MARCOS VINÍCIOS DA SILVA MACHADO

ADAPTIVE POLLING INTERVALS FOR NETWORK MONITORING
Marcos Vinícius da Silva Machado

Adaptive Polling Intervals for Network Monitoring

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre Profissional em Ciência da Computação.


RECIFE
2017
M149a Machado, Marcos Vinícius da Silva

99 f.: il., fig., tab.

Orientadora: Judith Kelner.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2017.
Inclui referências.


004.6 CDD (23. ed.) UFPE- MEI 2018-003
Marcos Vinícius da Silva Machado

Adaptive Polling Intervals for Network Monitoring

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 11/08/2017

BANCA EXAMINADORA

______________________________
Profa. Dra. Judith Kelner
Centro de Informática / UFPE
(Orientadora)

______________________________
Prof. Dr. Paulo da Fonseca Pinto
Departamento de Engenharia Eletrotécnica / Universidade Nova Lisboa

______________________________
Prof. Dr. Rafael Roque Aschoff
Instituto Federal de Pernambuco / Campus Palmares
ACKNOWLEDGEMENTS

As always, firstly, I want to thank my family, my father Fernando Machado, my mother Edilene Machado, my brothers Luiz Fernando and Claudio Roberto, thank you guys for the support, this is as yours as mine. A big thank you to my mother/aunt Edjane Lopes, I appreciate all the things done to help me in this journey. Many thanks to my cousin Tarcísio Souza for the support, smart remarks and friendship, it is GG bro! Thanks to Lucas Souza, Arnaldo Neto, Paulo André, Igor Oliveira, Flavio Lopes. Brotherhood, friendship, and support!

I would like to thanks my advisor Prof. Judith Kelner for the patience, support, and helping me out to finish this work. Many thanks to my fellow research friends and coworkers for helping me out taking some of the burdens so I could concentrate on finishing this work.

And a heartfelt thank you for all the people that helped me to accomplish this and I did not remember to name here, you all are in my heart.
Never fear quarrels, but seek hazardous adventures.

—ALEXANDRE DUMAS, THE THREE MUSKETEERS
ABSTRACT

Monitoring is an essential task for the management of any network and several network functions dependent on it. In Software Defined Networking (SDN) is no different, even with all the benefits provided by it. Network monitoring helps to understand patterns in network traffic, allowing to capture its state and enhance its configuration. Furthermore, it aids the infrastructure management, the discovery of bottlenecks, the location of problems related to software and hardware, and enforces SLAs (Service Level Agreement). Cloud services are spread around the globe and are used constantly by several companies. However, ensure the safety and quality of these services is a difficult and non-stop chore. Network monitoring has a major part in assuring that the services are executed without problem. Ideally, it is expected that the monitoring solution does not overload the network, scales together with the network causing minimal impact, has a controlled resource usage and works for every network device independent of the vendor. The vendor lock-in problem was solved by the OpenFlow protocol, the main and default SDN protocol. However, the others aforementioned problems are dependent on the monitoring solution implementation and used strategies. Furthermore, to the best of our knowledge, no SDN monitoring solution solved these problems ensuring all the needed network monitoring capabilities. In order to address this gap, we proposed SHAMon, a network monitoring solution for Software Defined Networking, created to provide fine-grained and precise information, generating minimal network and resources overhead. We propose the use of the binomial algorithms used for congestion control by the TCP protocol to control our statistics request mechanism. We use all the OpenFlow features to aid in getting timely and refined statistics. Additionally, we use a proxy-server architecture to allow our solution to scale with the network infrastructure. To validate the use of binomial algorithms, we defined a set of experiments. The first set of experiments was intended on analyze two functions of our solution, the former being how variations of parameters of the congestion control algorithms are related to how much the polling time vary. The latter was how variations of the thresholds that limit the execution of the congestion control algorithms are related to the decision to vary the polling time. In addition, we did a comparison experiment with three other solutions, Payless, periodic polling, and OpenNetMon in the same network scenario. The major objective of this experiment was to highlight some of our qualities. The results from both experiments were good, the former showed us the behavior of the binomial algorithms and our solution. Our solution had a low error in network measurement, varying from 26% to 15%. The latter showed that our monitoring overhead was three times lower than the others, validating that our solution is both accurate and not network consuming.

Keywords: Software Defined Networking. Networking Monitoring. Network Management
O monitoramento de rede é uma tarefa essencial para o gerenciamento de qualquer rede. Várias funções de rede são dependentes dele. Em Software Defined Networking (SDN) não é diferente, mesmo com todos os benefícios fornecidos pelas redes SDN. O monitoramento de rede ajuda a entender padrões no tráfego, permitindo capturar seu estado e aprimorar sua configuração. Além disso, auxilia no gerenciamento da infra-estrutura, a descoberta de estrangulamentos e garante a aplicação de SLAs (Acordo de Nível de Serviço). Os serviços oferecidos na nuvem estão espalhados pelo mundo e são constantemente utilizados por várias empresas, no entanto, garantir a segurança e qualidade desses serviços é uma tarefa difícil e sem fim. O monitoramento de rede tem uma parte importante em garantir que esses serviços sejam executados sem problemas. Idealmente, espera-se que a solução de monitoramento não sobrecarregue a rede, escale causando impacto mínimo, tenha um uso de recursos controlado e funcione para cada dispositivo de rede independente do fornecedor. O problema de incompatibilidade de dispositivos entre fornecedores foi resolvido pelo protocolo OpenFlow, o principal protocolo SDN. No entanto, os outros problemas mencionados acima dependem da implementação da solução de monitoramento e das estratégias utilizadas. Além disso, nós desconhecemos uma solução de monitoramento para SDN a qual tenha resolvido todos os problemas citados e garanta todas as funções de monitoramento necessárias. Para preencher essa lacuna, propomos SHAMon, uma solução de monitoramento para Redes Definidas por Software, criada para fornecer informações precisas e refinadas, com sobrecarga mínima de rede e recursos. Propomos a utilização dos algoritmos binomiais usados no controle de congestionamento pelo protocolo TCP para controlar nosso mecanismo de solicitação de estatísticas. Utilizamos os recursos do protocolo OpenFlow para auxiliar na obtenção de estatísticas oportunas e refinadas. Além disso, usamos uma arquitetura proxy-servidor para permitir que nossa solução conseguisse escalar com a infraestrutura de rede. Para validar o uso dos algoritmos binomiais, definimos um conjunto de experimentos. O objetivo do primeiro experimento foi analisar dois procedimentos da nossa solução: como as variações de parâmetros dos algoritmos de controle de congestionamento se relacionam com quanto a variação do polling time das estatísticas, e como as variações dos limitantes de execução dos algoritmos de controle de congestionamento se relacionam à decisão de variar o tempo. Nós também compararam nossa solução com outras três soluções, Payless, polling periódico e OpenNetMon no mesmo cenário de rede. Os resultados de ambos os experimentos foram bons, o primeiro nos mostrou o comportamento dos algoritmos binomiais e da nossa solução. Nossa solução errou pouco na medição da utilização da rede, variando de 26% a 15%. No último experimento, vimos que a nossa sobrecarga de rede foi três vezes menor do que os outros, validando que nossa solução é precisa e sobrecarrega pouco a rede.

Palavras-chave: Redes Definidas por Software. Monitoramento de Rede. Gerenciamento de Rede
LIST OF FIGURES

2.1 Five logical measurement functions of monitoring operations, based on five logical layers from (LEE, LEVANTI; KIM, 2014) ........................................... 19
2.2 Software Defined Networking (SDN) architecture, based on SDN three layers image from (OPEN..., 2014) ................................................................. 22
2.3 Floodlight architecture, based on the (FLOODLIGHT, 2016) infograph. ........ 23
3.1 Explanation of the related work classification .............................................. 42
3.2 Related Works Timeline ............................................................................ 42
4.1 SHAMon in the SDN architecture .............................................................. 45
4.2 SHAMon setup without proxies ................................................................. 48
4.3 SHAMon setup with proxies ...................................................................... 49
4.4 SHAMon Controller modules ..................................................................... 50
4.5 SHAMon Proxy architecture ...................................................................... 50
4.6 Three step installation procedure of new flow rule. ................................. 52
4.7 OpenFlow procedures to get statistics. ...................................................... 53
4.8 Scheduling module threshold. ................................................................. 54
4.9 Description of the scheduling algorithm. .................................................. 56
4.10 Controller flow for new devices connecting. .......................................... 57
4.11 Controller flow for new Scalable High Accurate Monitoring (SHAMon) Proxy connecting. ............................................................. 58
4.12 Controller flow for SHAMon Proxy updates. .......................................... 59
5.1 Experiment network topology ................................................................. 69
5.2 Network physical configuration diagram ................................................. 70
5.3 Timeline diagram of the experiment ....................................................... 73
5.4 RMSE and network overhead for k and L variations ................................. 76
5.5 Network Utilization Measurement for k = -0.5 and L = 0.5 .................... 77
5.6 Network Utilization Measurement for k = -0.5 and L = 0.75 ................. 77
5.7 Network Utilization Measurement for k = -0.5 and L = 1 ...................... 78
5.8 Network Utilization Measurement for k = -0.75 and L = 0.5 ................. 78
5.9 Network Utilization Measurement for k = -0.75 and L = 0.75 ............... 79
5.10 Network Utilization Measurement for k = -0.75 and L = 1 ................. 79
5.11 Network Utilization Measurement for k = -1 and L = 0.5 ...................... 80
5.12 Network Utilization Measurement for k = -1 and L = 0.75 ................. 80
5.13 Network Utilization Measurement for k = -1 and L = 1 ...................... 81
5.14 Boxplot of polling time variation for k and L variations ......................... 83
5.15 Error and overhead measurement for the threshold experiment . . . . . . . . . . . . 85
5.16 Boxplot of time frequency variation for the threshold variations . . . . . . . . . . . . 87
5.17 Comparison of the solutions network utilization . . . . . . . . . . . . . . . . . . . . 90
5.18 Payless network utilization measurement . . . . . . . . . . . . . . . . . . . . . . 91
LIST OF TABLES

2.1 Layer activities summarization ............................................. 21
3.1 Network Monitoring Solutions Classification .......................... 32
3.2 Related works and SHAMon network monitoring functions ......... 43
4.1 SHAMon Classification according to the five network logical functions .... 46
5.1 Switches’ IP Addresses .................................................. 69
5.2 Distribution of switches, interfaces, and ports ...................... 71
5.3 Parameters summarization .............................................. 73
5.4 Parameters variation for the Scheduler algorithm experiment .......... 75
5.5 Polling time information for $k$ and $L$ variation .................. 82
5.6 Polling time information for threshold variation .................... 88
5.7 Parameters of the solutions for experiment .......................... 89
5.8 Network overhead and measurement error from the solutions .......... 90
<table>
<thead>
<tr>
<th>ACRONYM</th>
<th>EXPLANATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>DDoS</td>
<td>Distributed Denial of Service</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>OF</td>
<td>OpenFlow</td>
</tr>
<tr>
<td>SDN</td>
<td>Software Defined Networking</td>
</tr>
<tr>
<td>SHAMon</td>
<td>Scalable High Accurate Monitoring</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicators</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AIMD</td>
<td>additive increase/multiplicative decrease</td>
</tr>
<tr>
<td>MIMD</td>
<td>multiplicative increase/multiplicative decrease</td>
</tr>
<tr>
<td>MIAD</td>
<td>multiplicative increase/additive decrease</td>
</tr>
<tr>
<td>AIAD</td>
<td>additive increase/additive decrease</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>central processing unit</td>
</tr>
<tr>
<td>BLL</td>
<td>Business Logic Layer</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>RMSE</td>
<td>root-mean-square error</td>
</tr>
</tbody>
</table>
# CONTENTS

1 INTRODUCTION .................................................. 14
  1.1 Problem Statement ........................................... 16
  1.2 Research Question ........................................... 17
  1.3 Goals ..................................................... 17
    1.3.1 General Goal ......................................... 17
    1.3.2 Specific Goals ....................................... 17
  1.4 Organization of the Thesis .................................. 17

2 BACKGROUND .................................................. 19
  2.1 Network Monitoring ........................................ 19
    2.1.1 Collection ............................................ 20
    2.1.2 Representation ........................................ 20
    2.1.3 Report ............................................... 20
    2.1.4 Analysis .............................................. 20
    2.1.5 Presentation .......................................... 20
    2.1.6 Throughput ........................................... 21
    2.1.7 Overhead ............................................. 21
    2.1.8 Congestion ........................................... 21
    2.1.9 Network polling ...................................... 21
  2.2 Software Defined Networking ............................... 22
    2.2.1 Floodlight Controller .................................. 23
    2.2.2 OpenFlow Protocol .................................... 24
      2.2.2.1 Controller-to-switch messages ..................... 24
      2.2.2.2 Asynchronous messages ........................... 25
      2.2.2.3 Symmetric messages ............................... 26
  2.3 Binomial Algorithms ....................................... 26
  2.4 Root Mean Square Error (RMSE) ............................ 27

3 MONITORING IN SOFTWARE DEFINED NETWORKING ............... 28
  3.1 Literature Review Methodology ............................. 31
  3.2 Related Works ............................................ 31
    3.2.1 OpenSAFE ........................................... 32
    3.2.2 PayLess ............................................. 33
    3.2.3 OFMon ................................................ 33
    3.2.4 Floware ............................................. 34
    3.2.5 FleXam .............................................. 35
    3.2.6 FlowCover ........................................... 35
INTRODUCTION

Network monitoring is an essential task in network management, a required source of information for traffic engineering and network security. It helps to understand tendencies and patterns in network traffic, making possible to capture the state of a network and allows its configuration and improvement. Besides network improvements, it aids the infrastructure management, the discovery of bottlenecks, find software and hardware problems, and enforce the Service Level Agreement (SLA).

The increase of Internet-based, and highly scalable distributed computing systems in which computational resources are offered 'as a service' by cloud computing motivated industry and academia to adopt cloud computing and use it in a wide spectrum of applications ranging from lightweight services to high computationally intensive applications. The development was possible because of the evolution of the network environment, its technologies, protocols, and infrastructure. This paradigm shift created new necessities to guarantee the efficiency as well as productivity of the new services deployed.

Vendor lock-in, multi-tenancy and isolation, data management, service portability, elasticity engines, SLA management, and cloud security are some of the necessities cloud computing has. They can impact cloud computing services directly, and some of them are open issues still.

Besides the necessities related to the cloud computing environment, the boom of mobile and Internet of Things (IoT) devices changed services development, network infrastructure, and network management’s concerns. New issues emerged, and old ones came back requiring new approaches that fit this context.

For instance, a largest Distributed Denial of Service (DDoS) attack on OVH, an Internet Service Provider, was reported; it hit OVH with two simultaneous attacks whose combined bandwidth reached almost 1 Tbps (OVH, 2016). Furthermore, the Brian Krebs’ Blog was hit by 665 Gbps DDoS attack (KREBS, 2016) designed to knock the site offline.

Both attacks did not succeed, the former was monitored and mitigated by the OVH infrastructure itself. The latter had its contents delivered by Akamai, and the monitoring and defensive mechanism provided were sufficient to mitigate the attack. Both attacks were done by a botnet network made of IoT devices, which were easily kidnapped because of poor usage of security in communication and poor password choices.
These examples show that new threats are just around the corner in brand-new scales, and while they were addressed, new strategies are required if we do not want to suffer any casualty. For instance, traditional networks have protocols and mechanisms that are used to monitor itself and to lessen problems and threats. However, most of them need external hardware that is vendor related like NetFlow (CLAISE, 2004), one of the most used passive monitoring tool, or does not provide the necessary account information like Simple Network Management Protocol (SNMP).

As an effort to mitigate the issues related to network management, vendor lock-in, and deployment of network solutions, the paradigm of Software Defined Networking (SDN) was proposed (MCKEOWN et al., 2008) and it was induced by the community (OPEN…, 2014). SDN has widened the range of network solutions based on the following three main points:

(1) Network control plane is separated from the data plane;
(2) The control decisions are centralized in a controller;
(3) The forward logic functions are withdrawn from the hardware and aggregated to an application layer.

The OpenFlow (OF) protocol (PFAFF et al., 2012) is widely accepted as the default South-Bound API interface between switches and controllers in the SDN paradigm. It provides a global view of the network and per flow statistics collection at the controller, which allows complex and efficient monitoring solutions to be developed.

Accurate and timely monitoring statistics are valuable information for tasks like load balance, traffic engineering, accounting, intrusion detection and, networks attacks in general. Network monitoring has been an active field of research in the last decade because of the difficulty in measuring IP networks accurately, due to the large volume of traffic flows and the complexity of deploying a measurement infrastructure (KIM et al., 2016; CHOWDHURY et al., 2014; ADRICHEM; DOERR; KUIPERS, 2014; SUH et al., 2014; SHIRALI-SHAHREZA; GANJALI, 2013; YU CRISTIAN LUMEZANU, 2013).

However, even with all the works and studies presented, network monitoring still has the same concerns: the network overhead generated by the monitored information, scale together with the physical infrastructure without impact and minimal resource usage. Meanwhile, it is expected that it generates fine-grained and accurate information.

To address these issues we present Scalable High Accurate Monitoring (SHAMon) an SDN solution created to provide fine-grained and accurate network information, meanwhile generating minimum network overhead. We propose the usage of network congestion control algorithms to control the monitoring process of querying statistics. Additionally, we propose the usage of proxy instances to work between switches and controllers, grouping the statistics messages to minimize the overhead in large networks.
1.1 Problem Statement

As already mentioned, network monitoring is a fundamental task in network management; accurate and timely statistics are valuable information to control, improve, and protect the services managed. Several solutions of network monitoring exist in traditional networks but employing them are not easy because of:

- Acquisition of extra hardware to the network;
- Vendor lock-in solutions;
- Hard to configure;
- Work with other solutions.

Clouds environments and cloud solutions are being adopted more and more by enterprises, and monitoring is a task of paramount importance for both Providers and Consumers. For instance, private cloud adoption increased from 63 percent to 77 percent, driving hybrid cloud adoption up from 58 percent to 71 percent year-over-year (RIGHTSCALE, 2016). In addition, users running applications in an average of 1.5 public clouds and 1.7 private clouds (RIGHTSCALE, 2016).

Several key instrumental activities fundamental to a cloud are dependent on a fully functional monitoring system. The main ones are: (1) To control and manage hardware and software infrastructures, (2) to provide information and Key Performance Indicatorss (KPIs) for both platforms and applications, (3) the continuous monitoring of the Cloud and of its SLAs, (4) security management, and (5) billing.

SDN and OF protocol came to solve problems from traditional networks and to facilitate the network management as an open standard which is manufacturer independent, programmable, and easy to configure. The OF protocol was designed with monitoring capabilities, which allows SDN applications to monitor the network and its devices.

While the OF protocol allows the network and its devices to be monitored, the management of the monitoring process still must be created. Therefore, merely use OF protocol is not enough to create a network monitoring application that does not have one or more of the following problems:

- Resources overuse;
- Network overhead;
- Accurate, timely, and fine grained information;
- Scalable with the network growth.

Miscellaneous network monitoring projects and solutions in SDN have been developed as (CALERO; AGUADO, 2015; BALLARD; RAE; AKELLA, 2010; CHOWDHURY et al., 2014; JARSCHEL; ZINNER; TRAN-GIA, 2013; KIM et al., 2016; NACSHON; PUZIS; ZILBERMAN, 2016; SHIRALISHAHREZA; GANJALI, 2013; SU et al., 2014; SUH et al., 2014; TOOTOONCHIAN; GHOBADI; GANJALI,
2010; ADRICHEM; DOERR; KUIPERS, 2014; YU CRISTIAN LUMEZANU, 2013), but to the best of our knowledge none could address all of the aforementioned problems.

1.2 Research Question

Based on this context, the main research question investigated by this thesis is: if it is possible to create a new approach for network monitoring in SDN with minimal network overhead and resource usage, monitoring timely and accurately network flows obtaining fine grained information.

1.3 Goals

1.3.1 General Goal

The general goal of this work is to create a network monitoring solution that is accurate, timely and fine-grained that generates minimal overhead on the network and scale with the infrastructure growth.

1.3.2 Specific Goals

As our specific goals, we want:

- to validate the use of binomial congestion control algorithms to minimize the network overhead;
- to create an Application Programming Interface (API) that allows the monitored information to be used by any SDN application, in different levels of aggregation, and with different filters;
- to define an architecture for SDN monitoring solutions that allows it to scale with the infrastructure growth.

1.4 Organization of the Thesis

The remainder of this work is structured as follows:

- Chapter 2 reviews essential concepts used throughout this work. Firstly, we present the monitoring process and its logical divisions. Then, we do a small debrief about SDN, the Floodlight Controller, and move to OpenFlow protocol, explaining the structure of OF messages used. We finalize it by explaining the concepts of Binomial Algorithms, where they are used, and their properties;
■ **Chapter 3** presents the related works of SDN monitoring solutions, presenting them in a timeline and classifying them;

■ **Chapter 4** presents our solution overview, its architecture and functions dissecting each module with its functionalities;

■ **Chapter 5** contains the description of our evaluations, its objectives, the methodology used, and the results with a detailed explanation;

■ **Chapter 6** concludes our dissertation, presenting our achievements, contributions, and future works.
In this chapter, fundamental concepts explored by this work will be presented to enlighten the understanding of our approach and the choices we made.

### 2.1 Network Monitoring

The network monitoring process relates to the activity of supervising the network with the intent of early identification of threats, the discovery of patterns and trends and grasp the network state. With the results of monitoring operations, it is possible to build a model of the network’s current behavior, and this network model represents the operational status of the network (Lee; Levanti; Kim, 2014). Furthermore, with the network model is possible to fix bottlenecks, find problems in devices, keep track of the network usage, and plan investments to scale the network. As one may note, monitoring has a high impact on the maintenance, design of resource allocation, and network growth planning.

![Figure 2.1](image)

**Figure 2.1** Five logical measurement functions of monitoring operations, based on five logical layers from (Lee; Levanti; Kim, 2014)

The monitoring operations can be divided into five logical measurement functions: (1) collection, (2) representation, (3) report, (4) analysis and (5) presentation as in figure 2.1. This logical division reflects each step of the complete monitoring workflow in a sequential order, grouping by macro activities of the process, clarifying their functionality and interactions.
2.1. Network Monitoring

2.1.1 Collection

This layer collects the network raw measurement data. The collector entity can measure data in an active or passive manner, and the sampling methods can vary too. Active monitoring relates to the injection of test traffic and it directly evaluates what is wanted to be observed without any waiting. Passive monitoring depends on the network devices to gather the events and information, and it is either executed by network devices like routers and switches or is done by dedicated ones. The active monitoring is intrusive while the passive is not. Sampling is one way of reducing the overhead on monitoring nodes and performing efficient monitoring.

2.1.2 Representation

This layer is responsible for processing the measured data and put it in a particular format. The representation of measurements needs to be standardized so each analysis function does not convert the collected measurements into distinct intermediate forms of representation and the different analysis functions can use the same collected measurements. Additionally, the measurements from heterogeneous and distributed collection devices need to be synchronized in time.

2.1.3 Report

This layer sends the measured data from the network devices to the management stations where higher-layer functions are typically placed. The reporting of measurements needs to be efficient regarding the bandwidth consumed for transferring the measurement data from the collection devices to the management stations.

2.1.4 Analysis

This layer is where the interpretation of data and the extraction of high-level information happen. Although varieties of analysis functions exist, the six most common analysis functions are general-purpose traffic analysis, estimation of traffic demands, traffic classification per application, mining of communication patterns, fault management, and automatic updating of network documentation.

2.1.5 Presentation

This layer presents measurement data to network operators in different formats, such as visual and textual representations.
Following the logical division presented above, each layer activities is summarized in table 2.1.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Presentation</td>
</tr>
<tr>
<td></td>
<td>Visualization of large measurement data</td>
</tr>
<tr>
<td></td>
<td>Analysis</td>
</tr>
<tr>
<td></td>
<td>Estimation of traffic demands</td>
</tr>
<tr>
<td></td>
<td>Traffic classification</td>
</tr>
<tr>
<td></td>
<td>Mining communication patterns</td>
</tr>
<tr>
<td></td>
<td>Fault management</td>
</tr>
<tr>
<td></td>
<td>Report</td>
</tr>
<tr>
<td></td>
<td>Periodic polling vs. event-triggered polling</td>
</tr>
<tr>
<td></td>
<td>Representation</td>
</tr>
<tr>
<td></td>
<td>Standardization of measurement format</td>
</tr>
<tr>
<td></td>
<td>Synchronization of measurements</td>
</tr>
<tr>
<td></td>
<td>Collection</td>
</tr>
<tr>
<td></td>
<td>Active and passive monitoring sampling</td>
</tr>
</tbody>
</table>

The network monitoring activity has some functions and concepts that are needed to understand the related problems and difficulties. They are defined as follows:

2.1.6 Throughput

Throughput, in general, can be defined as the maximum rate at which something can be processed. In network communication, throughput can be defined as the rate of successful message delivery over a communication channel.

2.1.7 Overhead

Overhead, by definition, is the resources required to set up a specific activity. So, network overhead is the network resources needed, usually bandwidth, to execute some network function.

2.1.8 Congestion

Network congestion is the drop in the network quality service because one or more nodes are carrying more data than it can handle. It generates several problems in a network as packet loss, delay, and new connections block.

2.1.9 Network polling

Network polling refers to actively sampling the status of a network device. It is one of the most used techniques to monitor the network and devices behavior.
2.2 Software Defined Networking

SDN is the physical separation of the network control plane from the forwarding plane, and where a control plane manages several devices. It is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today’s applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. The OF protocol is a foundational element for building SDN solutions (OPEN…, 2014).

![Software Defined Networking (SDN) architecture](image)

- Directly programmable: Network control is directly programmable because it is decoupled from forwarding functions;
- Agile: Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs;
- Centrally managed: Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch;
- Programmatically configured: SDN lets network managers configure, manage, secure, and optimize network resources very quickly via dynamic, automated SDN programs,
which they can write themselves because the programs do not depend on proprietary software;

- Open standards-based and vendor-neutral: When implemented through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

The SDN architecture is divided in three layers: application, control, and infrastructure. This division was made to decoupling the intelligence from the infrastructure and concentrate it in the control layer as presented in figure 2.2. To connect the infrastructure layer with the control layer the OF protocol is used as the standard. Between the control layer and the application layer there are no standards, but Representational State Transfer (REST) is the most used.

The infrastructure layer is formed by the switches, and devices that implement the OF protocol, the SDN controllers made the control layer, and the application layer contains the applications developed to work with the controller to improve the network.

### 2.2.1 Floodlight Controller

![Floodlight Architecture](FLOODLIGHT.png)

**Figure 2.3** Floodlight architecture, based on the (FLOODLIGHT, 2016) infograph.

Project Floodlight is the world’s leading open source Software Defined Networking (SDN) Controller. The project is committed to open source, open standards, and open API (FLOODLIGHT, 2016). The Floodlight project offers a fair quantity of features like:

- Module loading system that make it simple to extend and enhance;
2.2. SOFTWARE DEFINED NETWORKING

- Easy to set up with minimal dependencies;
- Supports a broad range of virtual- and physical- OpenFlow switches;
- Can handle mixed OpenFlow and non-OpenFlow networks – it can manage multiple “islands” of OpenFlow hardware switches;
- Designed to be high-performance – is multithreaded from the ground up.

The Figure 2.3 shows the Floodlight architecture, in which the applications developed to work with the Floodlight controller use the northbound API to communicate. The Floodlight controller uses the Indigo data plane interface, which is an OpenFlow agent, to manage the OpenFlow state, devices, and configurations, as well as handle the network sockets. It is important to observe that the Floodlight Controller is composed by the controller entity and the Indigo interface, which mainly act as the SDN’s southbound interface.

2.2.2 OpenFlow Protocol

The OpenFlow protocol was proposed by (MCKEOWN et al., 2008) as a way for researchers to run experimental protocols in the networks they use every day. The OF architecture consists of three basics concepts:

1. The network is built up with OpenFlow-compliant switches;
2. The control plane consists of one or more OF controllers;
3. A secure control channel connects the switches with the control plane.

An OF switch is a device that forwards packets according to its flow table. This flow table has sets of flow entries, each of which has match fields, counters, and instructions. The controller is the entity responsible for populating and manipulating the table’s entries according to its policies and configurations, and the OF protocol is the interface responsible for allowing the communication between them. The OF protocol supports three message types - controller-to-switch, asynchronous, and symmetric - each with multiple sub-types.

Today, the OpenFlow protocol is the standard southbound protocol used in SDN to manage its devices. The latest release is the 1.5.1 version (NYGREN et al., 2015). However, the newest implemented version is the 1.3.x (PFAFF et al., 2012), which is the reason we used this version in our experiments.

2.2.2.1 Controller-to-switch messages

Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch, and they may or may not require a response from the switch. The message types are Features, Configuration, Modify-State, Read-State, Packet-out, Barrier, Role-Request, and Asynchronous-Configuration.

The OF mechanism to collect the network state is done using Controller-to-switch messages of the type Read-State. An OFPMP_FLOW_REQUEST request is sent to the switches,
which is answered with one or more `OFPMP_FLOW_REPLY`. The Controller can ask for **individual flow**, **aggregate flow**, **group**, **table**, **port**, **queue**, **group counter**, and **meter** state information. The request can be made by the controller itself, or by any application on it. The `OFPT_FLOW_REMOVED` message that is sent by a switch when a flow is removed from the flow table carries the latest state information of the flow. Bellow is the body of the reply to an `OFPMP_FLOW_REQUEST`.

```c
/* Body of reply to OFPMP_Flow request. */
struct ofp_flow_stats {
  uint16_t length; /* Length of this entry. */
  uint8_t table_id; /* ID of table flow came from. */
  uint8_t pad;
  uint32_t duration_sec; /* Time flow has been alive in seconds. */
  uint32_t duration_nsec; /* Time flow has been alive in nanoseconds beyond duration_sec. */
  uint16_t priority; /* Priority of the entry. */
  uint16_t idle_timeout; /* Number of seconds idle before expiration. */
  uint16_t hard_timeout; /* Number of seconds before expiration. */
  uint16_t flags; /* One of OFPFF_. */
  uint8_t pad2[4]; /* Align to 64−bits. */
  uint64_t cookie; /* Opaque controller−issued identifier. */
  uint64_t packet_count; /* Number of packets in flow. */
  uint64_t byte_count; /* Number of bytes in flow. */
  struct ofp_match match; /* Description of fields. Variable size. */
}; OFP_ASSERT(sizeof(struct ofp_flow_stats) == 56);
```

Individual flow state request is the most fine-grained information available to be requested from a switch. It carries the **table id**, **duration time**, **priority**, **time to expire**, **cookie**, **bytes** and **packets**. The time duration comes in seconds and nanoseconds. The first is the time the flow has been alive, the second is the time the flow has been alive beyond the second duration. There are two times to expire, idle timeout and hard timeout: idle timeout is the maximum time a flow rule can stay idle before being removed; and hard timeout is the maximum time a flow can stay in a switch.

2.2.2.2 **Asynchronous messages**

Asynchronous messages are initiated by the switch and used to update the controller of network events and changes to the switch state. Asynchronous messages are sent without a controller soliciting them from a switch. Switches send asynchronous messages to controllers to denote a packet arrival, switch state change, or error. The four main asynchronous message types are **Packet-in**, **Flow-Removed**, **Port-status**, and **Error**.
2.3. BINOMIAL ALGORITHMS

2.2.2.3 Symmetric messages

Symmetric messages are initiated by either the switch or the controller and sent without solicitation, in either direction. The Symmetric messages types are Hello, Echo, and Experimenter.

2.3 Binomial Algorithms

In network management, binomial algorithms are a class of nonlinear congestion control algorithms, introduced by (BANSAL; BALAKRISHNAN, 2001). The stability of the Internet to date has in large part been due to the congestion control and avoidance algorithms. They are used as rules to increase/decrease the length of the Transmission Control Protocol (TCP) window controlling the bandwidth usage.

The binomial algorithms class join the most commonly used congestion control algorithms by the generalization of their formula. The generalized formula is:

\[ I : w_{t+R} \leftarrow w_t + \frac{\alpha}{w^k_t}; \alpha > 0 \]
\[ D : w_{t+\delta} \leftarrow w_t - \beta w^{l}_{t}, 0 < \beta < 1 \]

where \( I \) refers to the increase in window as a result of the receipt of a window acknowledgements in a round-trip time (RTT) and \( D \) refers to the decrease in window on detection of congestion by the sender, \( w_t \) the window size at time \( t \), \( R \) the RTT of the flow, \( \alpha \) and \( \beta \) are constants.

For \( k = 0, l = 1 \), we get additive increase/multiplicative decrease (AIMD); for \( k = -1, l = 1 \) we get multiplicative increase/multiplicative decrease (MIMD) used by slow start in TCP (JACOBSON, 1988); for \( k = -1, l = 0 \) we get multiplicative increase/additive decrease (MIAD); and for \( k = 0, l = 0 \) we get additive increase/additive decrease (AIAD), thereby covering the class of all linear algorithms. They are called binomial congestion control algorithms, because their control expressions involve the addition of two algebraic terms with different exponents.

Binomial algorithms have some interesting properties that make them excellent to be congestion control algorithms, as showed by (BANSAL; BALAKRISHNAN, 2001), and presented here as well:

- Convergence to fairness: with some restrictions to the K and L values, the generalized binomial algorithm converge to fairness and efficient operating point;
- Reduction in oscillations: depending on the increase and decrease rule, the algorithm can be more responsive to network oscillations;
- Scale on bottleneck bandwidth: the bottleneck bandwidth gets distributed fairly among the sources even in the presence of multiple bottlenecks.

The Binomial algorithms are used as a mechanism of congestion control in TCP, and are in the "black box" category, which considers the network as a black box, assuming no knowledge
of its state, other than the binary feedback upon congestion. (BANSAL; BALAKRISHNAN, 2001) presented an extensive analysis of the Binomial algorithms, multiple comparisons, behavior rules, and how to get the best results using them.

2.4 Root Mean Square Error (RMSE)

Root-mean-square error (RMSE) is one of the most commonly used measures of differences between values (sample and population values) predicted by a model and the values actually observed. This value is computed by taking the average of the squared difference between each predicted value \( p_i \) and its correspond correct value \( a_i \). The root-mean-square error is the square root of the mean-squared error. The root-mean-squared error gives the error of the same dimensionality as the actual and predicted values. The RMSE is defined as follows:

\[
RMSE : \sqrt{\frac{(p_1 - a_1)^2 + \ldots + (p_n - a_n)^2}{n}}
\]
Monitoring traditional networks can become a hard challenge because of the quantity and diversity of the generated traffic. Furthermore, network devices such as switches, routers, etc., are inflexible and they cannot deal with different types of network traffic due to the underlying hardwired implementation of routing rules (YASSINE; RAHIMI, 2015). Consequently, these problems complicate the task by:

- Making it difficult for network operators to measure in short timescales the status and dynamics of the network in effective and efficient ways (TSO; PEZAROS, 2013);
- Requiring the ability to measure different types of network traffic at different time-scales for tasks such as traffic engineering and congestion detection to guarantee application performance;
- Putting network operators in unprecedented stressful situations to satisfy users’ expectations for delivering applications with guaranteed Quality of Service (QoS) that have ubiquitous accurate traffic monitoring and measurement mechanisms;
- Requiring today’s interactive media applications to capture performance degradation factors along the end-to-end network path.

The SDNs paradigm and OpenFlow are overcoming these problems. The centralized architecture of SDN allows that all the device state information be accessible at the controller for all the applications. This behavior easy the analysis of the state information, it allows a smooth sharing of the analysis, and it makes simple to take action. By centralizing the control plane, the SDN model provides a simplified working model for large networks that are characterized by highly dynamic workloads, user/device/application mobility, and policy-driven connectivity (GIGAMON, 2014).

The controllers and the OF protocol make possible to create solutions independent of manufacturer, specific technologies, or special instrumentation. The switches devices used in traditional networks are now simple forwarding elements without embed control or software. In theory, they do not have proprietary specific manufacturer setup configurations, which reduces misconfiguration errors that may result in very undesired network behavior (including, among others, packet losses, forwarding loops, setting up of unintended paths, or service contract violations). Indeed, while rare, a single misconfigured router is able to compromise the correct
The OpenFlow is the main southbound protocol used to connect the network devices to the SDN controller. Its standardization, the various messages, and the SDN architecture provides means to avoid misconfiguration. The **Read-State** message provides meaningful information of the network which allows SDN applications to make an insightful analysis of the network, and that had been the base for most of the network monitoring tools.

With the goal of gathering information from the network entities efficiently, researchers use differently the OpenFlow, the default protocol for southbound communications in SDN. These different approaches to get the information from the network entities are created because requesting deliberately is harmful to the network and it can generate a huge network overhead and stress the SDN controller.

As presented in Chapter 2, we use the five logical layers division for the monitoring process: **(1) Collection layer, (2) Representation Layer, (3) Report Layer, (4) Analysis Layer, (5) Presentation layer**. In SDN, the Report, Representation, and Collection layer are directly related with the OF protocol. An advantage of using OF is that the measurements follows the protocol standard.

The monitoring solutions in SDN can collect the statistics of the network devices actively, passively, or both. Most of the monitoring solutions use the events related to the state of the network, devices, flows, and others of the OF protocol as triggers to polling the statistics.

Additionally, they can associate some period to their routines, be pure event-driven, or they can use both technics. Most of the early monitoring solutions used only one type of routine, while the newest use both, which makes them be more flexible and non-network consuming.

Essentially, the monitoring solutions in SDN provide the monitoring information for other SDN applications, these other applications will analyze the information and act upon it. The provided data can vary from a simple network analysis to a complete one, or it can present a specific purpose analysis, e.g. traffic engineering, security, QoS.

It is ideal that the solutions generate information that is accessible internally in the SDN controller, and externally by some API with the collected statistics. However, this is not the case. In some cases, the monitoring solutions do not define any way to have its information accessible. Even when the researchers develop a POC implementation only, it would be helpful if an API was described at least. Furthermore, a Graphical User Interface (GUI) to present all the data allowing graph creation and data manipulation is decisive to help managers to understand the network state, trends, and solve problems.

Following the network monitoring logical layers, we classified the related works by how they execute each layer routines. Image 3.1 present our classification.

The Collection is the first logical activity. It is mostly divided in the literature as active and passive, and we followed this classification. Active entail works that inject traffic in the network to get measures with it. Passive entail works that just monitoring the traffic traversing...
the network, measuring it.

The advantages of active monitoring are that it is possible to measure whatever is needed, under whatever circumstances. We just need to inject traffic and measure. The passive monitoring does not have this advantage and must wait until the wanted traffic cross to measure it. However, it does not create any additional traffic, which may cause overhead in the network.

The Representation, as know as the normalization layer, is responsible to normalize the data gathered to a common. The OpenFlow protocol handles this routine already.

The Report function of the monitoring solutions is classified as event triggered and periodic. When the devices report the collected statistics when a specific event happens, they are classified as the former. When the devices report periodically they are classified as the latter. The devices can report in both ways and be classified as both.

The Periodic report gives the advantage of a constant influx of information, which can make sure a better understanding of the network state. However, it can overload the network, so controlling and tuning the time period fits the scheduling problem class, which is NP-Hard. Event trigger reports are useful in uncommon occurrences and to measure specific traffic occurrences. Furthermore, event trigger report can create a time-lapse void of information when no events trigger the report function.

The Analysis routine is sorted between the ones that do analyze the reported data and the ones that do not. Often, the solutions make the resulted information available. Moreover, some use the results to improve their own services.

The Presentation function is arranged into the ones that exhibit the information using a REST API, the ones that use a GUI, and the ones that did not define a presentation method. The usual approach is the REST API. GUI are scarce, and some works do not define the Presentation function because the general focus is the monitoring itself.

It is a functional requirement for every monitoring solution to have a method to present the analyzed information. However, some works do not want to provide a monitoring solution, just an improvement to the monitoring field and its functions, and because of that, there are works that do not explain how the data will be presented. This is common, but we do not think this is a health behavior because this makes the researchers do not think how they will provide the information, sometimes making their improvement unfit in a real-world case.

In the set of works that implement a method to present their data, there are two approaches: REST API and GUI. Presenting the data with a REST API is a more versatile choice, which allows the network manages to use the API with some other application, or use it in their own application. Meanwhile, the GUI approach provides a graphical visualization of the data generally with no API.
3.1 Literature Review Methodology

The literature review acted as the groundwork to write this dissertation. We searched for the most cited and newest SDN network monitoring proposals. Our goal was to form a solid understanding of the field, and, afterward, we wanted to find how the network monitoring problems were being addressed.

We did our research in Elsevier (ELSEVIER, 2017), IEEE Xplore (XPLORE, 2017), Springer (SPRINGER, 2017), and Google Scholar (SCHOLAR, 2017). We used the same keywords to search at each base. We started choosing the newest surveys of the field to get a general overview. Afterwards, we singled works with alike proposal to ours. Them, we did a thoughtful quality evaluation of the works, discarding the ones we judged poor in quality. As the result of this process, we sorted eleven related works to compare with our approach.

The classification of the works presented in figure 3.1 was the result of the investigation of the network monitoring surveys and field study. They provided the common division of the monitoring activities, and how they were handled generally.

Another enlightenment provided by our literature review was the most common problems present in the monitoring solutions. They were summarized as follows:

- capture a general status of the network;
- capture an instantaneous status of the network;
- packet inspection;
- packet loss;
- network overhead;
- latency;
- scale mechanism.

We will explain in each work subsection how each solution solved the problems if they were solved, and summarize this information in the final remarks of this chapter.

3.2 Related Works

Network monitoring in SDN, in the last decade, was an active field of research with a myriad of works, which addressed the problem with different approaches. The following image 3.2 shows a timeline of the monitoring solutions.

These solutions present methods for optimizing statistics collection, different data aggregation levels of the monitored information, multiple sampling methods, insightful analysis of the collected information, and much more innovations to the monitoring field.
3.2. RELATED WORKS

<table>
<thead>
<tr>
<th>Solution</th>
<th>Collection</th>
<th>Report</th>
<th>Analysis</th>
<th>Presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSAFE</td>
<td>Active</td>
<td>Event-triggered</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PayLess</td>
<td>Active and Passive</td>
<td>Periodic</td>
<td>Yes</td>
<td>REST</td>
</tr>
<tr>
<td>OFMon</td>
<td>Passive</td>
<td>Periodic</td>
<td>Yes</td>
<td>REST and GUI</td>
</tr>
<tr>
<td>Floware</td>
<td>Passive</td>
<td>Event-triggered</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>FleXam</td>
<td>Passive</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FlowCover</td>
<td>Active</td>
<td>Periodic</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>OpenSample</td>
<td>Passive</td>
<td>Periodic</td>
<td>Yes</td>
<td>REST</td>
</tr>
<tr>
<td>OpenTM</td>
<td>Active</td>
<td>Periodic</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>OpenNetMon</td>
<td>Active</td>
<td>Periodic</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>FlowSense</td>
<td>Passive</td>
<td>Event-triggered</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Isolani et al.</td>
<td>Active</td>
<td>Periodic</td>
<td>Yes</td>
<td>REST</td>
</tr>
</tbody>
</table>

3.2.1 OpenSAFE

The first monitoring solution is OpenSAFE (Open Security Auditing and Flow Examination) (BALLARD; RAE; AKELLA, 2010). It is a system for enabling the arbitrary direction of traffic for security monitoring applications at line rates. It can handle any number of network inputs and manage the traffic in such a way that while filtering packets at line rate it can be used by many services.

OpenSAFE consists of three important components: a set of design abstractions for thinking about the flow of network traffic; ALARMS (A Language for Arbitrary Route Management for Security), a policy language for easily specifying and managing paths; and an OpenFlow component that implements the policy. Additionally, ALARMS is a flow specification language created to simplify management of network monitoring appliances.

OpenSAFE works using the idea of paths as the basic abstraction of describing the selection of traffic and the route this particular traffic should take. Several parts compose a path: inputs, selections, filters, and sinks. Inputs can only produce traffic, sinks can only receive traffic, and filters must do both. At a high level, each path begins with an input, applies an optional choice criterion, routes matching traffic through zero or more filters, and ends in one or more sinks.

To test OpenSAFE and ALARMS they used a software implementation of OpenFlow using virtual machines. (BALLARD; RAE; AKELLA, 2010) claim that using OpenSAFE makes monitoring large-scale networks easier than ever before, but this work did not present any result.
3.2. RELATED WORKS

3.2.2 PayLess

PayLess proposes a low-cost efficient network statistics collection framework (CHOWDHURY et al., 2014). It was built on top of an OpenFlow controller’s northbound API and provides a high-level RESTful API. The monitoring framework takes care of translate high-level monitoring requirements expressed by the applications. It also hides the details of statistics collection and storage management.

It provides an abstract view of the network and a uniform way to request statistics about the resources. PayLess was developed as a set of pluggable components. Well-defined interfaces abstract the interaction between these components. Hence, one can develop custom components and plug into the PayLess framework. Varying tasks, like data aggregation level and sampling method, is customizable in PayLess. Furthermore, it studies the resource-accuracy trade-off issue in network monitoring and proposes a variable frequency adaptive statistics collection scheduling algorithm.

The network monitoring applications, built on top of this framework, use the RESTful API provided by it and will remain shielded from the underlying low-level details.

To validate their framework, they evaluated the link utilization and the overhead generated by monitoring the network utilization, measuring the overhead, measurement error, and accuracy. The experiments showed that PayLess could follow the link utilization, and got almost every traffic spike but not all. The overhead generated was less than the default periodic polling, and in some cases, it sends 50% less, although with a higher level of accuracy the network overhead increases twice from the lower level of accuracy.

3.2.3 OFMon

OFMon is an OpenFlow based monitoring tool, designed in accordance with the architecture of ONOS (Open Network Operating System), which monitor all OpenFlow messages between ONOS controllers and OpenFlow switches (KIM et al., 2016). ONOS, is a concept of distributed SDN controller that has been recently proposed to mitigate the bottleneck problem. However, it still has problems to decide when and how to distribute the control plane workload due to the lack of performance assessment provided by ONOS, thus OFMon was created to address this issue.

OFMon is divided into four components: Database, OFMonitor, CLI-based application, and GUI-based application. The Database component has the cumulative monitoring results of OpenFlow messages which are transmitted to and received from switches in the SDN network. The OFMonitor component records the monitoring results in Database when the OpenFlow messages are transmitted and received. The CLI-based application component provides a monitoring application using CLI. Finally, the GUI-based application component provides a monitoring application using Web-based GUI.

In order to evaluate the performance of OFMon, citeauthorKim2016 compared the
performance of the ONOS controller with and without OFMon. The metrics are the central processing unit (CPU) and memory used to determine the performance requirements of OFMon. The experiments showed that ONOS with OFMon has a 4% higher average usage of CPU that without OFMon. Furthermore, when the number of switches increases, the average of using CPU increases as well. The memory comparison showed that ONOS with OFMon consumes more memory, up to 6% than without OFMon. However, the average memory usage between ONOS with and without OFMon is stable, even if the number of switches increases.

3.2.4 Floware

Floware is an OpenFlow application that allows discovering and monitoring of active flows at any required aggregation level (NACSHON; PUZIS; ZILBERMAN, 2016). Floware balances the monitoring overhead among many switches to reduce its negative effect on network performance. In addition, Floware integrates with monitoring systems based on legacy protocols such as NetFlow. It proposes a new Flow Discovery technique that requires only few more control messages and flow-table entries distributed across the network to avoid overload to discover the individual flows whose statistics need to be collected.

Its main modules are responsible for: (a) generating the relevant flow-discovery entries (b) assigning them to switches (c) scheduling the expiration of active flows and (d) exporting flow statistics to the remote flow analyzer.

The focus of their evaluation was on the effect of flow assignment strategies on Floware performance. During the experiment, the number of flow-table entries that were installed (denoted as total flow entries) including flow-discovery entries, active flow entries, and other entries installed by the controller were recorded. The load on the switches can become even more dispersed if the monitoring load is not well-balanced, so they used Gini coefficient (Gini, 1912) to measure the dispersion of free flow-table entries in the network switches.

In order to get deeper insights into network performance during monitoring, the number of packet-in messages was detached for the monitoring and for routing purposes (denoted as routing packet-in messages and monitoring packet-in messages respectively), and the use of the memory of the controller was measured.

The results of their evaluation showed that balancing the monitoring load across switches using their new flow discovery technique greatly reduces the chance for full flow-table errors compared to using only the baseline for monitoring. Corroborating with the first result, they presented that the more balanced the distribution of free flow-table entries is (smaller Gini coefficient) the less redundant packet-in messages are in the network.

With a balanced distribution of flow records, it was possible to completely avoid errors (and access control messages) with only 900 entries in the flow-tables of the switches, in comparison with 2,400 entries when the baseline algorithm is used to collect statistics. In addition to saving switch resources, the proposed monitoring optimization saves controller
resources as can be seen from the lower memory consumption of the controller.

Floware research showed that it is possible to bring legacy network monitoring solutions to the SDN environment. Further, they presented an algorithm to lower how many flow entries were installed on the flow-tables, and how to use the new OpenFlow 1.3 pipeline to completely decouple flow monitoring and routing decisions. However, they did not investigate the network overhead resulted by introducing the legacy network system, which is a great deal by the number of new network entries installed at the switches because of NetFlow. Another problem is that as Floware monitors the network statistics passively, it can not measure network utilization instantly.

3.2.5 FleXam

FleXam is a flexible sampling extension for OpenFlow that enables the controller to reach packet-level information (SHIRALI-SHAHREZA; GANJALI, 2013). Simply stated, the controller can define which packets should be sampled, what part of the packet should be selected, and where they should be sent.

Packets can be sampled stochastically (with a predetermined probability) or deterministically (based on a pattern), making it flexible for different applications. At the same time, it is simple enough to be done entirely in the data path. The controller can also request switches to only send part of packets that are needed (e.g. headers only, payload, etc.) and define where they should be sent, make it possible to easily manage and distribute the load.

FleXam includes two types of sampling: (1) select each packet of the flow with a probability of $\rho$, and (2) select $m$ consecutive packets from each $k$ consecutive packets, skipping the first $\delta$ packets. The first case is the stochastic sampling. The second case is a generalized version of the deterministic sampling.

FleXam is a proposition for an OpenFlow extension. They did not experiment, but they explained how it should be implemented in a OF switch and the changes needed to be done in the OpenFlow protocol.

3.2.6 FlowCover

FlowCover is a low-cost high-accuracy monitoring scheme to support several network management tasks inspired by the visibility and central control of SDN (SU et al., 2014). Moreover, most of the existing approaches select the polling switch nodes by sub-optimal local heuristics. They leverage the global view of the network topology and active flows to decrease the communication cost by formulating the problem as a weighted set cover, which is proved to be NP-hard. Heuristics are presented to get the polling scheme efficiently and handle flow changes practically.

Their approach tries to cut the communication cost of monitoring by aggregating the polling requests and replies. They leverage the global view of SDN to optimize the monitoring
strategies. The polling scheme changes dynamically in accord with the traffic crossing the network.

They provide a general framework to ease various monitoring tasks such as link utilization, traffic matrix estimation, anomaly detection, etc. They introduce a globally optimized flow statistics collection scheme. Their approaches select target switches by the view of all active flows instead of on a per-flow basis.

FlowCover was evaluated from different aspects such as the reduced communication cost, overheads, accuracy and the performance of handling flow changes. In the reduced communication cost experiment showed that FlowCover saves up to 47.2% of the total communication cost, regardless of the network topology and the number of the active flows. For the overhead, they examined the construction time of the weighted set cover problem and the polling scheme calculation time. The problem construction time occupies roughly 10% of the total calculation time.

To evaluate the accuracy of FlowCover they used two metrics: accurate flow ratio (AFR) which indicates the percentage of accurate flows; average accuracy of traffic matrix (TM) estimation which represents the error between the measured and real traffic matrix. The AFR fluctuates around 90% in both topologies. The accuracy of TM estimation was always above 99%. By the increasing packet loss rate, they showed that the AFR resists to it and the accuracy of TM estimation falls gradually from 99.9% to 98.1%.

The flow changes evaluation showed that the flow change heuristic does not increase too much communication cost compared with the always re-compute method. This is because most of the new flows have been covered by the current polling scheme.

FlowCover shows an interesting approach to the monitoring problem, creating a polling scheme to balancing which switches to request and which flow information to request from them. However, their polling scheme makes impossible to measure the network utilization on the switches that wildcard matches were used to collect statistics. Another problem is the static polling time, which may be good to estimate traffic matrixes, but is a problem to measure every type of flow in a network.

3.2.7 OpenSample

OpenSample is a low-latency, sampling-based network measurement platform targeted at building faster control loops for software-defined networks (Suh et al., 2014). It uses sampling for network monitoring, which allows it to have a 100 millisecond control loop rather than the 1–5 second control loop of earlier polling-based approaches.

The architecture of OpenSample uses i) packet sampling to capture packet header samples from the network with low overhead and ii) uses TCP sequence numbers from the captured headers to reconstruct nearly exact flow statistics.

OpenSample uses a single, centralized collector that combines samples from all switches
3.2. RELATED WORKS

in the network to build a global view of traffic in the network at both flow and link granularity. Exploiting TCP information to increase estimation accuracy for any given sampling ratio. This TCP-aware analysis is the key innovation that OpenSample incorporates.

To evaluate OpenSample the authors compared its loop control against earlier counter-polling based approaches. OpenSample-TCP performs significantly better than either polling or OpenSample-MLE because it detects and schedules elephant flows earlier. In most cases, it achieves performance close to a nonblocking switch, even for small (1MB) flows. Often it outperforms the alternatives by 25-50%.

The scalability of the OpenSample collector was evaluated as well, and it was able to process more than 100,000 samples per second. This means that their current OpenSample implementation can handle samples from at least 285 switches assuming each switch sends 350 samples per second. That implies a single OpenSample is able to handle production data centers servicing 4K or 8K hosts with 1:2 or 1:5 oversubscription ratio, respectively.

However, they use the fact that most traffic sent in data centers today is TCP, utilizing the TCP header information to measure the flow average rating, which may help to detect elephant flows earlier, but will result in problems to detect network jitters, and packet drops. Furthermore, as it is not using the switches’ counters to get the real rates of the flow, it will not reflect the real instantaneous utilization but an approximation of it.

3.2.8 OpenTM

OpenTM is a traffic matrix estimation system for OpenFlow networks. OpenTM uses built-in features provided in OpenFlow switches to directly and accurately measure the traffic matrix with a low overhead (TOOTOONCHIAN; GHOBADI; GANJALI, 2010). Additionally, OpenTM uses the routing information learned from the OpenFlow controller to smart choose the switches from which the flow statistics could be obtained, thus reducing the load on switching elements.

OpenTM’s logic is quite simple. It keeps track of all the active flows in the network, gets the routing information from the OpenFlow controller’s routing application, discovers flow paths, and periodically polls flow byte and packet-count counters from switches on the flow path. Using the routing information, OpenTM constructs the TM by adding up statistics for flows originated from the same source and sent to the same destination. OpenTM can create different types of TMs with different aggregation levels for sources and destinations.

They propose different switch querying strategies: (a) querying the last switch, (b) querying switches on the flow path uniformly at random, (c) round-robin querying, (d) non-uniform random querying that tends to query switches closer to the destination with a higher probability, and (e) querying the least loaded switch.

To evaluate OpenTM they measured their coverage rate and compared their different query strategies measuring the average throughput of each one. The coverage rate measured in all cases is within 50 seconds or just 10 queries.
The comparison between the strategies showed that the non-uniform random querying method has the best performance, as it tends to query switches closer to the destination with higher probability. Both the round-robin and uniform random querying methods are performing very close to each other, but they were worse than the non-uniform querying method. However, the overall difference among all these schemes was relatively small.

While OpenTM results were good enough to get the traffic matrix of the network, it can not measure network utilization instantly because of the five second static polling time. Additionally, it uses different strategies to select which switch to query the flow information, making it impossible to get all utilization regarding one switch.

3.2.9 OpenNetMon

OpenNetMon is an open-source software implementation to monitoring per-flow metrics, especially throughput, delay, and packet loss in OpenFlow networks (ADRICHEM; DOERR; KUIPERS, 2014). OpenNetMon continuously monitors all flows between predefined link-destination pairs on throughput, packet loss, and delay. In order to determine throughput for each flow, as well as packet loss, OpenNetMon regularly queries switches to retrieve Flow Statistics, polling on regular intervals for every distinct assigned path between every node pair that is designated to be monitored.

To ensure the max flow coverage, and because the arrival of flows can vary greatly, OpenNetMon monitors flows with an adaptive behavior, by increasing the polling intervals when flows arrive or change their usage characteristics and decreasing the polling interval when flow statistics converge to a stable behavior.

The adaptive nature of OpenNetMon might also be useful in avoiding excessive route flapping when flows are reallocated based on a new fine-grained view of the state of the network.

Per-flow packet loss is computed by OpenNetMon by polling flow statistics from the first and last switch of each path, and, then, subtracting the increase of the source switch packet counter with the increase of the packet counter of the destination switch. To calculate path delay, it regularly injects packets, at every monitored path, at the first switch, such that probe packet travels exactly the same path, and have the last switch send it back to the controller. The controller estimates the complete path delay by calculating the difference between the packet’s departure and arrival times, subtracting with the estimated latency from the switch-to-controller delays.

The measurement of accuracy and the packet overhead match the size of each flow is done by injecting packets, as well, for each path with a rate relative to the underlying sum of flow throughput.

To evaluate their solution, the authors use a video stream to model traffic, comparing with Tcpstat. The throughput measurements performed by Tcpstat and OpenNetMon, only differ on average by 16 KB/s (1.2 %), which shows that most of the transmitted traffic is taken into
account by the measurements. However, the standard deviation is 17.8% which is a high value, but they justify it by a lack of synchronization between the two measurement setups. The packet loss calculation is not accurate, but the measurements give a good enough estimation to detect service degradation.

The delay measurements were calculated in the control plane, in an isolated VLAN in the data plane, and as experienced by the end-user application. The control plane calculations showed that it is unsuitable to use as a medium for time-accurate delay measurements. However, the calculation on the data plane worked, and, as expected, the measurement in the VLAN showed a lower delay than the end-user application experience.

OpenNetMon approach for monitoring the network is interesting and to a certain point accurate. The issue seems to be the static polling time that may request the information from the switch a moment before the switch updates the counters, making the counter of the switch updates two times before the next request, generating spikes in the measurement. Their packet loss measurement is not very accurate, but it gives a good enough estimation to detect service degradation.

3.2.10 FlowSense

FlowSense is an approach to performance monitoring in flow-based networks. The authors propose a new approach for high accuracy utilization monitoring without measurements cost (YU CRISTIAN LUMEZANU, 2013). Rather than rely on on-demand active polling of switch counters, they infer performance by passively capturing and analyzing control messages between the switches and the centralized controller. Their key insight is that control messages sent by switches to the controller carry information that allows them to estimate performance.

They measure link utilization (the bandwidth consumed by flows traversing the link) in OpenFlow networks. They are able to compute utilization only at discrete points in time. These checkpoints are determined by FlowRemoved arrivals at the controller and by the values of the timeout associated with the expired entry. However, it is limited to reporting the average utilization over a flow entry’s duration and cannot capture instant utilization at any point in time.

To evaluate FlowSense, the authors chose as metrics accuracy, the refresh rate to estimate the link’s utilization, and quickness to estimate the link’s utilization at a specific time. They compared the utilization obtained by FlowSense with that gathered from continually polling switches A and B at 1s intervals. FlowSense reports utilization values that are similar to those inferred through polling, with a tiny delay, because of the passive monitoring behavior.

To evaluate the granularity of the measurements, a simulation of FlowSense was performed using real-world enterprise traces. They computed the average time between Flow Removed events, under the assumption that all flows arrive on the same link, and find out that a flow expires, and thus enables them to refresh the utilization measurements, every 16ms. They computed the average time between two consecutive utilization check-points for each port, and
for half of the incoming links, the average time between two utilization measurements is at most one second and for almost 90% of the links under 3 seconds.

Related to quickness to estimate the link’s utilization in a specific time, they showed that FlowSense would have to wait more than 100s to capture the complete utilization. However if an application is willing to trade-off some accuracy for timeliness, it can have a reasonable estimate of a link’s utilization at a particular checkpoint under 10s, rather than having to wait for 100s.

Flowsense measures the network passively, collecting the statistics when the flow rules are removed from the switches. While this results in a good traffic matrix, it is not enough to measure the instantly network utilization. To enable it to monitor the instantly utilization all flow entries should have a tiny hard timeout, which would impact the router application of the Controller, as well as create an overhead on the resource usage of the switches and Controller.

### 3.2.11 Isolani

(ISOLANI et al., 2015) proposes an interactive approach to SDN management through monitoring, visualization, and configuration that includes the administrator in the management loop. Their solution includes three main components: Monitoring Manager, Visualization Manager, and Configuration Manager.

Monitoring Manager is responsible for retrieving updated information about the network and storing it in a local Database. Visualization Manager comprises the Statistics Processing and Chart Visualizations modules. With information stored in the Database, the Statistics Processing module is able to aggregate data per host, switch, controller or even the entire network to be used by the Chart Visualizations module. Configuration Manager allows the Administrator to check and configure SDN-related parameters on network controllers through the management interfaces.

To evaluate their approach, the authors measure control channel load and resource usage considering the administrator interactions over the experiment time span. They vary the SDN controller configuration parameter of idle timeout, and they also change the monitoring polling interval to understand the impact of Read-State messages as well.

The first value of idle timeout is set to 5 seconds, and with this value, the controller is processing a large number of Packet-In messages. With the value changed to 60 seconds is possible to visualize a dramatic decrease in control packets processed both by the controller and network devices. However, this configuration also affects immediately resource consumption, especially in terms of idle rules and control traffic rate towards the controller. With this rule, idle timeout configuration nearly 77.7% of all forwarding rules remain idle and upload control traffic increases almost threefold.

With the value in 60 seconds, the traffic towards the controller remains very high and is mostly composed of Read-State reply messages. The idle timeout, then, changes to 40 seconds and the control traffic rate generated by these messages is much reduced. They also noted that
to a flow rule is classified as idle, it needs to be monitored twice without changing its counters, which happens when the idle timeout is between 5 and 15 seconds.

(ISOLANI et al., 2015) approach allows the monitoring solution to control the polling interval, which results in more or less accurate measurement. However, when the polling interval is lower enough to capture instant network state, the generated network overhead is not worth it. Furthermore, the solution can be used to request information in a polling rate that gives enough information to generate a general traffic matrix and does not overuse the network. Although their monitoring solution did not have a novel approach, the authors evaluation of network monitoring, the OpenFlow messages, and the resource usage by switches and Controller are insightful.
### Related Works Classification

**Active**
- Entails injecting test traffic onto a network and monitoring the flow of that traffic

**Passive**
- Entails monitoring traffic that is already on the network

**Representation**
- **OpenFlow**
  - Uses OpenFlow standard to represent the metrics

**Report**
- **Periodic**
  - Entails periodic reports of statistics

**Event**
- Entails event-driven reports of statistics

**Analysis**
- **No**
  - Entail solutions that do not analyze the statistics collected

**Yes**
- Entail solutions that analyze the statistics collected

**Presentation**
- **REST**
  - Entail data presentation by a REST API

- **Not Defined**
  - Entail solutions that did not define a data presentation

- **GUI**
  - Entail data presentation by a GUI

---

**Figure 3.1** Explanation of the related work classification

**Figure 3.2** Related Works Timeline
3.3 Final Remarks

As can be observed, even with all the research in network monitoring field, the solutions fail to achieve a set of requirements essential in network management. Most of them can determine the general network status and create a general traffic matrix. However, none can determine instant network status, with minimal overhead and a scale mechanism. The ones that can be accurate enough to catch the necessary information use too much network resource and have a high error rate.

Besides, simple measurements of packet loss for a flow and delay measurement between network devices are important information for a network manager. While their approaches to the problem are different, novel, and contribute to the network monitoring problem, none of them are ideal to be used in a real environment. Table 3.2 shows the requirements a monitoring solution should have and the impact of the solution on the network environment based on our review of each of them.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Status</th>
<th>Instant Status</th>
<th>Packet Inspection</th>
<th>Network Overhead</th>
<th>Packet Loss</th>
<th>Latency</th>
<th>Scale Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSAFE</td>
<td>-</td>
<td>-</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>PayLess</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Great</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>OFMon</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Minimum</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Flowware</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Great</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FlowCover</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Minimum</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>OpenSample</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Minimum</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenTM</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Minimum</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>OpenNetMon</td>
<td>Yes</td>
<td>Partial</td>
<td>No</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>FlowSense</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Minimum</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Isolani</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Great</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.2 lists six network monitoring functions that all network monitoring solution should have:

1. status: collect enough information to present a general status of the network;
2. instant status: collect information timely, allowing it to present an instant status of the network;
3. Packet Inspection: a packet inspection routine or should allow packet inspection routine from others solutions;
4. Packet Loss: calculate the packet loss rate of the network traffic or allows that it can be measured;
5. Latency: calculate the delay between its nodes;
6. Scale Mechanism: a scale mechanism so that the monitoring solution can grow with the infrastructure without any drawbacks.

All the related works are presented in the table 3.2, which describes if each one attends the six monitoring functions. The network overhead column is not a network function but is a requirement for every network solution that they generates the least possible network overhead, so that it does not degrade the running services. The values of this column vary from least, normal, to great in ascending order of network overhead generated. All the passive monitoring tools generate minimum network overhead because of their nature and the active monitoring solutions that only collect information to present a general status too.

Payless and Isolani solutions are capable of presenting an instant status. However, to do so, they reach a great network overhead. OpenNetMon authors’ mentioned in their research that their solution is able to capture a snapshot of the network, but, no result was shown, as well as no result about the network overhead of this feature.

This thoughtful analysis of the latest and significant network monitoring research point to the necessity of a solution that can give the basic network monitoring functions with little degradation. This, united with the real cases explained in Chapter 1, are strong evidence that the OpenFlow networks need a monitoring solution that can better handle these issues. So we propose our solution SHAMon, a novel monitoring solution, that to the best of our knowledge is the most complete solution presented.

At the end of the table 3.2, our SHAMon, is listed. Our solution is short on the latency function and does not have a packet inspection routine. The latency function is troublesome to calculate in OF networks as presented by (ADRICHEM; DOERR; KUIPERS, 2014), so we propose it as a future feature. The packet inspection routine is a complex task which can be done by another SDN application with no concerns related to our solution.
4.1 Overview

Even with a rich number of works in the SDN network monitoring field, still, there are areas for further development. And looking to fill the gaps and the already presented problems, we researched and developed Scalable High Accurate Monitoring (SHAMon), a novel SDN network monitoring solution. Our approach differs from the others because of our scheduling algorithm, with a per-flow granularity which can be aggregated, and a REST API that allows all the collected statistics to be used by other applications.

SHAMon was created to be an SDN application, to fully use the OpenFlow protocol and the northbound API to account the network devices, acknowledge new flows in the network, and collect statistics from the devices. These routines are required for SHAMon to work properly, as well as to monitoring the network.

The figure 4.1 shows exactly how SHAMon solution fits in the SDN architecture. Our solution works along with others applications in the Controller, exchanging information, and using the Controller interfaces.

We designed SHAMon considering the five logical divisions of the network monitoring task explained in Chapter 3. So, we choose how best execute each logical function ensuring that they followed our main goals. This task is not an easy one because we must balance between
accuracy, granularity, and lower network overhead.

Hence, for the Collection activity, we decided to collect statistics actively and passively. This choice was motivated by:

1. the OpenFlow protocol structure - the protocol has a built-in procedure that enables the SDN Controller to receive all the statistics of a flow rule when it is finished. This is the passive mechanism used by most of the SDN passive monitoring solutions;
2. passive OF built-in mechanism is not enough - while we used the built-in mechanisms of the OF protocol to passively monitor the network it is not enough to form a picture of the network state, instantaneous or not.
3. flows that do not end - if an SDN application install a flow with no time-out, that flow needs to be actively monitored because the passive mechanism would never be executed.

The OpenFlow protocol normalizes our information like in the presented related works. We decided to use periodic polling of information from the devices and event-based polling. The event-based polling mechanism is the already explained passive mechanism for the finished flows. For the periodic polling, we implemented a closed loop to control the period which the information is reported. We used the statistics from the devices to update the time period. The full explanation of the reporting process will be done in section 4.2.2.

With each wave of statistic reported we do an analysis to calculate the throughput and packet drop rate. We make an instantaneous status measurement of the network, and keep an average measurement of the network status. When this measurement is ready we make it available via REST.

So, following table 3.1 our solution handle the monitoring functions as presented in table 4.1.

<table>
<thead>
<tr>
<th>Table 4.1</th>
<th>SHAMon Classification according to the five network logical functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification of Monitoring Solutions in SDN</td>
<td></td>
</tr>
<tr>
<td>Solution</td>
<td>Collection</td>
</tr>
<tr>
<td>SHAMon</td>
<td>Active and Passive</td>
</tr>
</tbody>
</table>

In the following subsections, we will explain in detail how SHAMon was designed, how it works, and how we address the two main concerns aforementioned.

4.2 | Shamon Architecture and Functions |

There are some requirements that a network monitoring solution must fill so that it can be really used in a network. The main one is that it can not generate stress nor overhead in the
communication channel that it is monitoring. This is an easy to see demand, since if the network solution floods the communication channel the traffic of the running services will deteriorate.

Another related problem is the usage of CPU and memory from SDN applications. Monitoring solutions generally have a small CPU usage, however, they are memory consuming. Additionally, allowing the monitored data to be used by other applications, internal or external, is a requirement for the SDN architecture and helps the development of different applications to solve specific network problems.

We designed an architecture that could address these problems. We dealt with the overhead problem by first controlling the polling of statistics using binomial algorithms presented in section 2.3. The full explanation of how we used it is done in section 4.2.2. This control allowed our solution to smartly pace our statistic polling, hence making our solution conscious of the network state.

The second step to deal with the overhead generation was that we decided to use a Proxy-Server architecture. In this scheme, our solution has two entities: the one that manages, the server, that we named SHAMon Controller. And the one that works as a proxy gathering the statistics, that we named SHAMon Proxy. The controller entity is placed with the SDN Controller, and the proxy ones can be placed dispersed at the network.

It is important to mention that our controller entity has all the necessary features to monitor the network, and both the SHAMon controller and SHAMon proxy can collect network statistics from the OF devices. We decided to use this architecture because:

1. it is a well-established architecture used by several monitoring solutions like Zabbix (ZABBIX, 2016) and OpManager (MANAGEENGINE, 2016);
2. with this architecture it is possible to set proxies scattered at the network near the network devices so that the polling messages and answers do not cross the network. Instead, the proxies query the statistics of the devices that it is handling, compress, and send the bulk of information to the SHAMon Controller. This feature, if carefully done, can minimize the network overhead;
3. it allows that our solution scale with the network growth;

Figure 4.2 shows how our solution is placed in the SDN network environment with no SHAMon Proxies. The SHAMon Controller application placed with the SDN controller uses the OpenFlow protocol to query statistics from the active flows in the network directly from the switches.
Figure 4.2 SHAMon setup without proxies
Figure 4.3 shows how our solution is placed in the SDN network environment with SHAMon Proxies. The SHAMon Proxies are assigned to monitor the near OpenFlow devices querying them for statistics and sending the collected info to the SHAMon Controller. The SHAMon Controller will monitor the OpenFlow devices that are not being monitored by any SHAMon Proxy, querying them directly.

![Figure 4.3 SHAMon setup with proxies](image-url)
We designed SHAMon controller and SHAMon proxy separating the functionalities of them into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

SHAMon controller can be divided into four modules: (1) REST, (2) Controller module, (3) Scheduler module, and (4) Storage module as showed in figure 4.4. Each module will be explained in the following subsections.

SHAMon proxy can be divided into three modules: (1) Controller module, (2) Scheduler module, and (3) Communication module as showed in figure 4.5. The first and second module has functions similar to the SHAMon controller modules and the third one will be explained in the subsection 4.2.5.
4.2. SHAMON ARCHITECTURE AND FUNCTIONS

4.2.1 Storage Module

A history of the taken network measurements is a highly valued information to an administrator because it is a difficult task to understand the network state and its behavior, so, it helps to have previous knowledge. Hence, the storage of the collected statistics becomes an essential requirement for any network monitoring solution.

Moreover, the granularity and frequency are critical factors for the storage activity itself. The higher both of them are pushed, more memory and resources will be needed.

In SHAMon we store information at flow level, which is the most detailed information that can be collected from the OF devices. To balance against the highly detailed level of information being stored by our module, we allow the storing frequency of the data to be configurable, so it can suit the network manager necessities.

This setup allows the information needed to replay the traffic of the network to be available under the frequency of the captured data. One may notice that from the stored data is possible to derive the overall traffic of the OF devices, and with some manipulation, it is possible to aggregate or segregate in several levels, which facilitates the understanding of the network state.

We designed the storage process with two levels, the first one a memory level storage using hashmaps data structures, and the second one a database level storage. The newest collected statistics would be stored in the first level and after a configurable amount of time, the information would be stored in the database. This approach allows a fast query to the most recent information and, if needed, a calm and thoughtful analysis of the data stored in the database.

4.2.2 Scheduler Module

The main activities of this module are to listen to new flows that arrive and to schedule new requests of statistics of the acknowledged flows. It may seem easy at first glance, however, how to address this tasks following the requirements of a monitoring solution make it a difficult problem. This module is present in the SHAMon Controller and Proxy and it does the same task in both of them.

There are two general approaches to make flow rules in the SDN environment: we can do them statically with the SDN Controller, and we can do them dynamically when a new communication between nodes from the network is attempted. Usually, both approaches are used, the default rules that maintain the network are made statically, and there is an SDN application that makes new rules for new communications.

Figure 4.6 explains the dynamic process of flow rule creation of a SDN Controller.

Because of these approaches, our Scheduler module must check the flow rules written in the OF devices to track them. Moreover, it listens to dynamically created flows too. With the record of all flows created and active at the network, the scheduling of requests for statistics can begin. So, besides listening to flows, we must track the devices connected to the SDN Controller.
to ensure that we are tracing all the flow rules.

However, there is a little difference between how the SHAMon Controller and SHAMon Proxy start to seek the flows and devices. The SHAMon Controller becomes active with the SDN Controller and starts to monitor every OF device that the SDN Controller is handling. When a SHAMon Proxy is configured, it does the OF handshake with the device that it will monitor, registering itself. When the registration is done, the SHAMon Proxy communicates with the SHAMon Controller to release the device, then it starts to collect statistics of all the active flows from the device. It is important to notice that the device is still managed by the SDN Controller, although the monitoring is done by the SHAMon Proxy.

As already explained in section 4.1, there are two methods to gather statistics using the OpenFlow protocol: the first one is when installing a new flow rule at the OF device, we request that the device sends a FlowRemoved packet with the latest flow statistic. The second one is to query the OF device statistics about the active flows or other high-level aggregations. These two messages were presented in Chapter 2, and figure 4.7 shows how it works.

The two methods are the default procedure to monitoring when using OpenFlow protocol. Our solution utilizes both. One may think that get the statistics when a flow ends may be wasted work, however, the information that comes in the FlowRemoved packet is the final accounted statistics, and actively request statistics may not get the final counters.

All instances of SHAMon with the Scheduler module are configured so that, wherever a flow ends, the devices which the flow was transversing send a FlowRemoved message with the latest statistics. While this is enough to know how much traffic passes through the flow, it is not enough to understand the behavior, or even to get an instant report of a flow.

Because of that, during the flow life, the Scheduler module queries the OF device for the flow’s statistics. However, the query procedure must be done in a responsible way, so that the network does not suffer any overhead. Solve the scheduling problem of statistic requests for all devices across the network is the second main function of this module.
Job scheduling is a well known NP-Hard computer science problem, and the solutions vary according to which parameters must be maximized or minimized. In our case, we want the minimal necessary quantity of request queries crossing the network that can guarantee maximum accuracy.

In order to do that, we are monitoring each flow individually so that we can provide the best possible accuracy. Monitoring each one separately guarantees that we get individual information as well, which we use in our scheduling algorithm. However, using this approach makes difficult to minimize the number of queries. Even with the downside of the burden caused by this approach, we chose it because we believe the benefits cover the problems.

To address the problem of monitoring flows individually our scheduling algorithm schedules flow requests separately and differently. Our algorithm uses information from the collected statistics of the flow to manage the interval between requests dynamically.

We chose to use the number of packets and the number of bytes that traverse the flow during the time between the last statistic and the new one as the information to help control the scheduling. These two counters are important because they easy the identification from elephant flows and ultra-active flows to barely active flows, helping to manage the frequency of requests.

Based on TCP successful examples with congestion control algorithms to handle the traffic in the network, we use binomial algorithms to control the frequency of the requests, which are a class of nonlinear congestion control algorithms, as explained in chapter 2.

The adaptation of the binomial algorithm functions for our solutions is as follow:
Every monitored flow starts with a default request time for statistics and as the behavior of the number of packets and the number of bytes change we increment or decrement the request time.

We adapted the $I$ function to increment the request time, and the $D$ function to decrement it. For each flow, we verify if the instantaneous throughput and the number of packets are higher than the superior threshold or are lower than the inferior threshold. If any of them is higher than the superior threshold, it means that the number of bytes or the throughput is higher than the acceptable, so the $D$ function is used to decrement the interval between requests. Additionally, if the throughput or number of bytes is lower than the inferior threshold, it means that the rate of one of them is smaller than the acceptable, so the $I$ function is used to increment the request interval. Figure 4.8 shows how this mechanism work.

![Figure 4.8 Scheduling module threshold.](image)

The goal of using the thresholds is to maintain the rate which we request statistics in an acceptable margin. The superior and inferior thresholds are configurable via REST or configuration file, and it is vital that the network manager set them according to the network capabilities. If one does not configure rightly these parameters, it is possible that when the traffic of the monitored flows increases, the scheduler may increase the request time as well, causing issues to the network.

As presented in chapter 2 the binomial congestion control algorithms behavior depends on the parameters and constant values. As these values are of great importance and a manager may want to change them according to the network behavior, we allow them to be manually
configured via REST or configuration file.

Our algorithm to schedule requests works as described below by the flowchart in figure 4.9.

Looking at figure 4.9 one may note that there are two limits for the frequency values: one being the maximum and the other being the minimum. These values are important because of the spikes in the network, packet loss, and other factors that may make the calculated incoming rate of bytes and packets increase or decrease drastically.
New statistics arrives

Are the Bytes or Packets transported higher than the upper?

YES

Calculate decrease value

NO

Are the Bytes or Packets transported lower than the lower?

YES

Calculate increase value

NO

Is the new frequency higher than the maximum limit?

YES

Use the maximum frequency limit

NO

Is the new frequency lower than the minimum limit?

YES

Use calculated frequency

NO

Use same frequency

Update statistics request frequency

Figure 4.9 Description of the scheduling algorithm.
4.2.3 Controller Module

The Controller module manages the Storage and the Scheduler module. It selects the data to be stored from the Scheduler and sends to the Storage module. Further, it handles new OF devices connected to the network and SHAMon Proxies as well. Finally, it works as a Business Logic Layer (BLL) layer between the Storage and Scheduler module, and the REST module in the SHAMon Controller, and as a business layer between the Scheduler module and the Communication module in the SHAMon Proxy.

We select the information to store in a simple and straightforward way. All flow’s statistics have a timestamp tag associated with it, and when the statistics get old enough we store them in the database. Additionally, the timestamp allows the reconstruction of the traffic. The statistics are from the SHAMon Controller or SHAMon Proxy, and they are maintained at memory level to allow fast access.

Further, this module handles new connections from OF devices. When the SDN Controller detects a new device, it alerts our Controller module so that it can do the initial procedure. We begin the procedure by checking if the device is assigned to the SHAMon Proxy. If it is assigned and it is being handled by it, we just need to listen to statistics, else we assign the Scheduler Module to monitor the device and its flows. Figure 4.10 explains the behavior just described.

![Figure 4.10 Controller flow for new devices connecting.](image)

Besides devices, the Controller module listens to new SHAMon Proxies as well. It is possible to new proxies to be connected to the network as needed changing the arrangement, so we need to deal with it as well. The first thing the Controller module does is to check if any of the OF devices assigned to the Proxy is managed by the SHAMon Controller. If there are, the Scheduling Module is signaled to stop overseeing the devices, and the information will be provided by the SHAMon Proxy. Figure 4.11 presents this procedure.
Figure 4.11 Controller flow for new SHAMon Proxy connecting.

When the list of devices monitored by the SHAMon Proxy change, the proxy itself will communicate with the SHAMon Controller alerting the update. If a device is removed from the proxy list of devices, it is automatically assigned to the SHAMon Controller when the update arrives. Additionally, if any device that is being monitored by the Controller is assigned to the Proxy, the Controller will stop monitoring it and it will just receive statistics from the proxy. Figure 4.12 shows the workflow of this procedure.

The REST and Communication modules use the Controller module as the link interface between the lower modules, as a BLL. This is important so that high-level business logic can be used by the Communication and REST module easing the access to the information by any user or application.
4.2. SHAMON ARCHITECTURE AND FUNCTIONS

Figure 4.12 Controller flow for SHAMon Proxy updates.
4.2.4 REST Module

The main function of this module is allowing users or applications to access the collected information. We developed a REST API so that all the statistics collected by the SHAMon Controller and Proxies were available. This allows that the information can be accessed using any programming language.

To request statistics a Request object must be sent to the SHAMon Controller. This object has several tags that help filter, and get the information at the level needed. The message format is as described below:

```json
{
    "Request" : {
        "AggregateLevel" : ["shamonproxy", "datapath", "port", "flow"],
        "Filter" : {
            "Flowid" : [...] ,
            "FlowMatch" : {
                "MacAddressSrc" : ". . ." ,
                "MacAddressDst" : ". . ." ,
                "IpSrc" : "..." ,
                "IpDst" : "..." ,
                "PortSrc" : "..." ,
                "PortDst" : "..."
            },
            "Datapaths" : [...] ,
            "ShamonProxyIds" : [...] 
        }
    }
}
```

- **AggregateLevel**: can be used to choose which level to aggregate the statistics. The values can vary from `shamonproxy` for grouping all the flows being monitored by instances of SHAMon Proxies, `datapath` for grouping all flows by switches separated by table, `port` to group all flows monitored by output port, and `flow` to display all flows;

- **Filter**: can be used to filter the statistics. It is possible to filter by `flowid`, Media Access Control (MAC) address, Internet Protocol (IP) and port number from source and destination using the `FlowMatch` and its fields. `Datapaths` can be used by passing one or a list of datapathids to filter which switch or switches the statistics are wanted. Finally, `ShamonProxyId` can be used to filter which SHAMon Proxy the statistics are wanted and can be done by passing one value or a list as well;
4.2. SHAMON ARCHITECTURE AND FUNCTIONS

- **Frequency**: can be used to tell the number of times the response from this request will be sent. It can be a *one-time* request, or a time interval in milliseconds, seconds or minutes.

The Request object is detailed using JavaScript Object Notation (JSON). This object is registered using SHAMon REST API, and after the registration being successful SHAMon Controller will send the stored statistics that were requested. The object returned is a JSON Response object showed below.

The Response object comes with some of the fields transmitted in the Request object. These fields are *AggregateLevel*, *Filter*, *Datapaths*, and *ShamonProxyIds*. These fields are sent to allow the receiver to validate, or use them on a more specific request. The Response will vary according to the aggregate level requested, so it can be in four different formats. All statistical values presented in the Response object are the results of the difference between the newest value collected and the old ones. The *Statistics* field is the one with the information requested and is divided as follows:

1. **Shamonproxy Aggregate Level**:

```json
{
    "Response" : {
        "AggregateLevel" : "...",
        "Filter" : {
            "Flowid" : [...],
            "FlowMatch" : {
                "MacAddressSrc" : "...",
                "MacAddressDst" : "...",
                "IpSrc" : "...",
                "IpDst" : "...",
                "PortSrc" : "...",
                "PortDst" : "..."
            },
            "Datapaths" : [...],
            "ShamonProxyIds" : [...]  
        }
    },
    "Statistics" : {
        "shamonproxies" : {
            "<shamonproxyid>" : {
                "flowids" : [...],
                "packetsIn" : "...",
                "packetsOut" : "...",
                "bytesIn" : "...",
                "bytesOut" : "...",
                "datapaths" : {
                    "<datapathid>" : {
                        "flowids" : [...]
                    }
                }
            }
        }
    }
}
```
4.2. SHAMON ARCHITECTURE AND FUNCTIONS

- **shamonproxies**: if *shamonproxy* was chosen as the aggregate level the *shamonproxies* tag will be present in the response. The statistics in this tag will be organized by *shamonproxyids*;

- **shamonproxyid**: has all the latest aggregated information of the flows monitored by the SHAMon Proxy. The *flowids* tag is a list with all flows being monitored. The *packetsIn* tag is the number of packets that arrived and the *packetsOut* is the number of the packets sent by the switches monitored by the proxy. *BytesIn* tag shows the sum of bytes from the packets that arrived and *bytesOut* tag shows the sum of bytes from the packets sent by the switches monitored by the proxy. The tag *datapaths* separates the information of the proxy by switches presenting all the aforementioned information according to the switch instead of the proxy.

2. **Datapath Aggregate Level**:

```json
{
  "Response" : {
    "AggregateLevel" : "...",
    "Filter" : {
      "Flowid" : [...],
      "FlowMatch" : {
        "MacAddressSrc" : "...",
        "MacAddressDst" : "...",
        "IpSrc" : "...",
        "IpDst" : "...",
        "PortSrc" : "...",
        "PortDst" : "...
      },
      "Datapaths" : [...],
      "ShamonProxyIds" : [...]
  }
}
```
4.2. SHAMON ARCHITECTURE AND FUNCTIONS

- **datapaths**: is a list of all switches monitored organized by the switch’s datapathids;
- **datapathid**: has the switch’s latest collected statistics. It has `packetsIn`, `packetsOut`, `bytesIn`, and `bytesOut` fields, and they do the same calculation as the `shamonproxies` aggregation level, but related to the switch;
4.2. SHAMON ARCHITECTURE AND FUNCTIONS

- **tables**: lists all the flow tables that are being used by the switch. They are separated by *tableid*;
- **tableid**: lists all the flow rules that are in the table;
- **flowid**: brings each flow rule by its identification and relevant information. It has *packetsIn*, *packetsOut*, *bytesIn*, and *bytesOut* that have the statistics of packets and bytes of the flow. *portIn* is the port where the packets of the flow are arriving and *portOut* is the port where the packets of the flow are being sent;
- **FlowMatch**: exhibits the match information necessary to classify a packet to this flow rule. It may have the source and destination MAC address, source and destination IP address, and source and destination port if present in the rule.

3. **Port Aggregation Level**

```
{
    "Response" : {
        "AggregateLevel" : "...",
        "Filter" : {
            "Flowid" : [...],
            "FlowMatch" : {
                "MacAddressSrc" : "...",
                "MacAddressDst" : "...",
                "IpSrc" : "...",
                "IpDst" : "...",
                "PortSrc" : "...",
                "PortDst" : "..."
            },
            "Datapaths" : [...],
            "ShamonProxyIds" : [...]}
    }
    "Statistics" : {
        "ports" : {
            "datapaths" : {
                "<datapathid>" : {
                    "<portNumber>" : {
                        "flows" : {
                            "<flowid>" : {
                                "packetsIn" : "...",
                                "packetsOut" : "...",
                                "bytesIn" : "...",
                                "bytesOut" : "...",
                                "tableid" : "...",
                                "FlowMatch" : {
                                    "MacAddressSrc" : "...",
                                    "MacAddressDst" : "..."}
                        }
                    }
                }
            }
        }
    }
```
4.2. SHAMON ARCHITECTURE AND FUNCTIONS

- **ports**: is a list of switches aggregating the flows by output port;
- **datapaths**: list with switches identified by *datapathid*;
- ** datapathid**: object with all active ports of the switch;
- ** portnumber**: port index with a list of all flows being transmitted;
- **flows**: list with flow objects;
- ** flowid**: flow identifier with packets in and out, bytes in and out, table identifier, and *flowmatch* object;
- ** flowmatch**: flow rule match attributes object with source and destination IP address, source and destination MAC address, source and destination port if present in the rule.

4. Flow Aggregation Level

```json
{
  "Response" : {
    "AggregateLevel" : "...",
    "Filter" : {
      "Flowid" : [...],
      "FlowMatch" : {
        "MacAddressSrc" : "...",
        "MacAddressDst" : "...",
        "IpSrc" : "...",
        "IpDst" : "...",
        "PortSrc" : "...",
        "PortDst" : "..."
      }
    }
  }
}
```
4.2. SHAMON ARCHITECTURE AND FUNCTIONS

- `flows`: a list with all the flows being monitored;
- `flowid`: flow object identified uniquely by its id. It has the packets in and out information, bytes in and out, port in and out, table identifier, datapath identifier, and `flowmatch` object;
- `flowmatch`: flow rule match attributes object with source and destination IP address, source and destination MAC address, source and destination port if present in the rule.

4.2.5 Communication Module

The Communication module establishes a connection between the SHAMon Controller and SHAMon Proxy. Besides this main task, it is possible to configure monitoring parameters, remove or add monitored devices to the proxies. In addition, it is this module that summarizes
the collected information, creating a REST Message with the latest statistics collected, to send it to the SHAMon Controller.

When the SHAMon Proxy starts, it initiates the connection with all the OF devices it will monitor. This initial connection is important because the proxy will send to the SHAMon Controller the list of devices it is currently monitoring and if for some reason the proxy could not start monitoring the device, the SHAMon Controller will monitoring it.

Furthermore, the Proxy will be sending the newest collected statistics in a configured time frequency. The Controller manages the time-frequency, and it has no relation to the calculated schedule time of the monitored flows. The Proxy will keep monitoring each flow individually, collecting statistics and when the time comes to send them it will get all of them, package, compress and send them to the Controller.

If a device is removed or added to be monitored, an update message will be sent to the SHAMon Controller to update the lists of devices monitored, executing the procedure explained in Figure 4.12.
5

EVALUATION

To evaluate our solution we did a couple of experiments. In our first experiment, we wanted to show the behavior of our scheduling algorithm. Our second experiment compares our solution with two others, highlighting our achievements. The rest of this chapter is organized as follows. Section 5.1 describes the methodology of the experiments. Section 5.2 describes the experiments, their purpose, what they want to demonstrate, and their results.

5.1 Methodology

To execute both experiments, we used four Lanners machines named, Lanner 1, Lanner 2, Lanner 3, and Lanner 4. Each Lanner handles a part of the logical topology as present in figure 5.1. To simulate the OpenFlow switches we used OpenVSwitch (FOUNDATION, 2016) to act as bridges over physical interfaces. To create each host we used KVM (KVM, 2016).

Lanner 1 had the switch SW0 and the OpenFlow Controller. We used two physical network interfaces from it as ports for SW0. Lanner 2 had two switches, SW1 and SW2, each switch using three physical interfaces resulting in six physical interfaces being used. Lanner 3 and Lanner 4 had two switches each, SW3 and SW4 in Lanner 3, SW5 and SW6 in Lanner 4, each switch using one physical interface. We created four hosts in Lanner 3, and another four in Lanner 4 using KVM. Each host used Ubuntu Server 16.04 amd64 as its operational system. We used Iperf3 (Energy Sciences Network (ESnet), 2016) tool to simulate the traffic between hosts. Figure 5.2 shows the network physical configuration of the experiment, presenting each network interface used by each Lanner.

Each host had a virtual interface created so that it could connect to the virtual switch. Each switch received a static IP, as well as each host. Table 5.2 describes the network configuration of the switches and its interfaces. Table 5.1 lists the IP addresses of the switches.

The OpenFlow Controller communication was done out-of-band. No routing application was used so we created all the flow rules necessary for the experiments to be executed manually. We used the command line Wireshark application Tshark (COMBS, 2017) to capture the traffic between the Controller and the switches. To process the collected data, we used NumPy (NUMFOCUS, 2017), and to create the plots we used Plotly (PLOTLY, 2017). All the experiments
Figure 5.1 Experiment network topology

were executed 31 times with all data being collected, with an interval of 90 seconds between each one.

Table 5.1 Switches’ IP Addresses

<table>
<thead>
<tr>
<th>Switch</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW0</td>
<td>10.0.25.240</td>
</tr>
<tr>
<td>SW1</td>
<td>10.0.25.241</td>
</tr>
<tr>
<td>SW2</td>
<td>10.0.25.242</td>
</tr>
<tr>
<td>SW3</td>
<td>10.0.25.243</td>
</tr>
<tr>
<td>SW4</td>
<td>10.0.25.244</td>
</tr>
<tr>
<td>SW5</td>
<td>10.0.25.245</td>
</tr>
<tr>
<td>SW6</td>
<td>10.0.25.246</td>
</tr>
</tbody>
</table>

As explained above, during the experiment network traffic was simulated using Iperf3. We named the simulated traffic model, and during the explanation and results of the experiments, the model traffic will be compared with what our and others solutions measured.
Figure 5.2  Network physical configuration diagram
### Table 5.2 Distribution of switches, interfaces, and ports

<table>
<thead>
<tr>
<th>Switch</th>
<th>Interface</th>
<th>MAC Address</th>
<th>Virtual</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW0</td>
<td>p130p1</td>
<td>00:90:0b:2a:2c:78</td>
<td>No</td>
<td>6</td>
</tr>
<tr>
<td>SW0</td>
<td>p131p1</td>
<td>00:90:0b:2a:2c:79</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>SW1</td>
<td>p130p1</td>
<td>00:90:0b:28:50:88</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>SW1</td>
<td>p132p1</td>
<td>00:90:0b:28:50:8a</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>SW1</td>
<td>p133p1</td>
<td>00:90:0b:28:50:8b</td>
<td>No</td>
<td>3</td>
</tr>
<tr>
<td>SW2</td>
<td>p131p1</td>
<td>00:90:0b:28:50:89</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>SW2</td>
<td>p134p1</td>
<td>00:90:0b:28:50:8c</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>SW2</td>
<td>p135p1</td>
<td>00:90:0b:28:50:8d</td>
<td>No</td>
<td>3</td>
</tr>
<tr>
<td>SW3</td>
<td>p132p1</td>
<td>00:90:0b:2a:2c:9a</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>SW3</td>
<td>veth0-h1</td>
<td>fe:00:00:00:00:01</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>SW3</td>
<td>veth0-h2</td>
<td>fe:00:00:00:00:02</td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td>SW4</td>
<td>p133p1</td>
<td>00:90:0b:2a:2c:9b</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>SW4</td>
<td>veth0-h3</td>
<td>fe:00:00:00:00:03</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>SW4</td>
<td>veth0-h4</td>
<td>fe:00:00:00:00:04</td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td>SW5</td>
<td>p134p1</td>
<td>00:90:0b:2a:2c:64</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>SW5</td>
<td>veth0-h5</td>
<td>fe:00:00:00:00:05</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>SW5</td>
<td>veth0-h6</td>
<td>fe:00:00:00:00:06</td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td>SW6</td>
<td>p135p1</td>
<td>00:90:0b:2a:2c:65</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>SW6</td>
<td>veth0-h7</td>
<td>fe:00:00:00:00:07</td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td>SW6</td>
<td>veth0-h8</td>
<td>fe:00:00:00:00:08</td>
<td>Yes</td>
<td>2</td>
</tr>
</tbody>
</table>
5.2 Experiments and Results

5.2.1 Scheduler Algorithm Experiment

To validate our scheduler algorithm we designed an experiment that analyzes the behavior of the binomial algorithm and its control of the monitoring requests time.

As explained, the binomial algorithm rules generalize the class of all linear control algorithms. For \( k = 0, l = 1 \), we get AIMD; for \( k = -1, l = 1 \), we get MIMD. For \( k = -1, l = 0 \) we get MIAD; and for \( k = 0, l = 0 \) we get AIAD. These four classes determine the behavior of the increment and decrement function, therefore how much will be the amount of increment or decrement that the frequency time between each request for statistics will change.

Additionally, in the binomial algorithms, there are two constants, \( \alpha \) and \( \beta \), which are the values that will be exponentiated by \( k \) and \( l \), respectively, to establish the amount to be increase or decrease of the frequency time. In TCP, \( \alpha \) and \( \beta \) values usually are functions of the current TCP window value. Additionally, each class of congestion control algorithm define some rules for the \( \alpha \) and \( \beta \) values so that the algorithm can be TCP-friendly.

Likewise, the upper and lower thresholds of the byte and packet variation are parameters that directly change the request time. As already explained in section 4.2.2, when the number of packets or throughput cross the lower threshold the request time will be increased and when one crosses the upper threshold it will be decreased.

Further, the minimum request time configured for the Scheduler is important because when a flow throughput becomes intense the request frequency can converge to it depending on the lower threshold value. The opposite is true as well, as the flow becomes less and less active, the request frequency will converge to the maximum request time depending on the upper threshold value.

The behavior of the Scheduler is important in our monitoring tool because it will manage the start of our requests for statistics, the frequency, and how it will decrease or increase the request time. These are important factors that directly affects the precision of our solution, hence the need to present the effects of the variation of these parameters. Table 5.3 summarize all the parameters, its importance in our Scheduler.

To accomplish the evaluation, we used a 3-level tree topology scenario as presented in figure 5.1. We generated User Datagram Protocol (UDP) flows between hosts for a total duration of 60 seconds. The figure 5.3 timing diagram shows the start of the flows, the throughput, and the end time of each flow. This setup is the same that the (CHOWDHURY et al., 2014) used in their experiments to demonstrate the capabilities of PayLess. For each traffic simulation between hosts, a couple of flow rules were created and placed in each switch, summing up to eight flow rules per switch.

In addition, to understand how the variations of some of table 5.3 parameters may change the behavior of our Scheduler algorithm we decided to vary some of them. We selected \( k, l, \) upper and lower packet threshold, and upper and lower byte threshold.
5.2. EXPERIMENTS AND RESULTS

Table 5.3 Parameters summarization

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Importance</th>
<th>Used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>Exponent of the alfa constant</td>
<td>Find out increment of the request time</td>
</tr>
<tr>
<td>L</td>
<td>Exponent of the beta constant</td>
<td>Find out decrement of the request time</td>
</tr>
<tr>
<td>α</td>
<td>Base value to be exponentiated</td>
<td>Find out increment of the request time</td>
</tr>
<tr>
<td>β</td>
<td>Base value to be exponentiated</td>
<td>Find out decrement of the request time</td>
</tr>
<tr>
<td>Minimum Request Time</td>
<td>Minimum allowed value to the request time</td>
<td>As minimum value allowed</td>
</tr>
<tr>
<td>Maximum Request Time</td>
<td>Maximum allowed value to the request time</td>
<td>As maximum value allowed</td>
</tr>
<tr>
<td>Upper Packet Threshold</td>
<td>Maximum allowed value of packet variation</td>
<td>Find out if request time must be decreased</td>
</tr>
<tr>
<td>Upper Byte Threshold</td>
<td>Maximum allowed value of byte variation</td>
<td>Find out if request time must be decreased</td>
</tr>
<tr>
<td>Lower Packet Threshold</td>
<td>Minimum allowed value of packet variation</td>
<td>Find out if request time must be increased</td>
</tr>
<tr>
<td>Lower Byte Threshold</td>
<td>Minimum allowed value of byte variation</td>
<td>Find out if request time must be increased</td>
</tr>
</tbody>
</table>

Figure 5.3 Timeline diagram of the experiment

Before we choose the $k$ and $l$ values to experiment, we chose $\alpha$ and $\beta$ because their values affect the request time too. Looking at the default values for $\alpha$ and $\beta$ for each class of the binomial algorithm we decided to use 0.5 for each one because they are the most friendly values and the effect of them would be minimum.

Based on $\alpha$ and $\beta$ values, and the behavior explanation of the binomial algorithms from (BANSAL; BALAKRISHNAN, 2001) about the $k$ and $l$ values we decided to vary $k$ values by -0.5, -0.75, and -1.0, $l$ values by 0.5, 0.75, and 1.0, and to do all combinations between the two parameters. The minimum and maximum polling interval for our scheduling algorithm were set to 500ms and 5s, respectively.

We set the Upper and lower packet threshold to 8 and 4 packets per request, and upper and lower byte threshold to 400 and 100 bytes per request respectively. We chose these values because we wanted the algorithm to be sensitive to flows changes, increasing and decreasing the
request time as much as possible, allowing the effect of the $k$ and $l$ variations to be observed.

We measured the throughput per second of the network, the time request variation for each flow, and the number of requests for statistics per second. We did all measurements using the switch SW0. We used the number of requests for statistics per second to measure the overhead generated by our solution. We calculated the root-mean-square error (RMSE) between the measured throughput and the real (model) throughput to estimate our solution monitoring error.

The second part of this experiment, we varied the upper and lower thresholds. The lower packet threshold varied from 4, 32, and 64 per request, and the upper packet threshold varied from 8, 64, 128 per request. The lower byte threshold varied from 300, 1600, and 3200 per request while the upper byte threshold varied from 600, 2200, and 4000 per request.

The threshold values were chosen in order to validate how the Scheduler algorithm would behave with the three types of flows from the experiment. In the experiment, we have H1 to H8 long and little bandwidth consuming flow, H2 to H7 short, with high burst bandwidth usage, and H3 to H6 a medium flow with a medium bandwidth usage. Each threshold value was singled to fit best each type of flow.

The minimum and maximum polling interval for our scheduling algorithm were set to 500ms and 5s, respectively. Finally, $\alpha$ and $\beta$ values were set to 0.5, the same as the previous, and $k$ values were set to the -0.5 and $l$ to 0.5 because they are the less drastic combination of values.

We measured the throughput per second of the network, the time request variation for each flow, and the number of requests for statistics per second. We did all measurements using the switch SW0. We used the number of requests for statistics per second to measure the overhead generated by our solution. We calculated the RMSE between the measured throughput and the real (model) throughput to estimate our solution monitoring error.

We chose these parameters because the changes that will happen with each one of them, as well as how they will impact the monitoring are important to be studied. Additionally, they allow a network administrator to know exactly what needs to be changed so that our solution can fit in their network. Furthermore, they directly improve the accuracy and the reaction granting a timely monitoring. Table 5.4 summarize the parameters and the values that they will vary in this experiment.

The results of the $k$ and $l$ variation experiment enlightened how increment and decrement of the request frequency time affect our Scheduler algorithm. According to the formula of the binomial algorithms, a fixed positive value of $\alpha$ with higher negative values of $k$ make the amount to be incremented higher. Higher positive values for $l$, with a fixed positive value of $\beta$, make the amount to be decremented higher.

If it varies greatly, the measurement can have lots of spikes depending on how much incrementing and decrementing happens. If it varies little the measurement will ignore fluctuations of the traffic and may generate peaks as well. Clearly, the type of the traffic will dictate the needed values.
Table 5.4 Parameters variation for the Scheduler algorithm experiment

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>-0.5 -0.75 -1</td>
</tr>
<tr>
<td>l</td>
<td>0.5 0.75 1</td>
</tr>
<tr>
<td>Upper Packet Threshold</td>
<td>8, 64, 128 packets</td>
</tr>
<tr>
<td>Upper Byte Threshold</td>
<td>600, 2200, 4000 bytes</td>
</tr>
<tr>
<td>Lower Packet Threshold</td>
<td>4, 32, 64 packets</td>
</tr>
<tr>
<td>Lower Byte Threshold</td>
<td>300, 1600, 3200 bytes</td>
</tr>
</tbody>
</table>

Further, the traffic in our experiment is constant with high fluctuations, and because of that a higher \( l \) with a medium \( k \) value may be the best-suited values. To check this assertion, the error measurement is one of the best indicators. Figure 5.4 shows the error measurement for each \( k \) and \( l \) variation with the network overhead generated.

Figure 5.4 validates the assertion made above, showing that with a \( l \) with higher values, we had the lowest measurement error, however, the network overhead generated by them was the highest. The set of measurements with the lowest \( k \) values were the ones with the lowest network overhead, but the error was high. One might note that in all of our measurement the error was not higher than 26 Mbps and our worst network overhead generated was 18 requests per second.

This information is important because there were two flow rules per host that communicated in the experiment, which gives a network overhead measurement of 3 requests per second per flow in the worst case.

To present the monitoring and its errors in a practical way, and to understand what the measurement error in the network monitoring means, and the effects of the \( k \) and \( l \) variation, we plotted the utilization measurement of each variation with the model traffic in figures 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13.

Figure 5.10 is an example of an \( l \) value above of what is required, with zero measurements at some points because of the tiny request timed which makes this mountain outline like measurement. Additionally, figure 5.11 is an example of a \( k \) value below than what is needed, making the measurement ignores the traffic end’s between hosts of the experiment.
Figure 5.4: RMSE and network overhead for k and L variations.
5.2. EXPERIMENTS AND RESULTS

Utilization Measurement (K=-0.5, L=0.5)

Figure 5.5  Network Utilization Measurement for $k = -0.5$ and $L = 0.5$

Utilization Measurement (K=-0.5, L=0.75)

Figure 5.6  Network Utilization Measurement for $k = -0.5$ and $L = 0.75$
5.2. EXPERIMENTS AND RESULTS

Utilization Measurement (K=0.5, L=1.0)

![Graph of Utilization Measurement for K=0.5, L=1.0]

**Figure 5.7** Network Utilization Measurement for \( k = -0.5 \) and \( L = 1 \)

Utilization Measurement (K=0.75, L=0.5)

![Graph of Utilization Measurement for K=0.75, L=0.5]

**Figure 5.8** Network Utilization Measurement for \( k = -0.75 \) and \( L = 0.5 \)
5.2. EXPERIMENTS AND RESULTS

**Utilization Measurement (K=-0.75, L=0.75)**

![Graph showing Throughput (Mbps) over time for k = -0.75 and L = 0.75]

*Figure 5.9* Network Utilization Measurement for $k = -0.75$ and $L = 0.75$

**Utilization Measurement (K=-0.75, L=1.0)**

![Graph showing Throughput (Mbps) over time for k = -0.75 and L = 1.0]

*Figure 5.10* Network Utilization Measurement for $k = -0.75$ and $L = 1$
5.2. EXPERIMENTS AND RESULTS

Utilization Measurement (K=-1.0, L=0.5)

Figure 5.11  Network Utilization Measurement for $k=-1$ and $L=0.5$

Utilization Measurement (K=-1.0, L=0.75)

Figure 5.12  Network Utilization Measurement for $k=-1$ and $L=0.75$
5.2. EXPERIMENTS AND RESULTS

The higher \( l \) values did not affect the H1 and H8 polling time variance, even with a lower \( k \) value, because the traffic is constant and has a low bandwidth usage. That might be a problem if the bandwidth usage was below the byte or packet lower threshold, which would make the time fluctuate. For the traffic between H2 and H7, higher \( l \) values with small \( k \) values suit it the best, as there are gaps of no traffic. High values for \( l \) and lower values for \( k \) would not be so bad for the measurement monitoring, however, the variance of the polling time would skyrocket. Lastly, for the traffic between H3 and H6, a high or moderate \( l \) value and a small \( k \) value work the best, because it is a moderate stream that does not last all the experiment and is in the middle of it.

Table 5.5 contains the mean, standard deviation, and variance for the traffic between each host for all \( k \) and \( l \) variation. The values have confirmed our expectations for the behavior of the variance of the polling time. It is important to take in consideration that high variance alone is not a bad thing, it will depend on the flow traffic.

Figure 5.14 presents a box plot of the polling times for each \( k \) and \( l \) variation. From it, one might note that the \( k \) and \( l \) values induce where the polling time will orbit. Higher values for \( l \) and small values for \( k \) makes the polling time be near the minimal polling time, lower \( k \) values and small \( l \) values makes the polling time be near the maximum polling time, and moderate values make it states in the middle of minimum and maximum polling time. Another observation is that moderate values lead to common behavior for the polling time.
### Table 5.5 Polling time information for $k$ and $L$ variation

<table>
<thead>
<tr>
<th>$K$</th>
<th>$L$</th>
<th>Flow</th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.5</td>
<td>0.5</td>
<td>H1 &lt;-&gt; H8</td>
<td>1892.1</td>
<td>63.2401</td>
<td>3999.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2 &lt;-&gt; H7</td>
<td>2630.99</td>
<td>70.7562</td>
<td>5006.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3 &lt;-&gt; H6</td>
<td>2840.28</td>
<td>54.5599</td>
<td>2976.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H1 &lt;-&gt; H8</td>
<td>1854.86</td>
<td>59.529</td>
<td>3543.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2 &lt;-&gt; H7</td>
<td>2480.39</td>
<td>54.0137</td>
<td>2917.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3 &lt;-&gt; H6</td>
<td>2764.03</td>
<td>25.7771</td>
<td>664.459</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>H1 &lt;-&gt; H8</td>
<td>976.38</td>
<td>10.3171</td>
<td>106.442</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2 &lt;-&gt; H7</td>
<td>1388.09</td>
<td>37.3283</td>
<td>1393.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3 &lt;-&gt; H6</td>
<td>1954.39</td>
<td>9.24467</td>
<td>85.4639</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H1 &lt;-&gt; H8</td>
<td>1935.69</td>
<td>66.1966</td>
<td>4381.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2 &lt;-&gt; H7</td>
<td>2703.79</td>
<td>44.7068</td>
<td>1998.69</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3 &lt;-&gt; H6</td>
<td>3028.43</td>
<td>29.7717</td>
<td>886.353</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>H1 &lt;-&gt; H8</td>
<td>1946.06</td>
<td>10.3171</td>
<td>8148.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2 &lt;-&gt; H7</td>
<td>2552.3</td>
<td>40.399</td>
<td>1632.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3 &lt;-&gt; H6</td>
<td>2904.34</td>
<td>36.9722</td>
<td>1366.95</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>H1 &lt;-&gt; H8</td>
<td>3268.07</td>
<td>5.50378</td>
<td>30.2916</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2 &lt;-&gt; H7</td>
<td>4276.48</td>
<td>21.4334</td>
<td>459.391</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3 &lt;-&gt; H6</td>
<td>4273.81</td>
<td>19.4642</td>
<td>378.856</td>
</tr>
<tr>
<td></td>
<td>0.75</td>
<td>H1 &lt;-&gt; H8</td>
<td>2712.44</td>
<td>105.524</td>
<td>11135.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2 &lt;-&gt; H7</td>
<td>4006.04</td>
<td>23.6454</td>
<td>559.105</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3 &lt;-&gt; H6</td>
<td>4254.13</td>
<td>7.3964</td>
<td>54.7068</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>H1 &lt;-&gt; H8</td>
<td>1188.67</td>
<td>23.2517</td>
<td>540.643</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H2 &lt;-&gt; H7</td>
<td>2144.48</td>
<td>138.284</td>
<td>19122.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>H3 &lt;-&gt; H6</td>
<td>3430.35</td>
<td>135.681</td>
<td>18409.4</td>
</tr>
</tbody>
</table>
Figure 5.14 Boxplot of polling time variation for $k$ and $L$ variations
The upper and lower threshold variation was designed to vary the reactions limits of our algorithm so that we could analyze the effects it may cause. Both byte and packet variation measurement have a couple of thresholds for lower and higher values, as already explained. As we have different types of traffic crossing the network in the experiment, different thresholds affect them uniquely. We named the bytes thresholds as Lower Byte Threshold (LBT) and Upper Byte Threshold (UBT), and for the packets threshold, Lower Packet Threshold (LPT) and Upper Packet Threshold (UPT).

It is important to understand how the thresholds work in our solution because the duration of the polling time will only be modified if any of the rates cross the thresholds, and, naturally, the polling time will converge to stay between the thresholds. Because of that, and the traffic nature of the flows of the experiment, we have three types of values for the thresholds, the first one accepts small values with a narrow variation, the second accepts medium values with a broader variation, and the third one accepts big values with the broadest variation.

In the experiment, as the threshold values progressed by increasing the values and broadening the distance between them, the error measurement result showed that this value progression does not influence the bandwidth utilization measurement. The error measurement was almost equal for every combination. The network overhead had minimal variation between the combinations, with the number of requests per second going not over 10. Figure 5.15 shows how the measurement error behaved in each set of values. This information seems valid due to the fact that the thresholds changes modify how our solution reacts to network traffic fluctuations, not how much it will react.
5.2. EXPERIMENTS AND RESULTS

Figure 5.15 Error and overhead measurement for the threshold experiment
5.2. EXPERIMENTS AND RESULTS

However, the polling time behaved noticeably different from the \( k \) and \( l \) experiment. Figure 5.16 presents the box plot of the polling time variation for this experiment separated by flow and aggregated by threshold combinations and table 5.6 presents the mean, standard deviation, and variance. The mean of the polling time in all combinations was almost equal, which explains why the network overhead is mostly the same for every combination.

The variance too behaved differently, varying greatly from \( k \) and \( l \) experiment. Additionally, the variance values were different between each combination of threshold values as well. High variance in a flow measurement, alone, is not bad, it will depend on the nature of the traffic. In general, high variance means that the measurement is sensitive to the variations rates of the flow and react easily. So, if in this experiment the \( k \) and \( l \) values were higher, the variance would be higher.

The flows of H1 and H8 were the ones to have the highest variance. This can be explained by the nature of the traffic, that has a low throughput and a gap of 5 seconds. Even with the high variance, most of the polling time values ranged from 1900 milliseconds to 2100 milliseconds, which is not a big range. In addition, the average of the values increased with the increase of the threshold values, as well as the concentration of polling times moved from the superior quartile to the inferior, which made us conclude that the zone of convergence is from 1900 to 2100 milliseconds.

H2 and H7 did not have a high variance in comparison with H1 and H8 flows, even with all the gaps with no traffic. The justification for this behavior is that the burst of data compensated the gaps, as well as the moderate values of \( k \) and \( l \). The byte threshold of 3200 and 4000 combined with the packet threshold of 32, 64, generated the traffic highest variance. On the other hand, the byte threshold with 1600 and 2200 and the packet threshold with 32, 64, generated the lowest variance. These made us conclude that the 3200, 4000 bytes as thresholds are higher than what is necessary for the flow, which made that the polling values fluctuate. Additionally, we reasoned that 1600 and 2200 are the ideal threshold values for the flow.

H3 and H6 traffic, defined as a high burst during a small time of the experiment, did not adjust well to the lower threshold values, resulting in high variance values. The only combination that worked was 3200 and 4000 as the byte thresholds with 64 and 128 as the packet thresholds. However it did not affect the polling time, maintain an equal mean for each combination.
5.2. EXPERIMENTS AND RESULTS

Figure 5.16: Boxplot of time frequency variation for the threshold variations.
### Table 5.6 Polling time information for threshold variation

<table>
<thead>
<tr>
<th>LPT</th>
<th>UPT</th>
<th>LBT</th>
<th>UBT</th>
<th>Flow</th>
<th>Mean</th>
<th>StD</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>300</td>
<td>600</td>
<td>H1 ↔H8</td>
<td>1891.92</td>
<td>84.5561</td>
<td>7149.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2629.38</td>
<td>60.8489</td>
<td>3702.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2838.58</td>
<td>76.7224</td>
<td>5886.32</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1600</td>
<td>2200</td>
<td>H1 ↔H8</td>
<td>1931.62</td>
<td>104.128</td>
<td>10842.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2644.07</td>
<td>53.7559</td>
<td>2889.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2838.58</td>
<td>76.7224</td>
<td>5886.32</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3200</td>
<td>4000</td>
<td>H1 ↔H8</td>
<td>1935.26</td>
<td>107.366</td>
<td>11527.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2642.64</td>
<td>54.6176</td>
<td>2983.08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2841.85</td>
<td>69.2612</td>
<td>4797.11</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>300</td>
<td>600</td>
<td>H1 ↔H8</td>
<td>1927.82</td>
<td>84.7229</td>
<td>7177.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2652.00</td>
<td>48.9228</td>
<td>2393.44</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2842.47</td>
<td>78.4452</td>
<td>6153.66</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>1600</td>
<td>2200</td>
<td>H1 ↔H8</td>
<td>1957.14</td>
<td>105.192</td>
<td>11065.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2616.68</td>
<td>73.3435</td>
<td>5379.27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2849.70</td>
<td>78.0818</td>
<td>6096.76</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>3200</td>
<td>4000</td>
<td>H1 ↔H8</td>
<td>1973.37</td>
<td>100.963</td>
<td>10193.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2647.67</td>
<td>54.5495</td>
<td>2975.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2847.87</td>
<td>67.4982</td>
<td>4556.01</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>300</td>
<td>600</td>
<td>H1 ↔H8</td>
<td>1985.69</td>
<td>87.5003</td>
<td>7656.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2636.70</td>
<td>60.5716</td>
<td>3668.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2832.39</td>
<td>71.7163</td>
<td>5143.23</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>1600</td>
<td>2200</td>
<td>H1 ↔H8</td>
<td>1977.85</td>
<td>97.9755</td>
<td>9599.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2649.93</td>
<td>59.1871</td>
<td>3503.11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2867.86</td>
<td>57.7763</td>
<td>3338.1</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>3200</td>
<td>4000</td>
<td>H1 ↔H8</td>
<td>1985.69</td>
<td>87.5003</td>
<td>7656.31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2 ↔H7</td>
<td>2636.70</td>
<td>60.5716</td>
<td>3668.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H3 ↔H6</td>
<td>2832.39</td>
<td>71.7163</td>
<td>5143.23</td>
</tr>
</tbody>
</table>
5.2. EXPERIMENTS AND RESULTS

5.2.2 Solutions Comparison

In this experiment, we compared our monitoring tool with three others in the same scenario used in the first experiment. We compared our solution with Periodic polling, Payless (CHOWDHURY et al., 2014), and OpenNetMon (ADRICHEM; DOERR; KUIPERS, 2014).

We compared the accuracy and the network overhead of our solution against the others in an equal scenario to show the performance of our solution. These comparisons highlighted our strong points. We used in our solution the parameters configuration that provided minimum measurement error with the lower network overhead. As for Payless and OpenNetMon, we used the configuration presented in their work that provided minimum measurement error as well. Table 5.7 summarizes all parameters and used values for each monitoring solution.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAMon</td>
<td>base time=500ms, min time=500ms, max time=5000ms, k=-1, l=0.75 , lower packet threshold=20, upper packet threshold=60, lower byte threshold=300 upper byte threshold=800</td>
</tr>
<tr>
<td>Payless</td>
<td>min time=500ms max time= 5000ms, lower threshold=100MB, upper threshold = 100MB, alfa=2, beta=6</td>
</tr>
<tr>
<td>Periodic Polling</td>
<td>Frequency = 1000 ms</td>
</tr>
</tbody>
</table>

We used the same topology and flows presented in figure 5.1 and figure 5.3 respectively. We measured the throughput and the amount of OpenFlow statistics requests sent for each solution. With this information we can measure the error of each solution against the model and the network overhead generated by the monitoring activity, therefore validating the accuracy and the network overhead of our solution.

All the solutions worked well in the Mininet environment. However, when we moved to the physical environment both Payless, and OpenNetMon did not work as intended. Even worse, OpenNetMon became too unstable to be used. Nevertheless, we still measured Payless, even with the wrong behavior, and the periodic polling.

OpenNetMon was too unreliable to be measured because it was not only a monitoring solution, it was a router too, and the router did not work well in the real environment taking a minimum of 90 seconds to create a path between hosts or not creating one at all.

Figure 5.17 shows the comparison of the network utilization of our SHAMon, periodic polling, and the model. Payless network utilization measurement is in figure 5.18.

The measurement comparison in figure 5.17 shows the first problem with periodic polling, huge spikes. We reasoned that periodic polling accumulates some of the throughputs from the first measurement to the next, resulting in throughput that should be in the first measurement.
being in the next one. However, it did not ignore the traffic variations, getting all gaps correctly. Our SHAMon did not have spikes, being very similar to the model measurement. However, it stretched some bursts of the network traffic, making the gaps narrower than the ones of the model.

Payless utilization measurement is completely different from the model, so we did not measure the error (RSME). It calculated the network utilization in bursts as if it was requesting statistics from the switches in long time intervals. However, to measure the network overhead generated we captured all the OpenFlow traffic sent from each monitoring tool. In contrast with what the measurement was presenting, Payless was requesting frenetically statistics and was receiving them. Table 5.8 presents this information.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Measurement Error</th>
<th>Network Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHAMon</td>
<td>22.6085523087</td>
<td>6.27399</td>
</tr>
<tr>
<td>Periodic Polling</td>
<td>24.2700587383</td>
<td>19.9864</td>
</tr>
<tr>
<td>Payless</td>
<td>Not Measured</td>
<td>44.8606</td>
</tr>
</tbody>
</table>

As we expected, our SHAMon error measurement was lower than the periodic polling, and we used the moderate parameters combinations, as shown in the Scheduler experiment. In that experiment, with a $k$ and $l$ value of -0.5 and 1.0 respectively our error was 15%, however, we got a network overhead of 17 requests per second.

The network overhead measured from SHAMon was 6.3 requests per second, which
is 3 times lower than the periodic polling. If we used the parameters values that gave us the smallest error, network overhead would still be lower than the periodic polling that was almost 20 requests per second.

Payless measured network overhead was almost 45 requests per second. That was an incredibly high rate and it worked as another indicator that something was wrong with their POC implementation in our physical environment.

5.3 Final Remarks

In this Chapter, we presented the evaluation of our solution and its results. In our first experiment, we evaluated the two main decisions of our scheduling algorithm: **when to increase or to decrease the request time**, and **how much to increase or to decrease**. To validate the "when", we varied the thresholds that determine when the increment or decrement function is executed. And, to validate the "how much" we varied the parameters of the binomial algorithms, used as the increment and decrement function. The major objectives of this experiment were to understand the behavior of our scheduling algorithm in a real scenario, demonstrate the significance of its two main decisions, and clarify how the thresholds and parameters may be used to assist the network monitoring.

In the second experiment, we tried to compare how our solution worked in contrast with other monitoring solutions in the same network scenario. The major objective of this experiment was to highlight some of our qualities. However, the solutions we selected to compare with did not work in a real physical environment, only in the virtual environment of Mininet. Even
though, we compared our solution with the periodic polling solution, which is the basic solution that every SDN Controller comes with. The results proved that our solution works better than the basic solution, we measured the network utilization better, which made our measured error lower, and the network overhead of our solution to be three times smaller than the periodic polling.
CONCLUSIONS

In this chapter we will present our final remarks, detailing our conclusions about our SHAMon and this work. We explain our contributions and finalize with our proposition for future works.

6.1 Conclusion

In this work, we presented our Scalable High Accurate Monitoring (SHAMon), a solution for SDN. We designed our solution to be modular, scalable, fully using the OpenFlow protocol and the northbound API to account the network devices, acknowledge new flows in the network, and collect statistics from the devices. We followed the five logical division of the network monitoring task as a guide to how the modules should be in our solution, which resulted in a low coupled and with high cohesion solution.

We used a proxy-server architecture, an established and commonly used design to allow scalability, which resulted in the creation of two entities, the SHAMon Controller and the SHAMon Proxy. The Proxy entity being responsible for requesting statistics and send them to the Controller entity via REST. An API was defined to allow other applications to use the collected monitoring information in different levels of aggregation.

As the most important procedure of our solution, scheduling new statistic requests was designed as its own module, the Scheduler module. We chose to use binomial algorithms as the congestion control method for the polling time of requests, and to the best of our knowledge, it had never been used to control monitoring traffic. The binomial algorithm would act when the difference of bytes or packets from the last statistic request and the new one crosses the upper threshold or the lower threshold. When the difference is a value lower than the lower threshold, the polling time would be increased by the amount calculated by the binomial algorithm, when the difference value is greater than the upper threshold the polling time would be decreased.

The use of binomial algorithms and thresholds to control the polling time proved to be a versatility choice, allowing our solution to be configured to monitor different types of traffic in different kind of ways. To analyze this aspect, as well as the behavior of our solution we defined a set of experiments. The first set of experiments was intended on analyze the two functions of
our solution, the former was how variations of parameters of the congestion control algorithms are related to how much the polling time vary. The latter was how variations of the thresholds that limit the execution of the congestion control algorithms are related to the decision to vary the polling time.

Therefore, in the first experiment, we varied $k$ and $l$ parameters of the congestion control and evaluate the utilization measurement, the measurement error, the network overhead, and the polling time variation. As a result, we could see how more aggressive and moderate values of $k$ and $l$ affect our solution, making our measurement error value vary from 26% in more moderate values to 15% in more aggressive ones. The results showed that to gain the lowest error measurement the network overhead would double.

The utilization measurement gave some visual notion of how our solution work and how the $k$ and $l$ variations modify its behavior. The box plot and the table with the polling time variations for each combination and flows of the experiment gave important insights of how the polling time behaved, giving background to understand the network overhead generated. They presented how aggressive values for $k$ and $l$ made the polling time of the flows increase or decrease varying from, for example, 976 milliseconds in an aggressive approach to 3268 milliseconds in a moderate approach for the H1 to/from H8 flow.

Further, we varied the values of the upper and lower thresholds, which are the control for how our solution reacts to traffic variation, choosing to increase or decrease the polling time. We evaluate the utilization measurement, the measurement error, the network overhead, and the polling time variation. We varied the values from lower and narrow thresholds to higher and broader thresholds for byte variation and packet variation. And we used moderate values for $k$ and $l$.

The first conclusion the evaluation provided was that varying only the "when" we react does not result in a high variation for the network utilization measurement, error measurement, and network overhead. If more aggressive values for $k$ and $l$ were used, we probably would see different results between combinations, however not big ones. The greater indicator of the effects of the values combinations was found in the box plot and table with the polling time variation. The variance of the polling for each flow changed greatly across combinations, showing how different types of traffics behave with different types of thresholds. Even with higher variances, the mean of the polling time maintained almost the same.

In our last experiment, we tried to compare our solution with three others, Payless, periodic polling, and OpenNetMon, however only one worked right in our physical environment. Payless executed but did not measure things right, as presented in the network utilization graph, and OpenNetMon was incredibly unstable to be of use.

Nevertheless, we compared our solution with the base case, periodic polling, and our results were great. Our network utilization measurement was pretty good with an error of 22.6 from the model, while the periodic polling approach got 24.2. Furthermore, we used a less aggressive set of configuration parameters for our solution in this experiment, as was presented
we could get a 15 measurement error. But our choice was made to balance the measurement and the network overhead, that was 3 times smaller than the periodic polling. We measured the network overhead generated by Payless, and it was another proof that it was not working correctly, the overhead generated was 2 times greater than the periodic polling.

Our solution proved to be highly configurations to different network environments and demonstrated that the network overhead generated was lower in comparison to other solutions. Further, it is possible to achieve higher network measurement precision, however, the network overhead increase with it. We believe that the research objective was achieved, and a monitoring solution for SDN networks which uses the OF protocol, pulling techniques with congestion control algorithm, different architecture in order to minimize the resource and network overhead was created and evaluated.

6.2 Contributions

This work’s contributions are summarized as follows:

- A new application for binomial algorithms as a control function for scheduling problems;
- A novel design for monitoring solutions following the logical division of monitoring activity with a scalable architecture;
- A high configurable monitoring solution with accurate measurements and low network overhead, which provides a well-defined REST API with a variety of aggregation levels for the information to be queried;
- A solid evaluation of the behavior of binomial algorithms being used as the main polling time control for a scheduler;
- A comparison of our designed solution and the case base of every SDN Controller, the periodic polling.

6.3 Future Works

The first step of evolution for our solution is to implement the delay measurement between the network devices. We see two possible strategies to be used, the first one is the one that the OpenNetMon adopted: creating an exclusive VLAN between all devices and send probes to calculate it. The second one is OpenSample’s strategy: extract or calculate this information from the header of the packets. We do not have plans to implement the packet inspection feature yet, as it is a complex feature and need a thoughtful analysis.

As a natural step for SDN applications, we want to evolve the REST API to allow monitoring functions, some remote tasks, and to schedule jobs for our monitoring solution to
execute later. These modifications will enhance the interoperability of our application with other SDN applications.

A graphical interface to support the proper visualization of the collected data, generating network utilization graphics in all the aggregate levels and for all OF devices connected to the network, topology graphics, on the fly. This interface would easy the configuration of the solution parameters. A traffic analyzer module to help the network administrators understand the network flows crossing their networks and helping to configure our solution by finding the best parameters configuration that fit the needs of the administrator.

Further, we will improve our comparative analyses of the SDN monitoring tools so that a better validation of our work can be presented. This analyses should be done in a set of different topologies with different types of traffics crossing the network. A validation of the scalability of our solution should be done as well, with and without controllers ready to scale.
REFERENCES


Gini, C. Variabilità e mutabilità. [S.l.: s.n.], 1912.


NYGREN, A. et al. Openflow switch specification 1.5.1. 2015.


