio of

[3]The choice reflects standard machine learning practices, where training/validation sets are 80/20 splits of total labelled-

the monitored query-related events that are *true*, and the *Recall*, *i.e.,* the fraction of detected *true* events. Fig 5.4 shows examples of the analysis for the queries in Table 5.1. As observed, the value of p_{thresh} can have a significant impact on the query result accuracy, especially on the Recall (many events missed by the queries). In addition, different queries follow different trends, which indicates that query-specific p_{thresh} settings are required.

Several methods can be considered to select the value of p_{thresh}. Main approaches in the literature [138] [139] rely on *Receiver Operator Curve* (ROC) analysis. The ROC depicts the accuracy performance of a classifier in terms of False Positive Rate (*i.e.,* the ratio of false positive results over the total number of negative ones) and True Positive Rate (*i.e.,* the Recall). Different values of the decision threshold (p_{thresh}) determine different operating points on the ROC, as such finding the optimal p_{thresh} corresponds to selecting the best ROC cut-off based on a optimality criterion. Widely-used approaches are, for instance, to select the cut-off that maximises Youden Index [140], or equivalently, the highest *Sensitivity+Specificity* point of the curve, where Sensitivity = Recall and Specificity = 1 - (False Positive Rate). However, any of these solutions is prone to incur a significant cost since for N monitoring queries, N optimal cut-offs should be computed (one for each monitoring query). To reduce the validation complexity, the proposed method operates by simply testing for each classifier a random set of p_{thresh} values in $[0.5, 1.0]$. After each test, the obtained Precision and Recall values (Fig 5.4) are compared against the Precision and Recall requirements (thresholds) of the query. The selected p_{thresh} value is the lowest for which both Precision and Recall meet the requirements. For example, in the case of *HH* and for 98% Precision and Recall requirements, a value $p_{thresh} \approx 0.9$ is selected in the example of Fig 5.4.

5.3.4 Run-time classification

Trained and configured classifiers are applied in the wild to pre-process the raw measurement data extracted for each active query. Specifically, at run time the classifiers generate decisions on query results by processing samples of the raw measurement data. This is referred to here as the *run-time classification* phase, which consists of two main operations. The first one is the sampling of measurement data to produce the input samples for the classifier, with each sample corresponding to a specific *aggregate key* of the query (e.g., a specific srcIP for HH detection). At run time, the sampling follows the same approach adopted in the Training phase (described in Section 5.3.2) that relies on two different methods, one for cardinality-based queries (where the *evaluate* predicate operates on a "count"), one for queries where *evaluate* operates on aggregated values such as mean, sum, standard deviation, etc. The second operation is to apply the classification function of the query to each input sample in order to predict whether the corresponding *aggregate key* is a positive query result or not. More specifically, for each sample the classification function returns a pair $[l_{predicted}, (p_0, p_1)]$. If $max(p_0, p_1)$ is above the p_{thresh} value selected for the query in the *validation*

samples set

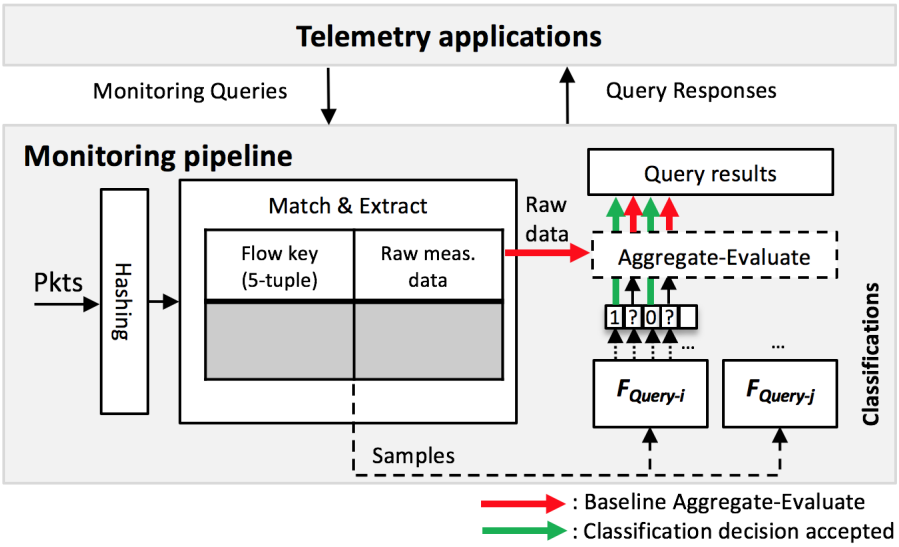


Figure 5.5: Implementation of classification-assisted query processing

phase, the result is accepted, *i.e.,* the predicted query result $l_{predicted}$ is kept and the portion of raw data matching the sample *aggregate key* is discarded. Otherwise, since the confidence on the classification is low, the same data are redirected to the *aggregate-evaluate* functions for standard processing.

5.4 Evaluation

This section investigates the benefits of the proposed classification-assisted query processing approach. Experiments are performed on a proof-of-concept implementation relying on a software packet-processing pipeline [8] [135], and they are based on the monitoring queries introduced in Chapter 5, *i.e.,* (i) Heavy hitters; (ii) DDoS attack; (iii) Slowloris attack; (iv) Bursty flow source.

The evaluation is conducted in three steps. At first, it quantifies the savings achieved by the proposed approach in terms of measurement data filtering for different monitoring queries. Then, it investigates the benefits of data filtering on the monitoring pipeline in terms of reduced query processing cost, focusing on query processing latencies and on the supported traffic speed. Finally, since the classifiers used by the proposed approach are not error-free by definition, the evaluation explores how effective the approach is in meeting stringent accuracy requirements on monitoring query results.

The experiment results demonstrate that the classification-assisted solution substantially improves the query processing efficiency. In particular, classifications can cut the data volume processed by *evaluate-aggregate* functions by more than 50%. This allows for significant reductions of the overall processing cost when crafting monitoring query responses, which can be achieved while maintaining 98% accuracy on all query results.

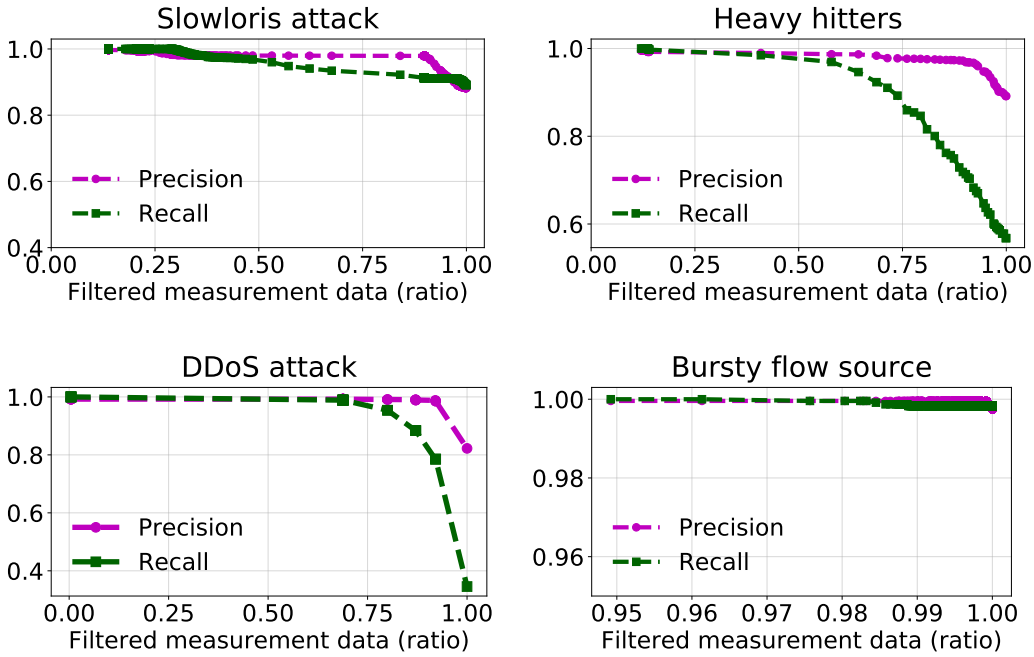


Figure 5.6: Raw measurement data filtering vs query result accuracy (Precision, Recall)

5.4.1 Implementation and evaluation setup

Classification-assisted query processing has been integrated with a traffic monitoring implementation based on software packet-processing (software dataplane), and running on a single CPU core. The tool is based on the framework in [8][135], and it relies on a hash table indexed on the flow 5-tuples to buffer raw measurements extracted from the traffic stream.

A representation of the implementation is shown in Figure 5.5. Query responses are generated in short reporting intervals T (default value, $T = 20ms$). At run time, measurement data samples (obtained from sampled flow-entries) are applied to the trained classifiers. If the classification decision is accepted, the flow-entries matching the *aggregate key* of the sample are excluded from further processing, and the predicted label is kept for the query response. For the remaining flow-entries, measurements are instead grouped by *aggregate key* and checked against the query predicates, which corresponds to the baseline *aggregate-evaluate* workflow.

The classification functions are trained using samples from a 10s CAIDA traffic trace [137], reserving 2s for *validation*. The classifiers are then used to process 30 minutes of CAIDA traffic without additional training. The sampling setup is fixed for the duration of the experiments with $k = 10\%$ and $K = 5$, where k is the sampling factor applied to the flow table, and K is the size of each sample expressed as number of flow-entries – for more details on sampling, the reader can refer to Section 5.3.2.

All experiments have been performed on an Intel i7-4790 CPU with 4 physical cores at 3.6 GHz and shared L3 cache of 8 MB.

5.4.2 Measurement data volume

At first, the performance of the approach is assessed in terms of the data volume involved in query processing. To this end, experiments with different p_{thresh} setups for each classifier are executed, measuring the effect of *filtered* data on *Precision* and *Recall*. The results are depicted in Fig. 5.6 and show that with an accuracy above 98% (Precision and Recall), $50, 55, 70$ and 98% of measurement data can be filtered for each of the four queries, respectively. The level of filtering achieved is significant, despite using a small 10s training set, but different query behaviours can be observed. These reflect a variety of query and traffic features (e.g., traffic distribution characteristics such as skewness and entropy) that make it more or less difficult to discriminate between positive and negative samples. Interestingly, the amount of data that can be safely filtered does not reflect the query *selectivity* level discussed in Sect. 5.2.2. While *Slowloris* is generally the most selective query in Fig. 5.2, no more than 50% of its measurement data can be filtered since confidence on the classifications is limited.

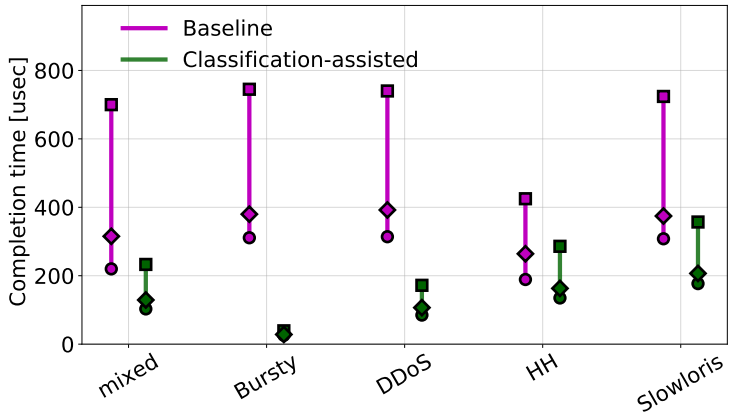
5.4.3 Query processing cost

Since the implementation is based on software packet processing, the processing cost is measured in terms of CPU time (t_{proc}) consumed to craft query responses from the raw data per reporting interval. To experiment with different query workloads, the flow-address space is split and each query is assigned to a /12 prefix. In the *mixed* workload the query is randomly selected from Table 5.1, while in the other workloads all queries are of the same type. For all experiments performed, the classifiers are configured to achieve an accuracy of 98% for both Precision and Recall.

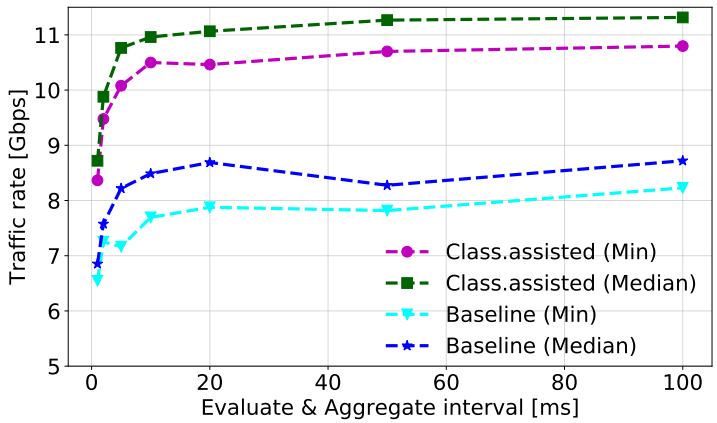
As shown in Fig. 5.7a, nearly 60% of CPU time is saved on average for a *mixed* workload, while the gain obtained for specific query types is in accordance with the filtering ratios in Fig. 5.6, e.g., the processing time for *Bursty* is reduced by more than 10x on average, as more than 90% of measurement data is filtered by classifiers. Reduced processing time can translate to more query responses handled over a reporting interval. In particular, assuming a constant time for raw data extraction[4], the guaranteed (minimum) number of simultaneous queries supported by our implementation is more than 3x higher than the baseline in *mixed* workload conditions.

Reduced query processing times can also improve the traffic speed supported by monitoring. To evaluate this, we split the experiment in 1 minute chunks, and for each one we measure the maximum traffic rate handled by the monitoring implementation (without dropping packets). As Fig. 5.7b shows, the classification-assisted approach can significantly speed up the monitoring pipeline, with gains in traffic rate up to 30%. Such gains (e.g., $+3$ Gbps) are higher than the ones obtained (see Sec.5.2.2) by fine-tuning the reporting interval ($\leq +1.5$ Gbps).

[4]In the implementation, this time is dominated by hashing and flow-entry retrieval executed for each packet, irrespectively of the query workload



(a) Total query processing time per interval $t_{proc.}$ (min, mean, max)



(b) Supported traffic rate

Figure 5.7: Benefits of the proposed approach on the monitoring pipeline used

5.4.4 Monitoring accuracy

Finally, we evaluate how effective the proposed solution is in meeting the requirements of query result accuracy. We select a target accuracy of 98% (Precision and Recall) to be used in the validation phase, and for each 1-minute chunk we compute the deviation from this threshold. For example, a $+0.01$ Recall deviation corresponds to 99% Recall. As shown in Fig. 5.8, the proposed approach satisfies, on average, the desired accuracy levels for all queries. Negative deviations, if any, never exceed -0.02. Interestingly, this result is based on a small $10s$ training set.

5.5 Limitations

Implementations of the proposed classification-assisted approach in real network telemetry systems can face problems concerning (i) the range of supported monitoring queries and (ii) the classifier (re)training. These are discussed below.

Query support limitations A key advantage of the classification-assisted approach is that it can be

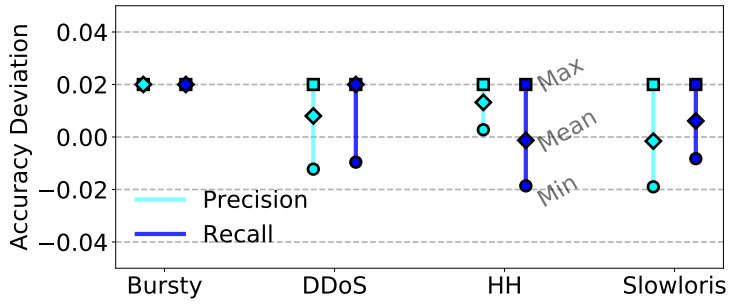


Figure 5.8: Accuracy deviations from 98% target obtained with 10s training

applied to a new query without requiring prior knowledge on it. However, for a new query to be supported by the proposed solution, two requirements must be satisfied. Firstly, the final output of the query *predicates* (*i.e.,* those conditions in the *evaluate* block of the query) should be a binary indicator (e.g., is src IP x a heavy hitter (1) or not (0)?), since the proposed machine learning-based workflow only supports binary classification. Secondly, the query *evaluate* conditions should be on a count (as in case of *DDoS attack* query) or on an aggregate value such as mean, sum, median, etc. While most of the monitoring queries considered in recent work [9] [8] [10] [54] satisfy these two conditions, others fall outside these premises. Representative examples are the *Flowlet size histogram* query in [54], which returns the histogram over length of flows, and the *Lifetime of connections* query in [10], which provides the durations of TCP connections. Both queries cannot be supported by classification-assisted query processing.

Training limitations The results in Section 5.4 have shown significant processing cost reductions and high monitoring accuracy for real traffic [137]. These results have been obtained with a single, 10-seconds only training and then using the same classifiers over the full experiment duration (30 minutes) with no retraining. However, in real network telemetry systems the classifiers should be retrained to handle traffic dynamics so as to keep ensuring high confidence on the classification output when traffic patterns change, which is important to guarantee high monitoring accuracy. A limitation of the proposed design is the lack of an online mechanism deciding on the re-training intervals based on traffic variations. Network telemetry operators can configure a re-training period for each query, however using a constant training frequency can result in limited information filtering ratios and/or low accuracy.

5.6 Summary

This chapter has tackled the problem of reducing the cost incurred by data processing when creating monitoring reports (query responses) from the raw measurement data extracted at the dataplane. To this end, a novel, classification-assisted, query processing approach has been presented which achieves intelligent filtering of measurement information using lightweight classifiers. This solution,

inspired by machine learning workflows, can support a wide range of monitoring queries, and is not tailored to a specific monitoring system design(s).

The proposed approach has been evaluated using representative query examples and real traffic traces, and it has been implemented on a realistic monitoring pipeline based on software packet-processing. The experimental results have shown that the classification-assisted approach can substantially reduce the cost of monitoring query processing based on intelligent filtering of the measurement data. In particular, it has demonstrated that at least 50% of the measurement data can be filtered through fast classifications, with data filtering levels above 90% for specific query types, while keeping the accuracy levels of query responses above 98% on average. The filtering translates into faster query processing, with 60% less CPU time needed on average to craft query responses from the raw measurement data. The improved query processing efficiency can allow for (i) larger numbers of simultaneous monitoring queries (3x more queries under *mixed* workload conditions), (ii) higher traffic rates supported (up to 30% increase in Gbps). Such benefits have been achieved using no more than 10-seconds classifier training for processing 30-minutes traffic.

Chapter 6

Conclusion and Future Research Directions

6.1 Overview

Traditional monitoring solutions operate on long timescales producing periodic reports, which are mostly used for manual and infrequent network management tasks. These practices have been recently questioned by the advent of software-defined networking, which has enabled automatic, frequent, and fine-grained network reconfigurations the effectiveness of which strictly depends on the accuracy and timeliness of monitoring reports. However, ensuring timely and precise monitoring updates is not a trivial task, especially when dealing with large network scales, massive and dynamic amounts of traffic, and stringent constraints on time and hardware resources. It requires the monitoring functionality to cope with increasing network sizes and traffic rates, and to achieve an efficient use of the available time and hardware resources.

This PhD thesis has addressed these research problems by investigating (*i*) how to perform traffic measurements using dataplane resources efficiently, (*ii*) how to efficiently extract the measurement data from the dataplane, (*iii*) how to transform this data into relevant monitoring knowledge at a reduced cost while avoiding processing bottlenecks, and (*iv*) how to deliver knowledge to network decision-making processes in a time-efficient manner. Concerning the first goal, *i.e.,* the efficient use of dataplane resources, this thesis has shown that adaptive monitoring functions can be integrated with the packet-processing pipeline to preserve monitoring report accuracy while facing the dynamic resource availability in the dataplane. Such adaptive functions have been shown to significantly improve the monitoring resilience under adverse operating conditions and in face of bottlenecks in the packet-processing pipeline. In regard to the second goal, *i.e.,* the efficient measurement data extraction from the dataplane, it has been demonstrated that valid tradeoffs between monitoring precision and resource consumption, generally outperforming the results of existing techniques, can be automatically achieved using a self-tuning approach that entails only minimal parameter configurations. Considering the third objective, *i.e.,* reduced cost processing of measurement data, this thesis has proven that the cost for aggregating and evaluating huge amounts of monitoring data can be substantially reduced without significantly penalising the accuracy of monitoring results. This can be

obtained by performing lightweight classification tasks on small subsets of the measurement data. Concerning the last research goal, *i.e.,* timely monitoring information delivery, it has been shown that a distributed monitoring approach significantly helps reducing monitoring-induced delays especially when the decisions on network reconfigurations can be taken close to where the monitoring data is collected.

6.2 Thesis summary

In this section, the core chapters of the thesis are summarised, focusing on the main monitoring advances proposed and the key experimental findings.

Chapter 3 has presented a decentralised and self-adaptive monitoring framework for SDN. The proposed framework relies on a modular architecture designed to satisfy the diverse monitoring requirements of heterogenous management applications. Its decentralised nature, which differentiates it from state-of-the-art SDN monitoring systems, allows to avoid processing bottlenecks and to maintain high levels of monitoring reactivity (*i.e.,* knowledge delivery on short timescales) in the case of large-scale networks. To efficiently extract measurement data from the network switches, the framework implements *SAM* (Self-tuning Adaptive Monitoring), an adaptive solution that automatically reconfigures its setup under dynamic traffic conditions. Experimental results have demonstrated that the proposed framework can achieve considerable reductions of the monitoring latencies, as well as significant gains in terms of monitoring overhead without affecting the management application performance. Furthermore, these have shown, based on both synthetic and real traffic traces, that SAM outperforms state-of-the-art adaptive algorithms in terms of both monitoring precision and resource consumption on switches.

Chapter 4 has introduced MONA, an adaptive framework for software *dataplanes* which ensures resilience to bottlenecks while maintaining the accuracy of monitoring reports above a user-specified threshold. MONA dynamically reduces the measurement task sets under adverse conditions, and reconfigures them to recover from potential accuracy degradations. To quantify the monitoring accuracy at run time, MONA adopts a novel task-independent technique that generates accuracy estimates according to recently observed traffic characteristic. MONA is unique in its design, as it is the first approach, fully tailored to software dataplanes, where measurement operations are dynamically configured to achieve accuracy goals. Furthermore, it represents, to the best of the author's knowledge, the first attempt to coordinate a number of different measurement tasks by looking at their CPU-time consumption instead of (well-studied) memory issues. The results of MONA evaluation have shown considerable gains in performance, in terms of both improved resilience to bottlenecks and enhanced monitoring accuracy levels for a diverse set of representative measurement tasks.

Chapter 5 has presented a generalised approach for reducing measurement data-processing costs, *i.e.,* those costs, in terms of data aggregation and evaluation, incurred by monitoring systems

when responding to monitoring queries based on the raw information extracted at the dataplane. Different to state-of-the-art solutions, which are mainly tailored to specific systems/implementations, the proposed approach does not depend on particular monitoring designs and is applicable to a wide range of monitoring queries. It relies on classifiers that *learn* recent traffic properties and apply this knowledge to generate monitoring responses from subsets of the measurement data. The experiment results have demonstrated that the proposed solution significantly reduces data processing costs while ensuring accurate monitoring reports.

6.3 Future directions

The monitoring advances presented in this thesis achieve the goal of improved timeliness and accuracy of monitoring reports while facing stringent constraints on time and hardware resources. These benefits are key for handling (*i.e.,* extracting, processing, delivering) huge amounts of heterogeneous monitoring data, which is expected to be an essential requirement of future network infrastructures. Indeed, the evolution of dataplanes towards higher programmability and flexibility allows for more, different types of measurements to be collected from traffic streams. At the same time, monitoring data is expected to grow in volume due to higher network scales and increasing traffic rates. The solutions offered by this thesis are very helpful for facing these challenges. In particular, the architecture presented in Chapter 3 allows the monitoring functionality to cope with increasing network sizes, the adaptive functions in Chapter 4 enable heterogeneous monitoring data to be collected in a resource-efficient manner from high-speed traffic, and the approach introduced in Chapter 5 allows for more efficient processing of large monitoring datasets.

In future networks, management processes can benefit from these solutions to obtain all the necessary knowledge for reconfiguring the network on short timescales and at high levels of granularity. This is essential for recovering from network performance issues quickly and with very precise and selective countermeasures, which enables the introduction of new services that are extremely time-sensitive and heavy in resource consumption, such as virtual reality applications and holographic communications. At the same time, this allows to immediately detect and address attacks before users and services are adversely impacted, which is important for meeting the increasing demand of services for security guarantees.

Furthermore, this thesis has provided key tools for reducing the monitoring costs by improving the monitoring resource-efficiency, which is fundamental for facing the increasing demand of operators for real-time network telemetry [9] [10]. In particular, the switch polling mechanism proposed in Chapter 3 allows to extract measurement data with reduced consumption of the switch resources. By performing dynamic and intelligent reconfigurations of the measurement operations, the adaptive solutions in Chapter 4 reduce the amount of resources (CPU cores, in particular) to be devoted to traffic monitoring. Lastly, the methodology in Chapter 5 opens the door to the online execution of network telemetry tasks of increasing complexity, leveraging machine learning techniques to

produce elaborate and accurate monitoring reports in a much more lightweight manner.

Despite the improvement of the state of the art achieved by this thesis, there is plenty of room for further work to extend the proposed solutions and address their main limitations. The following subsections present potential future research directions based on the work in this thesis.

6.3.1 SDN monitoring framework

The decentralised framework introduced In Chapter 3 allows for reduced monitoring delays in software-defined networks and for the efficient extraction of monitoring information from OpenFlow-enabled switches. Future work can investigate how to enhance the overall framework functionality with solutions that further reduce the monitoring *overhead* on the network resources. Specifically, two research directions can be followed.

One direction can be the design of adaptive algorithms for the efficient synchronisation of monitoring data between the different MMs (Monitoring Modules) operating in the network. Three main conditions motivate this research line. First, such solutions would naturally fit within the proposed monitoring architecture, which already includes a programmable synchronisation interface as part of each MM to enable advanced information exchange policies between MMs. Second, the evaluation results have clearly made the point for the deployment of these algorithms, as they demonstrated that substantial reductions of the synchronisation overhead can be achieved at run time without causing significant performance degradations in the network. The last factor is the recent shift of measurement systems towards *stream-based* solutions [93] [9], which can potentially flood the network with heavy streams of measurement data and as such pose the need for intelligent exchange of monitoring information.

A second research direction can be on extending the range of solutions to efficiently extract monitoring information from the network devices, in particular with the aim to support novel programmable switch architectures (e.g., PISA) and protocols (e.g., P4). The SAM algorithm proposed in Chapter 3 adopts a *pull-based* approach, where switches are explicitly polled for monitoring information, which fits well with the metrics extracted from OpenFlow-enabled switches – essentially, per-port and per-flow counters. Novel programmable dataplane technologies [17] [141], however, enable more complex and "stateful" measurement operations to be performed at the dataplane. Such enhanced programmability can potentially be exploited to further reduce the volume of measurement data collected from the switch, but this requires different, *push-based*, adaptive algorithms whose logic should be on the network device rather than on the MM.

6.3.2 Adaptive and accuracy-aware monitoring

The MONA framework presented in Chapter 4 allows to efficiently coordinate the execution of different measurement tasks running in a software dataplane. This is achieved at run time by reconfiguring measurement operations to obtain high global monitoring accuracy and avoid bottlenecks. A recent research trend has made the point for extending a similar rationale to the case of traffic

measurements on novel programmable switches. In particular, monitoring systems have started using the P4 protocol and the PISA switch architecture to perform a variety of elaborate measurement tasks in hardware, key examples being Sonata [9], Marple [54], and In-band Network Telemetry [59] frameworks. Accuracy-aware mechanisms need to investigated, developed, and integrated to these frameworks to dynamically allocate switch resources in response to increasing monitoring demand (e.g., more concurrent measurement tasks) or increasing monitoring state (e.g., more memory or computation required by specific measurement tasks). This problem is challenging, as the execution of complex measurement tasks on these devices can incur serious bottlenecks on switch resources in terms of both memory and processing *stages*, e.g., available match-action pipelines in a P4-enabled device.

6.3.3 Classification-assisted query processing

The approach in Chapter 5 enables reduced-cost monitoring query processing by relying on intelligent filtering of measurement data. The methodology proposed is meant to be generic with respect to the design/implementation of monitoring systems, as it focuses on data processing operations, such as aggregation and evaluation of monitoring query predicates, that are general attributes of query-based monitoring systems. However, to enable such a methodology to run on different monitoring implementations, a few practical challenges need to be addressed. The main one concerns how to implement the required processes to build the classifiers, *i.e.,* training and validation analysis, for different monitoring system designs. Future work will explore how this can be achieved, especially to support the case of traffic measurements running on commodity switches or novel programmable switch architectures.

6.4 Concluding remarks

Providing timely, granular and precise information is becoming extremely challenging for monitoring systems given the evolution of networks towards higher scales and traffic volumes, and due to stringent constraints on time and hardware resource availability. This PhD thesis has addressed this important challenge in the context of software-defined networks and dataplanes based on software packet-processing. Novel methodologies, designs, and frameworks have been introduced, which allow for a more scalable monitoring functionality and improve the key tradeoffs between monitoring information accuracy and resource consumption.

Overall, the solutions presented in this thesis take a new step towards the goals of efficient measurement collection and reporting, lightweight measurement data processing, and timely monitoring knowledge delivery. These solutions contribute to the effort for ensuring accurate and timely monitoring reports, building the *knowledge* required for reconfiguring network resources quickly and in a precise manner. With the adoption of Software-Defined Networking principles in future network designs, such monitoring knowledge will be more important than ever for the effective management

of networks. Indeed, this will be the enabler for automatic management processes operating on short timescales and warranting precision when troubleshooting failures, detecting anomalies, or adapting the behaviour of demanding services.

Bibliography

[1] D. Tuncer, M. Charalambides, S. Clayman, and G. Pavlou. Adaptive resource management and control in software defined networks. In *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 18-33, Mar. 2015.

[2] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. 2012. Procera: a language for high-level reactive network control. In *Proc. ACM HotSDN*, 2012.

[3] C.Monsanto, J.Reich, N.Foster, J.Rexford, and D.Walker. Composing software defined networks. In *Proc. USENIX NSDI*, 2013.

[4] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: fine grained traffic engineering for data centers. In *Proc. ACM CoNEXT*, 2011.

[5] Mohammad Alizadeh et al. CONGA: distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review* 44, 4, Aug. 2014.

[6] Cisco Netflow, https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html

[7] Tcpdump official website, https://www.tcpdump.org/

[8] M. Moshref, M. Yu, R. Govindan, A. Vahdat. Trumpet: timely and precise triggers in data centers. In *Proc. ACM SIGCOMM*, 2016.

[9] A. Gupta et al. Sonata: query-driven streaming network telemetry. In *Proc. ACM SIGCOMM*, 2018.

[10] Yifei Yuan et al. 2017. Quantitative Network Monitoring with NetQRE. In *Proc. ACM SIGCOMM*, 2017.

[11] Minlan Yu. 2019. Network telemetry: towards a top-down approach. In *ACM SIGCOMM Computer Communication Review* 49, 1, Feb 2019.

[12] N. McKeown et al. OpenFlow: enabling innovation in campus networks. In *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 69-74, Apr. 2008.

[13] The *Click* modular router. https://github.com/kohler/click.

[14] Ben Pfaff et al. The design and implementation of open vSwitch. In *Proc. USENIX NSDI*, 2015.

[15] DPDK, http://dpdk.org/.

[16] L. Rizzo. NETMAP: A novel framework for fast packet I/O. in *Proc. USENIX ATC*, 2012.

[17] P. Bosshart et al. P4: programming protocol-independent packet processors. In *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87-95, Jul. 2014.

[18] P. Bosshart et al. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proc. ACM SIGCOMM*, 2013.

[19] H. Kim and N. Feamster. Improving network management with software defined networking. In *IEEE Communication Magazine*, vol. 51, no. 2, pp. 114-119, Feb. 2013.

[20] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proc. ACM HotSDN*, 2013.

[21] L. Hendriks, R. Schmidt, R. Sadre, J. Bezerra and A. Pras. Assessing the quality of flow measurements from OpenFlow devices. In *Proc. TMA*, 2016.

[22] J. C. Mogul et al. Devoflow: cost-effective flow management for high performance enterprise networks. In *Proc. ACM Hotnets*, 2010.

[23] T. Zhang. An adaptive flow counting method for anomaly detection in SDN. In *Proc. ACM CoNEXT*, 2013.

[24] T. Koponen et al. Onix: a distributed control platform for large-scale production networks. In *Proc. USENIX OSDI*, 2010.

[25] Z. Bozakov, A. Rizk, D. Bhat, and M. Zink. Measurement-based Flow Characterization in Centrally Controlled Networks. In *Proc. IEEE INFOCOM*, 2016.

[26] C. Yu et al. Software-defined latency monitoring in data center networks. In *Proc. PAM*, 2015.

[27] C. Yu et al. FlowSense: monitoring network utilization with zero measurement cost. *Proc. PAM*, Hong Kong, China, Mar. 2013, pp. 31-41.

[28] O. Alipourfard, M. Moshredf, M. Yu. Re-evaluating measurement algorithms in software. In *Proc. ACM Hotnets*, Philadelphia, PA, USA, Nov. 2015.

[29] Y. Yuan, R. Alur, and B. T. Loo. NetEgg: Programming network policies by examples. In *Proc. ACM Hotnets*, Los Angeles, CA, USA, Oct. 2014.

[30] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proc. NSDI*, San Jose, CA, USA, Apr. 2010.

[31] S. Agarwal, M. Kodialam, and T. Lakshman. Traffic engineering in software defined networks. In *Proc. IEEE INFOCOM*, Apr. 2013.

[32] Y. Chen, R. Griffith, J. Liu, R. H. Katz, A. D. Joseph. Understanding TCP Incast throughput collapse in datacenter. In *Proc. ACM WREN*, Barcelona, Spain, Aug. 2009, pp. 73-82.

[33] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. BLINC: multilevel traffic classification in the dark. In In *Proc. ACM SIGCOMM*, 2005.

[34] M. Yu et al. software defined traffic seasurement with OpenSketch. In *Proc. USENIX NSDI*, Lombard, IL, USA, pp. 29-42, Apr. 2013.

[35] Jaeyeon Jung, V. Paxson, A. W. Berger and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proc IEEE Symposium on Security and Privacy*, 2004.

[36] M. H. Bhuyan, D. K. Bhattacharyya and J. K. Kalita. Network Anomaly Detection: Methods, Systems and Tools. In *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 303-336, First Quarter 2014.

[37] P. Garca-Teodoro, J. Daz-Verdejo, G. Maci-Fernndez, and E. Vzquez. 2009. Anomaly-based network intrusion detection: Techniques, systems and challenges. In *Computer Security*, 28, 1-2, Feb 2009.

[38] M. Malboubi, "Optimal-Coherent Network Inference (OCNI): Principles and Applications," in *IEEE Transactions on Network and Service Management*, vol. 15, no. 2, pp. 811-824, June 2018

[39] How To Calculate Bandwidth Utilization Using SNMP. https://www.cisco.com/c/en/us/support/docs/ip/simple-network-management-protocol-snmp/8141-calculate-bandwidth-snmp.html

[40] Jacob Strauss, Dina Katabi, and Frans Kaashoek A measurement study of available bandwidth estimation tools. In *Proc. of the 3rd ACM SIGCOMM conference on Internet measurement (IMC)*, 2003.

[41] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: Existing techniques and new directions. In *Proc. ACM SIGCOMM*, 2002.

[42] Q. Zhao, Z. Ge, J. Wang, and J. Xu. Robust traffic matrix estimation with imperfect information: Making use of multiple data sources. in *Proc. ACM SIGMETRICS*, 2006.

[43] J. Gong et al. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. In *Proc USENIX ATC*, 2018.

[44] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *Proc IEEE INFOCOM*, 2016.

[45] Parveen Patel et al. Ananta: cloud scale load balancing. In *Proc. ACM SIGCOMM*. 2013.

[46] Qi Huang et al. Characterizing Load Imbalance in Real-World Networked Caches. In *Proc. ACM HotNets*, 2014.

[47] Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/

[48] Z. Liu et al. One sketch to rule them all: rethinking network flow monitoring with UnivMon. In *Proc. ACM SIGCOMM*, Florianopolis, Brasil, Aug. 2016, pp 101-114.

[49] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better NetFlow for data centers. In *Proc. USENIX NSDI*, 2016.

[50] eXpress Data Path. https://www.iovisor.org/technology/xdp

[51] Open vSwitch, https://www.openvswitch.org

[52] Sean Donovan and Nick Feamster. 2014. Intentional Network Monitoring: Finding the Needle without Capturing the Haystack. In *Proc. ACM HotNets*, 2014.

[53] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. 2016. Compiling path queries. In *Proc. USENIX NSDI*, 2016.

[54] S. Narayana et al. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. ACM SIGCOMM*, Aug. 2017.

[55] A. Roy et al. Inside the social network's (datacenter) network. In *Proc. ACM SIGCOMM*, London, UK, Aug 2015, pp. 123-137.

[56] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Dream: dynamic resource allocation for software-defined measurement. In *Proc. ACM SIGCOMM*, Chicago, IL, USA, Aug. 2014, pp. 419-430.

[57] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proc. ACM SIGCOMM*, 2016.

[58] sFlow, https://sflow.org/

[59] C. Kim et al. In-band network telemetry via programmable dataplanes. In *Proc. ACM SIGCOMM*, London, UK, Aug. 2015, Demo Session.

[60] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: traffic matrix estimator for OpenFlow networks. In *Proc. PAM*, Zurich, Switzerland, Apr. 2010, pp. 201-210.

[61] N. Van Adrichem, C. Doerr, and F. Kuipers. OpenNetMon: network monitoring in openflow software-defined networks. In *Proc. IEEE/IFIP NOMS*, Krakow, Poland, May 2014.

[62] L. Jose, M. Yu, and J. Rexford. Online measurement of large traffic aggregates on commodity switches. In *Proc. ACM Hot-ICE*, Boston, MA, USA, 2011.

[63] M. Moshref, M. Yu, and R. Govindan. Resource/accuracy tradeoffs in software-defined measurement. In *Proc. ACM HotSDN*, Hong Kong, China, Aug. 2013.

[64] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli and E. Waisbard. Volumetric Hierarchical Heavy Hitters. In *Proc IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018.

[65] K. Phemius, M. Bouet. Monitoring latency with OpenFlow. In *Proc. International Conference on Network and Service Management (CNSM)*, Zurich, CH, Oct. 2013.

[66] K. Yu et al. Software-defined Latency Monitoring in Data Center Networks. In *Proc. PAM*, New York, NY, USA, Mar. 2015.

[67] Anurag Khandelwal et al. 2019. Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks. In *Proc. USENIX NSDI*, 2019.

[68] F. Fusco, L. Neri. High speed network traffic analysis with commodity multi-core systems. In *Proc. ACM IMC*, Melbourne, Australia, Nov. 2010, pp.218-24.

[69] L. Deri, M. Martinelli, T. Bujlow and A. Cardigliano. nDPI: Open-source high-speed deep packet inspection. In *Proc.* 2014 International Wireless Communications and Mobile Computing Conference (IWCMC), Nicosia, 2014, pp. 617-622.

[70] M. Trevisan, A. Finamore, M. Mellia, M. Munafo, D. Rossi. Traffic analysis with off-the-shelf hardware: challenges and lessons learned. In *IEEE Communications Magazine*, Vol 55, Mar 2017, pp. 163-169.

[71] Receive Side Scaling, Microsoft, Redmond, WA, USA, Feb. 15, 2015. Available: http://msdn.microsoft.com/en-us/library/windows/ hardware/ff567236(v=vs.85).aspx

[72] Yang Zhou, Omid Alipourfard, Minlan Yu, and Tong Yang. 2018. Accelerating network measurement in software. In *SIGCOMM Computer Communication Review*, 48, 3 (September 2018)

[73] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In Proceedings of the Symposium on SDN Research (SOSR '17)

[74] M. Ghasemi, T. Benson, J. Rexford. Dapper: data plane performance diagnosis of TCP. In *Proc. ACM SOSR*, Santa Clara, CA, USA, Apr. 2017, pp. 61-74.

[75] Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. 2018. A Comparison of Performance and Accuracy of Measurement Algorithms in Software. In Proceedings of the Symposium on SDN Research (SOSR '18)

[76] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. J. Algorithms (2004).

[77] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, high performance ethernet forwarding with CuckooSwitch. In Proceedings of the ninth ACM conference on Emerging networking experiments and technologies (CoNEXT '13)

[78] G Cormode et al. 2005. An Improved Data Stream Summary: The Count-min Sketch and Its Applications. J. Algorithms (2005).

[79] A. Metwally, D. Agrawal, A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. ICDT,* Edinburgh, UK, Jan. 2005.

[80] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: taking control of the enterprise. SIGCOMM Comput. Commun. Rev. 37, 4 (August 2007)

[81] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazires, and Nick McKeown. 2014. I know what your packet did last hop: using packet histories to troubleshoot networks. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)

[82] Q. Huang et al. Sketch Visor: Robust Network Measurement for Software Packet Processing. In *Proc. ACM SIGCOMM*, Los Angeles, CA, USA, Aug. 2017.

[83] S. Chowdhury, M. Bari, R. Ahmed, and R. Boutaba. PayLess: a low cost network monitoring framework for software defined networks. In *Proc. IEEE/IFIP NOMS*, Krakow, Poland, May 2014.

[84] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. CSAMP: a system for network-wide flow monitoring. In *Proc. USENIX NSDI*, 2008

[85] Xuan-Nam Nguyen, Damien Saucez, Chadi Barakat, and Thierry Turletti. Optimizing rules placement in OpenFlow networks: trading routing for better efficiency. In *Proc. ACM HotSDN*, 2014.

[86] M. Moshref, M. Yu, R. Govindan, A. Vahdat. SCREAM: sketch resource allocation for software-defined measurement. *Proc. ACM CoNEXT*, Heidelberg, Germany, Dec. 2015.

[87] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. 2016. MOZART: Temporal Coordination of Measurement. In *Proc. ACM SOSR*, Santa Clara, USA, Mar. 2016.

[88] A. Tootoonchian and Y. Ganjali. HyperFlow: a distributed control plane for OpenFlow. In *Proc. USENIX INM/WREN*, San Jose, CA, USA, pp. 3-9.

[89] P. Berde, et al. ONOS: towards an open, distributed SDN OS. In *Proc. ACM HotSDN*, Chicago, Illinois, USA, Aug. 2014, pp. 1-6.

[90] Shan-Hsiang Shen, Aditya Akella. DECOR: A distributed coordinated resource monitoring system. In *Proc. IEEE IWQoS*, Coimbra, Portugal, Jun. 2012.

[91] A. Gonzales, and R. Stadler. Adaptive distributed monitoring with accuracy objectives. In *Proc. ACM INM*, Pisa, Italy, Sep. 2006, pp. 65-70.

[92] D. Valocchi et al. Extensible signaling framework for decentralized network management applications. In *Proc. IEEE/IFIP NOMS*, Istanbul, Turkey, Apr. 2016.

[93] T. Jirsik, M. Cermak, D. Tovarnak, P. Celeda.Toward Stream-Based IP Flow Analysis. In *IEEE Communications Magazine*, Volume 55, Issue 7, pp 70-76.

[94] *OpenConfig* project. *http://www.openconfig.net*.

[95] V. Sekar, M. K Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proc. ACM IMC*, Melbourne, Australia, Nov. 2010, pp 328-341.

[96] F. Baccelli, S. Machiraju, D. Veitch, J. Bolot. The Role of PASTA in Network Measurement. In *ACM SIGCOMM*, Pisa, Italy, Sep. 2006.

[97] 100 Megabit Ethernet anonymized packet traces without payload: WIDE-TRANSIT link. *http://mawi.wide.ad.jp/mawi/ditl/ditl2007/.*

[98] *POX* OpenFlow controller. *http://www.noxrepo.org*.

[99] The GEANT Topology, 2004. *http://www.dante.net/ server/show/nav.007009007*

[100] The Germany50 Topology, 2004. *http://sndlib.zib.de*

[101] M. Claeys et al. Hybrid multi-tenant cache management for virtualized ISP networks. In *Journal of Network and Computer Applications*, vol. 68, issue C, pp. 28-41, June 2016.

[102] M. Manzano, J.A. Hernandez, M. Uruena, E. Calle. An empirical study of Cloud Gaming. In *Proc. Network and Systems Support for Games , Workshop on*, Venice, Italy, Nov. 2012.

[103] D. Tipper et al. An analysis of the congestion effects of link failures in wide area networks. In *IEEE Journal of Selected Areas in Communications*, vol. 12, pp. 179-191, Jan. 1994.

[104] A. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind Akamai. In *IEEE/ACM Transactions on Networking*, vol 17, no. 6, pp. 1752-1765, Dec. 2009.

[105] S. Choi, B. Wong, G. Simon, C. Rosemberg. A hybrid edge-cloud architecture for reducing on-demand gaming latency. In *Journal of Multimedia Systems*, Issue 5/2014.

[106] G. Tangari, D. Tuncer, M. Charalambides, and G. Pavlou. Decentralized Monitoring for Large-Scale Software-Defined Networks. In *Proc. IEEE/IFIP IM*, Lisbon, Portugal, May 2017.

[107] I. Poese et al. Enabling content-aware traffic engineering. In *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 21-28, Oct. 2012.

[108] Gaikai video game streaming platform. *https://www.playstation.com/en-gb/explore/playstation-now/?smcid=psnow-vanityurl*.

[109] M. Wichtlhuber, R. Reinecke, and D. Hausheer. An SDN-based CDN/ISP collaboration architecture for managing high-volume flows. In *IEEE TNSM*, vol. 12, no. 1, pp. 48-60, Mar. 2015.

[110] D. Tuncer, M. Charalambides, G. Pavlou, N. Wang. DACoRM: a coordinated, decentralized and adaptive network resource management scheme. In *Proc. IEEE/IFIP NOMS*, Westin Maui Maui, HI, USA, Apr. 2012, pp. 417-425.

[111] Mininet: An Instant Virtual Network on your Laptop. http://mininet.org/

[112] Nikhil Handigol, Brandon Heller, Vimal Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments using Container-Based Emulation. In *Proc. ACM CoNEXT*, Dec. 2012.

[113] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proc ACM HotSDN*, Oct. 2010.

[114] M Jamshed et al. Kargus: a highly-scalable software-based intrusion detection system. In *Proc. ACM CCS*, Raleigh, NC, USA, Oct. 2012, pp. 317-328.

[115] M. Dobrescu, K. Argyraki, S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Proc. USENIX NSDI*, San Jose, CA, USA, Apr. 2012, pp 11-24.

[116] X. Yu et al. CountMax: A lightweight and cooperative sketch measurement for Software-Defined Networks. In *IEEE/ACM Transactions on Networking*, vol. 26, no. 6, pp. 2774-2786, Dec. 2018.

[117] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug. 2018.

[118] J. Moraney and D. Raz. On the Practical Detection of the Top-k Flows. In *Proc. International Conference on Network and Service Management (CNSM)*, Rome, Italy, 2018, pp. 81-89

[119] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *SIGCOMM Computer Communication Review*, 35, 4, Aug. 2005, pp. 181-192.

[120] G.Cormode and M. Hadjeleftheriou. Finding Frequent Items in Data Streams. In *Proc. PVLDB*, Aug. 2008.

[121] B. Atikoglu et al. Workload analysis of a large-scale key-value store. In *Proc. ACM Sigmetrics*, London, UK, Jun. 2012, pp. 53-64.

[122] J. Boite et al. Statesec: Stateful monitoring for DDoS protection in software defined networks. In *Proc. IEEE NetSoft*, Bologna, IT, 2017, pp. 1-9.

[123] G. Bianchi et al. Open Packet Processor: a programmable architecture for wire speed platform-indipendent stateful in-network processing. Available: https://arxiv.org/pdf/1605.01977.pdf, 2016

[124] F. Guo, Y. Solihin. An Analytical Model for Cache Replacement Policy Performance. In *Proc. ACM Sigmetrics*, Saint Malo, France, June 2006, pp. 228-239.

[125] D. Tam, R. Azimi, L. Soares, M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. APLOPS*, Washington DC, USA, Mar 2009, pp. 121-132.

[126] R. West, P. Zaroo, C. Waldspurger, X. Zhang. Online cache modeling for commodity multicore processors. In *Proc. ACM SIGOPS Operating System Review*, vol 44, Dec. 2010, pp.19-29.

[127] L. Zhao et al. Cachescouts: Fine-grain monitoring of shared caches in cmp platforms. In Proceedings of the conference on Parallel architectures and compilation techniques (PACT), 2007.

[128] D.Knuth. The Art of Computer Programming. Vol 2, page 232, 3rd edition.

[129] G. Tangari, M. Charalambides, D. Tuncer, and G. Pavlou. Adaptive Traffic Monitoring for Software Dataplanes. In *Proc. IEEE CNSM*, Nov 2017.

[130] G.Hasslinger, O.Hohlfeld. The Gilbert-Elliott Model for Packet Loss in Real Time Services on the Internet. In *Proc.* GI/ITG Measurement, Modelling and Evalutation of Computer and Communication Systems

[131] M. Alizadeh et al. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[132] K. Xie et al. 2018. On-Line Anomaly Detection With High Accuracy. *IEEE/ACM Transactions on Networking*, 26, 3 (2018).

[133] Behnaz Arzani et al. 2016. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *Proc. ACM SIGCOMM*, 2016.

[134] Lihua Yuan et al. 2011. ProgME: Towards Programmable Network Measurement. In *IEEE/ACM Transactions on Networking*, 19, 1, Feb. 2011.

[135] 2016. Trumpet. https://github.com/USC-NSL/Trumpet

[136] Surajit Chaudhuri et al. 2017. Approximate Query Processing: No Silver Bullet. In *Proc. ACM SIGMOD*, 2017.

[137] 2018. The CAIDA UCSD Anonymized Internet Traces Dataset - March 2018. http://www.caida.org/data/passive/passive_dataset.xml.

[138] Michael Mozer et al. 2001. Prodding the ROC Curve: Constrained Optimization of Classifier Performance. In *Proc. NIPS*, 2001.

[139] Corinna Cortes and Mehryar Mohri. 2003. AUC Optimization vs.Error Rate Minimization. In *Proc. NIPS*, 2003.

[140] Youden, W.J. (1950). Index for rating diagnostic tests.

[141] G. Bianchi, M. Bonola, G. Picierro, S. Pontarelli and M. Monaci. StreaMon: A software-defined monitoring platform. In *Proc.* 26th International Teletraffic Congress (ITC), Karlskrona, 2014.

[142] Y. Yu, C. Quien, X. Li. Distributed collaborative monitoring in software defined networks. In *Proc. ACM HotSDN*, Chicago, IL, USA, Aug. 2014, pp. 85-90.

[143] In-Band Network Telemetry. https://p4.org/p4/inband-network-telemetry/

[144] Srinivas Narayana et al. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. ACM SIGCOMM*, 2017.