4-23-2019 11:00 AM

Big Data for Traffic Engineering in Software-Defined Networks

Wander Queiroz, *The University of Western Ontario*

Supervisor: Capretz, Miriam, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Electrical and Computer Engineering

Abstract

Software-defined networking overcomes the limitations of traditional networks by splitting the control plane from the data plane. The network logic is moved to a component called the controller that manages devices in the data plane. To implement this architecture, it has become the norm to use the OpenFlow protocol, which defines several counters that are maintained by network devices. These counters are the starting point for Traffic Engineering activities. Traffic Engineering monitors several network parameters, including network bandwidth utilization. A great challenge for Traffic Engineering is to collect and generate statistics about bandwidth utilization for monitoring and traffic analysis activities. This becomes even more challenging if fine-grained monitoring is required. Network management tasks such as network provisioning, capacity planning, load balancing, and anomaly detection can benefit from this fine-grained monitoring. Because the counters are updated for every packet that crosses the switch, they must be retrieved in a streaming fashion. This scenario suggests the use of Big Data techniques to collect and process counter values.

The benefits of Big Data techniques for collecting and processing counter values are two-fold. First, Big Data techniques provide streaming processing tools that can deliver outputs in near real time, which can help early detection of anomalous traffic behaviour and possible proactive actions toward resolving the issue. Second, Big Data techniques provide batch processing tools that can deal with a large amount of historical data, which enable the use of Big Data analytics techniques to achieve a better understanding of traffic behaviour over time.

This research proposes an approach based on a fine-grained Big Data traffic monitoring method to collect and generate traffic statistics using counter values. This research work can significantly leverage Traffic Engineering. The approach can provide a more detailed view of network resource utilization because it can deliver individual and aggregated statistical analyses of bandwidth consumption. In the context of the proposed monitoring method, this research proposes a new approach to estimate the Traffic Matrix, a repository that maintains information about the traffic volume between all host origin-destination pairs. This research also proposes a traffic analysis method based on batch processing of historical traffic data. Experimental results show the effectiveness and potential of the proposed methods.

Keywords: Software-Defined Network (SDN), OpenFlow, Network Monitoring, Traffic Engineering, Traffic Matrix, Big Data Streaming, Traffic Analysis.

Co-Authorship Statement

The thesis presented here has been written by Wander Jácome de Queiroz under supervision of Dr. Miriam A. M. Capretz. Parts of the content of this thesis has been published in peer-reviewed journal, or is under review for publication. The research published in each paper has been mainly conducted and written by the principal author. The presented research is guided and supported by Dr. Miriam Capretz as the research supervisor.

A version of the material presented in Chapter 4 has been published in

- W. Queiroz, M. A. Capretz, and M. Dantas, "An approach for SDN traffic monitoring based on big data techniques," in *Journal of Network and Computer Applications*, vol. 131, pp. 28–39, Apr. 2019.
  - W. Queiroz - developed the model, collected experimental data, analyzed the results, evaluated the model and wrote the manuscript.
  - M. Capretz and M. Dantas - contributed to the interpretation of the work by critically revising the paper.

The material presented in Chapter 6 is currently under peer review for publication in

- W. Queiroz, M. A. Capretz, and M. Dantas, "A MapReduce Approach for Traffic Matrix Estimation in SDN," in *IEEE Access*, 2019.
  - W. Queiroz - designed the model and experiments, analyzed the results, evaluated the approach and wrote the manuscript.
  - M. Capretz and M. Dantas - contributed to the interpretation of the work by critically revising the paper.

Acknowledgements

First, I would like to express my deep gratitude to Dr. Miriam Capretz, my supervisor, for the support of my Ph.D. study, for her patience, guidance and for believing in me and in my research project. Her comments and suggestions had a huge impact on this thesis and improved the quality of this research work.

My sincere thanks also go to Dr. Mario Dantas, Dr. Wilson Higashino, Dr. Katarina Grolinger, Dr. Hany ElYamany, and Dr. He Fang for their help and encouragement.

I would like to offer my special thanks to my wife Leticia Queiroz and my beloved child Annie Queiroz for their unconditional support and making these years of research work a pleasant journey.

To my parents Ademar Moreira Queiroz (in memoriam) and Dilza Jácome Queiroz and brothers Jusley Jácome Queiroz, Ademar Moreira Queiroz Junior and Wladimir Jácome Queiroz for their support and encouragement.

A special thanks to my mother-in-law Ivonice Alves Rocha for solving our problems when we could not be there to solve them, and also for her love for Annie, Leticia and me.

Thanks to all my friends from TEB 346 and TEB 244, Roberto Barboza, Dennis Bachmann, Daniel Araya, José Miguel, Willamos Aguiar, Alexandra L'Heureux, Rafael Aguiar, Dr. Mahmoud ElGayyar and Norman Tasfi for providing a happy and healthy environment for research work.

Thanks to Prof. Quazi Rahman for the great privilege of being his TA and to all TA colleagues for the moments we shared in the labs and marking sessions.

Thanks to ECE department for the opportunity and the graduate coordinator Stephanie Tigert for always being available to help.

Contents

List of Figures

List of Tables

List of Acronyms

ARP	Address Resolution Protocol
API	Application Programming Interface
BPMN	Business Process Model and Notation
BR	Bytes Received
BRT	Bytes Received Throughput
BT	Bytes Transmitted
BTr	Bytes Throughput
BTT	Bytes Transmitted Throughput
ER	Entity-Relationship
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
LA	Lambda Architecture
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MDNS	Multicast Domain Name System
NOS	Network Operating System
OD	Origin-Destination
OF	OpenFlow
ONF	Open Networking Foundation
OSGI	Open Services Gateway Initiative
PR	Packets Received
PRT	Packet Received Throughput
PT	Packets Transmitted
PTr	Packet Throughput
PTT	Packet Transmitted Throughput
QoS	Quality of Service
REST	Representational State Transfer
SDN	Software-Defined Networks
TCP	Transmission Control Protocol
TE	Traffic Engineering
TM	Traffic Matrix
VLAN	Virtual Local Area Network
XML	Extensible Markup Language

Chapter 1

Introduction

Software-defined networking (SDN) is a paradigm designed to overcome the limitations of traditional networks [1, 10]. This approach makes the network more programmable and easier to configure. The problem with traditional networks is that the control plane and the data plane are embedded in the same device. As a result, no point in the network has a global view of the network. Such a design also makes it difficult to change the network configuration. Basically, each device must be manually configured to reflect a new configuration. This process becomes even more challenging for large networks. Implementing a new configuration can take large amounts of time and resources [10]. Several protocols have been proposed to implement SDN [11–14], but OpenFlow (OF) [15] is the industry standard nowadays.

A critical activity in SDN is traffic engineering (TE), which measures and manages network traffic [2]. TE plays an important role in network performance optimization by analyzing real-time traffic, predicting traffic, and designing routing mechanisms to improve network resource utilization [16, 17]. To carry out all these activities, network monitoring is crucial. One of the aspects of good monitoring is the design of network parameters. Network parameters are values that reflect current network status [17]. A widely used parameter to measure network performance is bandwidth utilization, which is sometimes referred to as network throughput.

One of the features introduced by the OF protocol is the use of counters [15]. Counters are maintained for a variety of objects in an OF switch. The introduction of counters enables a more direct approach to collecting traffic statistics. There are basically two approaches for collecting counter values: (1) indirectly using the SDN controller and (2) directly sending specific messages to the switches. The common approach is to use an SDN controller, which usually, in addition to collecting counter values, provides a wide range of configuration and service request APIs.

The starting point for TE in SDN is to collect counters provided by OF devices. These collected values enable monitoring tools to measure bandwidth utilization. In recent years, several monitoring tools have been proposed to measure bandwidth utilization [18–22]. However, a fine-grained measurement tool that leverages TE is still lacking. A fine-grained measurement provides details of network traffic and can be used as input to more advanced tasks such as finding traffic patterns and trends on routes and links.

OF counters can be retrieved in a streaming fashion. Because large networks usually consist of hundreds of hosts generating a huge amount of traffic, acquiring and processing this stream of data can be very challenging. This scenario strongly suggests the use of Big Data streaming techniques to address this issue because Big Data streaming is characterized by data streams in which data are received as a continuous, infinite, rapid, unpredictable, and time-varying sequence [23].

A significant output of traffic monitoring is a traffic matrix (TM). A TM provides the traffic volume going between any origin-destination (OD) pair of nodes in the network over a specific time interval [24]. This information is essential for several activities in traffic engineering (TE), such as switch load balancing and fault tolerance. For example, if a link fails, a new route between the OD pair affected by this failure needs to be selected among the available routes that connect these nodes. To select the best route, the volume of traffic in each possible route needs to be available for an optimal decision.

Due to the large number of combinations of OD pairs present in large networks, an accurate measurement of the TM becomes a hard task. One of the most significant challenges is the number of links between any pair of OD nodes. As the number of links increases, providing an accurate measurement becomes even harder. The traffic of each link in the path has to be retrieved to aggregate the traffic volume between A and B at a specific time. Collecting link traffic information at the same time is extremely difficult because of the lack of measurement infrastructure [25].

Using the appropriate APIs, an application can retrieve counter values for any switch connected to the SDN controller and use them to estimate the TM. However, even when counters had been introduced by the OF protocol, two challenges for TM estimation still remained: i) which approach to use to collect the traffic data (direct measurement or inference techniques), and ii) how to aggregate the traffic values to produce the estimated TM.

Direct measurement can lead to small errors in the estimated TM [26] because this technique aims to collect the traffic information in all devices and all links between the OD pairs in the network, but, as previously stated, this technique requires a measurement infrastructure. Inference techniques collect traffic samples and use specific methods to estimate the TM based on these samples. Both techniques provide the values that will be used to estimate the TM. However, the problem is how to aggregate these values. It is expected that the amount of data collected using direct measurement is larger than with inference techniques. If the collected data are going to be used for batch processing, this aggregation may not pose a challenge, but this is not the case for real-time processing. Due to the volume of data and to time constraints, especially for large networks, this aggregation can be challenging.

In addition to the traffic monitoring task, the data provided by the OF counters can be used to provide traffic analysis to deliver a wide range of insights about network usage to drive TE activities such as *flow management*, *fault tolerance*, and *topology update*. Figure. 1.1 provides the design of a TE approach based on Big Data techniques. Two modules compose the TE approach: the **Big Data Streaming** module and the **Traffic Engineering** module.

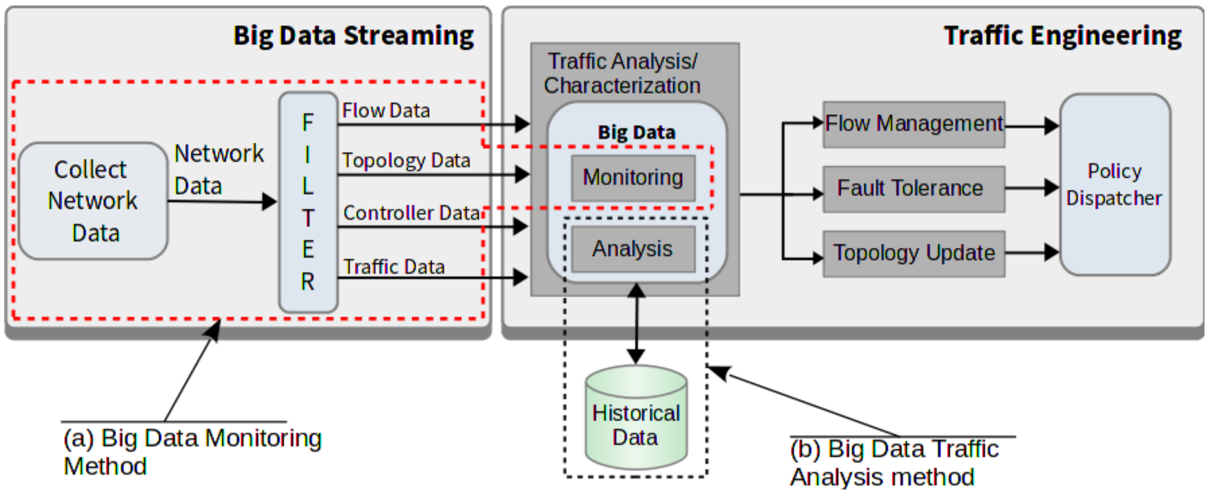


Figure 1.1: TE approach based on Big Data techniques.

As previously mentioned, the data coming from an OF switch can be collected in a streaming fashion because of the fast changes in the counters due to dynamic traffic crossing the switches. The **Big Data Streaming** module collects network data and forwards it to processing according to the data purpose (*Flow Data*, *Topology Data*, *Controller Data*, and *Traffic data*).

The **Traffic Engineering** module performs TE activities. The *Traffic Analysis/Characterization* activity processes the data coming from the **Big Data Streaming** module and performs the traffic *Monitoring* and *Analysis* tasks. The *Flow Management* activity carries out load balance on switches and controllers, the *Fault Tolerance* activity provides fast recovery in case of component failures, and the *Topology Update* activity works on network topology update for planned changes. These activities make decisions based on the data provided by the *Traffic Analysis/Characterization* activity.

This research proposes a Big Data traffic monitoring method (Figure 1.1 (a)) that provides on-line measurements of traffic throughput. The proposed method encompasses the **Big Data Streaming** module and the *Monitoring* activity to process streaming data and provide results in real time. Based on *Topology Data* and *Traffic Data*, the proposed method delivers fine-grained measurements of traffic throughput at the flow, port, link, path, and switch levels. ”Fine-grained” means the throughput of every switch, every port in each switch, every link, and every host origin-destination pair. The proposed method provides an approach for on-line TM estimation. To produce all the mentioned measurements a Big Data programming model called MapReduce [6, 7, 27] was used. The counter values collected from the SDN controller were processed by the MapReduce approach to generate the estimated TM for all the OD pairs in the network every *t* seconds.

This study also proposes a Big Data traffic analysis method (Figure 1.1 (b)). The Big Data traffic analysis method encompasses the *Analysis* activity and the *Historical Data* database to process historical data and reveal traffic trends and behaviour. Unlike the Big Data traffic monitoring method, the traffic analysis method is based on batch processing because it is expected a large volume of traffic data over time. The proposed method provides switch, switch port, link, and path traffic analysis.

To validate the Big Data traffic monitoring method, this study has provided an implementation following the design guidelines presented in Chapter 4. To validate the Big Data traffic analysis method this study provided an implementation following the design guidelines presented in Chapter 7.

1.1 Motivation

TE is an essential activity for SDN. The good health of the network depends on the activities performed by TE, such as load balancing and fault tolerance. The availability of real-time traffic statistics and historical traffic analysis will leverage TE to adapt to traffic changes and to ensure balanced usage of network resources.

Real-time traffic statistics play an important role in network monitoring for early detection of link and route congestion, enabling network administrators to take proactive actions to resolve congestion.

The design and implementation of a real-time traffic monitoring method face several challenges due to the need to process large amounts of data. This statement can be confirmed by reviewing a series of previous monitoring methods that usually selected a specific link, switch, and route to monitor.

A common problem faced by previous traffic monitoring methods was aggregating the collected statistics. For real-time measurement, depending on the amount of collected data, calculating aggregated link, switch, and route statistics may be time-consuming, making methods unable to meet real-time constraints. The aggregation process starts by storing the collected data, which arrive in a streaming fashion while previous data are still being processed. The stored data are then grouped by a predetermined key as they arrive in random order. The desired outputs are computed according to the grouped keys. Each one of the previous steps must be addressed by any approach, and in most cases, these approaches are application-dependent.

Big Data techniques provide robust and stable tools to tackle the aggregation problem. These tools provide a standardized, but flexible approach that can be used in a wide range of applications that must deal with streaming data and real-time processing. These tools provide a wide range of APIs to deal with different requirements and manage infrastructure horizontal scalability. Using Big Data techniques, large amounts of data can be processed seamlessly, and statistics can be provided in near real time.

The benefits of using Big Data techniques can also be extended to traffic analysis because the same type of aggregation used in traffic monitoring can also be applied to traffic analysis, with the difference being the amount of data. Traffic analysis requires more data because it searches for traffic characteristics such as traffic pattern. Traffic aggregations, as performed in traffic monitoring, are the basis for more advanced calculations.

1.2 Contributions

This research provides the following contributions to traffic monitoring and analysis tasks in the TE *Traffic Analysis/Characterization* activity for SDN:

- **Big Data traffic monitoring method**.
  - A novel method for monitoring network traffic based on Big Data techniques is presented. The proposed method defines various processes and a data model. Both the processes and the data model can be implemented using a variety of Big Data tools.
  - A fine-grained statistical analysis of network resources, such as the ratio of each port to the switch load and the throughput capacity used on each port, path, and switch, is provided. The deployment of the implementation is presented and provides a roadmap for a distributed environment that can run various Big Data tools.
  - An approach to the TM estimation problem using Big Data processing techniques and tools is presented, which to the best of our knowledge is the first attempt to use this approach. The proposed approach can produce the estimated TM between all OD pairs in the network in near real time, as well as the traffic volume in each active link in the network. Using the power of Big Data processing tools, the proposed

approach also produces what is called here the *actual* TM between all OD pairs. The actual TM is the difference between the currently calculated throughput and the last calculated throughput for each OD pair [18]. This TM enables network managers to keep track of the traffic evolution between any OD pair in near real time.

  – A direct measurement infrastructure to collect and process traffic data generated by the OF switches. The Big Data traffic monitoring method shows that direct measurement can be feasible if an appropriate infrastructure is provided, not only for collecting the counters, but also for aggregating the collected data.

- **Big Data traffic analysis method**.

  – A novel method for traffic analysis based on Big Data batch processing is presented. The method establishes the data dependency between the network resource statistics and traffic analysis of the resources. The generated traffic analysis can also be used to provide the historical behaviour of the network resources.

The methods proposed in this research work can help network administrators and monitoring applications to analyze the behaviour of network traffic and resource utilization. In general, these users perform TE activities to maintain healthy behaviour in the network traffic. The decisions made are based on information about the traffic in the devices and on resource utilization. This information plays a crucial role in decision-making. The more detailed the information, the better is the chance of a good response to an unexpected event. The proposed traffic monitoring and analysis methods provide a detailed view of network traffic and can help to improve both manual and automatic decisions.

For data center environments that can be reconfigured by "autonomic managers", the proposed methods provide continuous online traffic statistics and historical traffic analysis. These operations provide two main benefits to autonomic managers: (i) a wide range of traffic data are available to make decisions on dynamic reconfiguration, and (ii) once the decisions have been made and deployed in the network, the autonomous system can validate reconfiguration results by continuously incorporating the data provided by the proposed methods. This loop can operate without human intervention if the system requirements are met. The TE approach presented in Figure 1.1 shows that the proposed methods work together to provide data for the *Flow Management*, *Fault Tolerance*, and *Topology Update* activities to perform decision-making.

1.3 Thesis Organization

This thesis is organized as follows:

- Chapter 2 presents necessary background concepts to understand the scope of the research. It starts by presenting a review of software-defined networking and introducing the OpenFlow protocol and the activities of traffic engineering. In sequence, it presents the Big Data concepts used in this research, such as processing of streaming data, MapReduce, and Lambda Architecture.

- Chapter 3 presents an extensive review of research studies related to SDN traffic monitoring and traffic matrix estimation, which are the main contributions of this thesis.

- Chapter 4 presents the Big Data traffic monitoring method. The activities that compose the monitoring method are presented in detail. The description and definition of the data used to provide the statistics and the algorithms that collect these data are also presented.

- Chapter 5 presents the implementation of the Big Data traffic monitoring method. The SDN controller and the components of the implementation are described, as well as the deployment environment.

- Chapter 6 presents the MapReduce approach to estimating the TM. The approach includes the definition of basic data structures as well as the data collected to estimate the TM.

- Chapter 7 presents the proposed Big Data traffic analysis method. An activity diagram is presented based on the hierarchical data dependency needed to generate the traffic analysis.

- Chapter 8 presents an evaluation of the results obtained by implementing the proposed methods.

- Chapter 9 summarizes the thesis contributions and presents future work.

Chapter 2

Background

This section presents background concepts used in this thesis. The first section provides an introduction to SDN, describes the OF protocol and the activities of traffic engineering. Section 2.2 introduces Big Data concepts related to the streaming process and the MapReduce programming paradigm, which is the basis for the solution presented in Chapters 4, 6, and 7. Section 2.2 also introduces the concept of Lambda Architecture, an architectural solution used in Chapters 4, 6, and 7.

2.1 Software-Defined Networking

Current hardware-centric networks pose several operation and management challenges [1, 28]. The core of such a network's infrastructure contains forwarding devices like switches and routers. These have two logical components for traffic management: the control plane (which decides how to handle network traffic) and the data plane (which forwards traffic based on decisions made by the control plane) [3, 10, 29]. This approach makes management of traffic flows very time-consuming especially for large networks, because any new configuration has to be implemented manually in each device. Furthermore, different vendors have their own sets of rules to configure their devices, making the management task even harder.

A new network architecture has been proposed to overcome the problems and limitations of traditional networks. Software-defined networking (SDN) can be defined as "*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*." [30]. The main idea of SDN architecture is to split the control plane from the data plane (which just does forwarding). One consequence of this approach is that the control logic is implemented in a logically centralized controller and that network switches simply forward packets. The controller is now programmable, making it possible to establish dynamic routes for network packets. Figure 2.1 shows the deployment of control and data planes in traditional networks and SDN.

Compared with traditional networks, SDN provides the following advantages:

1. Global control. The controller has a global view of the network topology, network status, and application requirements.

2. Programmability and flexibility. The data plane can be dynamically programmed to improve network resource allocation.

3. Openness. Communications between the controller and forwarding devices do not depend on the device suppliers.

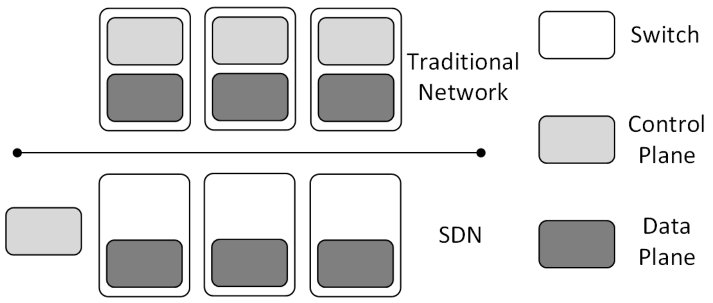


Figure 2.1: High-level illustration of control and data plane deployment (adapted from [1]).

SDN structure is presented in Figure 2.2. The components are described below [1, 3, 10, 29, 31]:

- *Data-Plane Layer*. Represents the interconnected forwarding devices. These devices have a set of instructions that specify actions to be taken on incoming packets (e.g., forward to specific ports, forward to the controller, rewrite header, drop temporarily or permanently).

- *South-Bound Open APIs*. Defines the set of instructions used for interaction between control-plane elements and forwarding devices. These also defines the communication protocol between them.

- *Control-Plane Layer*. Manages the forwarding devices using a controller (a network operating system (NOS) [4, 32]) to achieve specific application goals, and also provides an abstract view of the entire network infrastructure.

- *North-Bound Open APIs.* SDN Controllers provide services for applications by means of APIs. These APIs represent the northbound interface.

- *Application layer.* Applications benefit from decoupling between the control and data planes. They can use the northbound API to implement special purpose operations such as routing, monitoring, and traffic engineering.

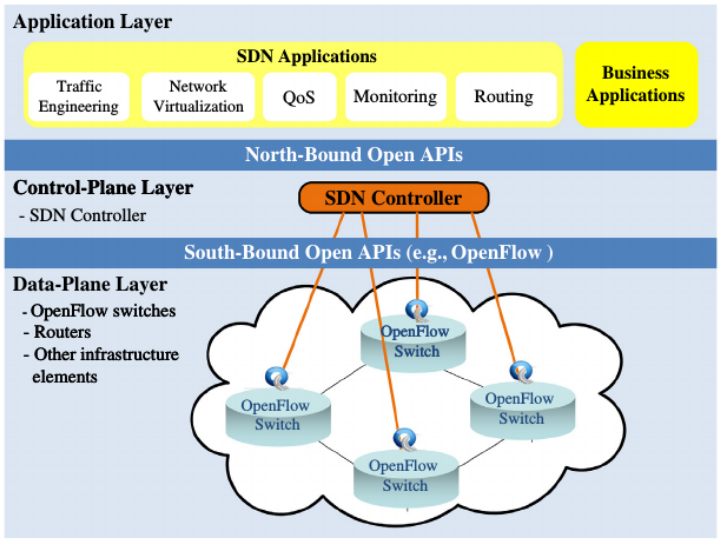


Figure 2.2: Basic SDN architecture [2].

A flow, in general terms, is a sequence of packets that traverse the network and that share a set of header field values such as the same source and destination IP addresses, the same VLAN identifier, and the same MAC address [10, 31]. Each new flow needs permission from the controller, which checks the flow against network policy. The SDN controller determines which flows are allowed to go through the data plane. More specifically, the first packet of every new flow is sent to the controller by the data plane to obtain permission to continue in the network and also to obtain its route across all the forwarding devices established.

SDN supports a centralized and distributed architecture. In a centralized architecture, one single controller manages the entire network. This controller must have a global view of the network topology and keeps track of all information about flows and loading on each forwarding device [33]. This centralized controller offers a single point of management for all network devices, but also presents several limitations such as lack of scalability and fault tolerance. Because the first packet of every new flow is forwarded to the controller, if the number of flows or the number of devices increases, the controller can easily become a bottleneck. Moreover, if the controller fails, the entire network can become unstable.

A distributed architecture shares the control function among several controllers (Figure 2.3). This solves the single point of failure problem and offers the following benefits [31]:

- *Scalability*. The number of devices that a single controller can manage is limited. Using multiple controllers helps distribute the load.
- *Privacy*. A set of devices can be assigned to customers who want to implement their own privacy policies. In this case, the controller managing these devices can protect information that customers do not want disclosed.
- *Incremental deployment*. Dividing device management among several controllers enables flexible incremental device deployment.

The distributed architecture introduces the need for controllers to communicate with each other. For this purpose the east and west bound APIs are used. Currently, the implementation is not standardized, and therefore each controller implements its own APIs [3]. The Internet Research Task Force has proposed a protocol called SDNi [34] to exchange information between SDN Domains Controllers. An SDN domain is a set of devices controlled by a single controller.

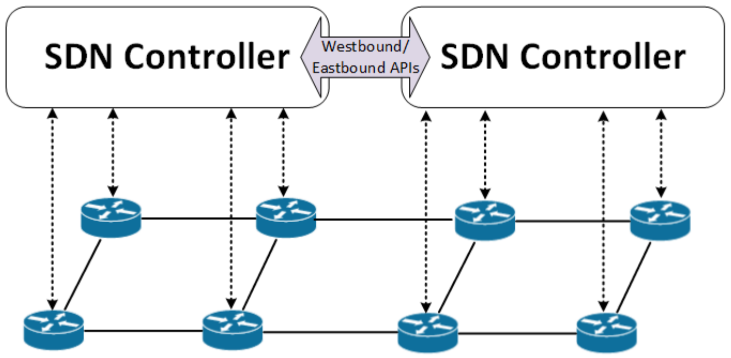


Figure 2.3: Distributed architecture (Adapted from [3]).

2.1.1 OpenFlow

According to Stallings [31], to implement an SDN, two requirements must be met:

- A common logic architecture must be present in all network devices managed by the controller. Even if different vendors implement this logic in different ways, the controller must see a set of uniform functions.
- A standard secure protocol must be used for communication between the controller and devices.

OpenFlow has met these requirements by providing a specification of the logical format of network switching functions and also by being a communication protocol between controllers and devices. The OpenFlow architecture consists of switching equipment managed by an OpenFlow controller (Figure 2.4). OpenFlow is defined by the *Open Networking Foundation* (ONF) in the *OpenFlow Switch Specification* [5].

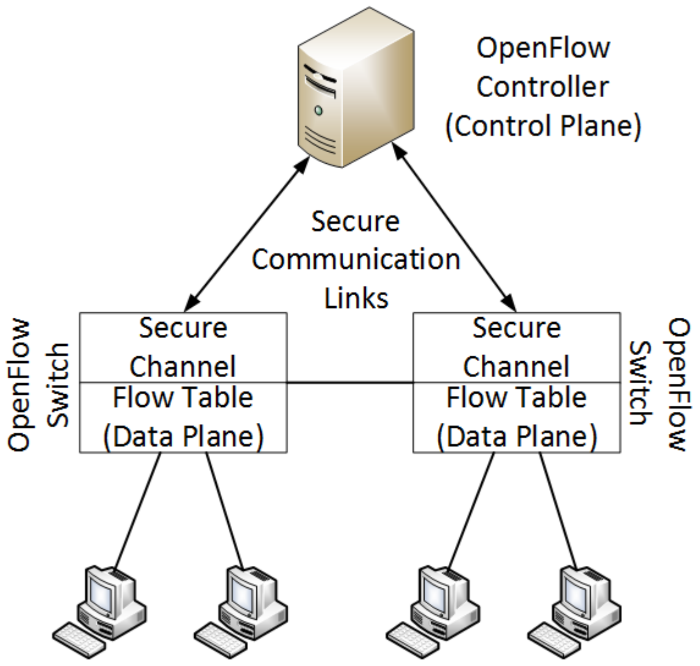


Figure 2.4: Basic OpenFlow architecture (Adapted from [4]).

The main activities of an OpenFlow switch are based on a flow table. In cases where more than one flow table exists, they are organized as a pipeline (Figure 2.5). A flow table consists of flow entries and each entry has six parts (Figure 2.5). *Match* fields are used to match against incoming packets and consist of the ingress port and packet headers. *Instructions* modify the action set of pipeline processing. *Counters* are updated when packets are matched. *Timeouts* specify the maximum amount of total time or of idle time before a flow is expired by the switch. *Priority* is used for matching precedence of flow entries. *Cookie* is the opaque data value chosen by the controller. The controller can use *Cookie* to filter flow statistics, flow modifications and flow detection.

Every flow table must accommodate a table-miss flow entry. This entry specifies the action in case a packet does not match any of the entries in the table. In this case, the packet can be forwarded to the controller, dropped, or directed to a subsequent table. The table-miss entry has basically the same behavior as any other flow entry. It is not included by default, and the controller can add or remove it at any time.

In terms of an individual OpenFlow switch, a flow is a sequence of packets that matches a

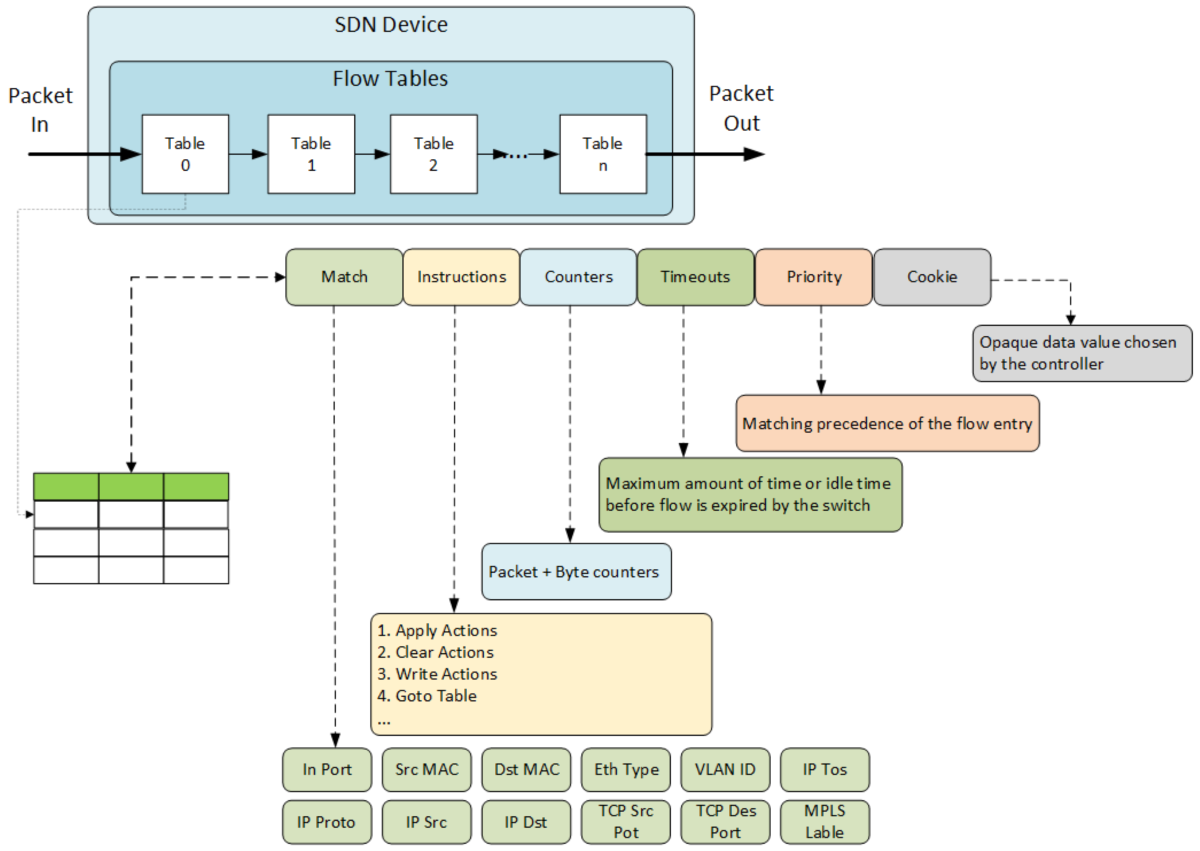


Figure 2.5: OpenFlow device.

specific entry in a flow table [31].

The *Instruction* part is a set of instructions that is applied to all matching packets. With these instructions, it is possible to modify the state of a packet, direct the packet to a specific port, forward the packet to another table, and pass metadata across the pipeline. Each packet is associated with an action set, which is a list of actions that are stored while the packet is processed by each table and executed when the packet leaves the pipeline. Actions can drop, modify, queue and forward packets [5].

When the device receives a packet, it proceeds in the following way [5, 31] (Figure 2.6):

1. Perform a table lookup in the first flow table, trying to find the highest-priority match. In case there is no match and no table-miss entry, the packet is dropped. In case of a match only on a table-miss entry, one of three procedures is specified:

   (a) Forward the packet to the controller. In this case, the controller can create a new flow for this packet or simply drop it.

   (b) Forward the packet to another flow table in the pipeline.

   (c) Drop the packet.

2. In case of a match on one or more entries, the match selected is the one with the highest priority value. The procedures specified for this case are:
    (a) Update counters.
    (b) Execute the instruction present in this entry. This instruction may update the action set, update metadata, and execute actions.
    (c) Forward the packet to a table in the pipeline or to an output port.
3. If the packet is sent to an output port, the action set is executed, and the packet is queued for output.

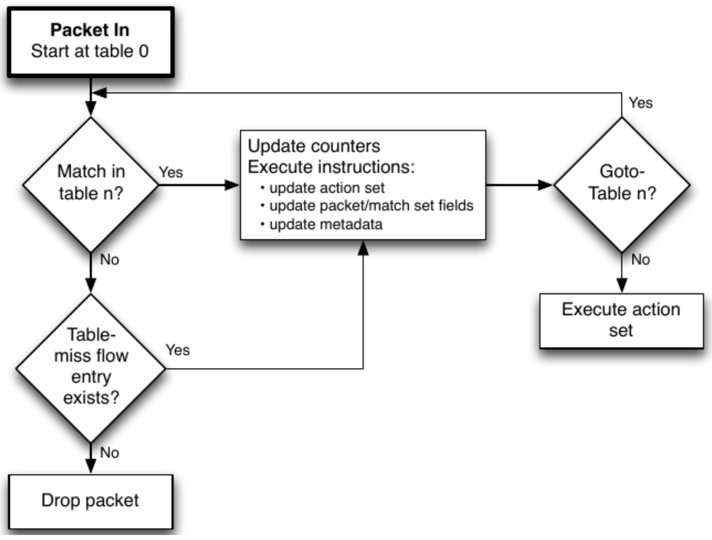


Figure 2.6: Packet flow through an OpenFlow switch [5].

Counters are an important component of a flow table. They are available for each flow table, flow entry, port, queue, group, etc. The OpenFlow 1.4 specification defines a set of counters, but a switch is not required to support them all (Table 2.1).

The scope of this research is located in the application layer (Figure 2.2), more specifically in traffic engineering. The next section provides an overview of this application.

2.1.2 Traffic Engineering

Traffic Engineering (TE) plays an important role in network performance optimization by analyzing real-time traffic, predicting traffic, and designing routing mechanisms to improve utilization of network resources [16, 17]. Using SDN in a network provides a more effective way to perform TE and to improve network performance [35].

Counter	**Bits**	
Per Flow Table		
Reference Count (active entries)	32	*Required*
Packet Lookup	64	*Optional*
Packet Matches	64	*Optional*
Per Flow Entry		
Received Packets	64	*Optional*
Received Bytes	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Port		
Received Packets	64	*Required*
Transmitted Packets	64	*Required*
Received Bytes	64	*Optional*
Transmitted Bytes	64	*Optional*
Receive Drops	64	*Optional*
Transmit Drops	64	*Optional*
Receive Errors	64	*Optional*
Transmit Errors	64	*Optional*
Receive Frame Alignment Errors	64	*Optional*
Receive Overrun Errors	64	*Optional*
Receive CRC Errors	64	*Optional*
Collisions	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Queue		
Transmit Packets	64	*Required*
Transmit Bytes	64	*Optional*
Transmit Overrun Errors	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*

Table 2.1: List of counters.

SDN traffic engineering can benefit from SDN characteristics to help solve its problems, which can be divided into flow management, fault tolerance, topology update, and traffic analysis and characterization [2]. Each of these topics presents several issues, including the ones depicted in Figure 2.7.

According to the basic operation of packet flow management described in Section 2.1.1, if a packet does not match any entry in the flow table other than the table-miss, this packet can be directed to the controller. This is usually the case for the first packet of every new flow [3]. In this case, the controller sends a forwarding entry to be installed in the flow table so that all subsequent packet of this flow can be forwarded without interacting with the control plane. If the network traffic presents a high number of new flows, significant overhead can result in the controller and in the data plane. The flow management component of TE should address this issue by balancing the load between switches and controllers.

Network reliability is based on the capability for fast recovery in case of failures in network components (controllers, switches, or links) [36]. Fast failure recovery is very challenging because the controller must calculate new routes and broadcast them to all affected switches. In addition, the limited flow-table resources at switches must be taken into account because

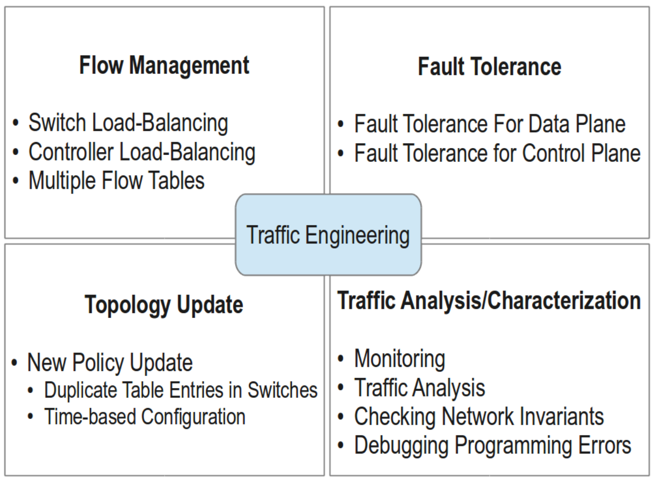

Flow Management
• Switch Load-Balancing
• Controller Load-Balancing
• Multiple Flow Tables
Fault Tolerance
• Fault Tolerance For Data Plane
• Fault Tolerance for Control Plane
Traffic Engineering
Topology Update
• New Policy Update
• Duplicate Table Entries in Switches
• Time-based Configuration
Traffic Analysis/Characterization
• Monitoring
• Traffic Analysis
• Checking Network Invariants
• Debugging Programming Errors

Figure 2.7: Scope of traffic engineering [2].

new entries will be added to the flow table of the affected switches.

The topology function update manages planned changes such as updating network policy rules instead of component failure. The controller dynamically configures the new global policy rules. This task must maintain the consistency of policy rules across switches because during this update, packets may be handled by different rules in different switches. Moreover, during the update, some packets may be dropped or delayed, causing poor network performance. Updating topology is even harder for a large network because it must be accomplished in near real time.

Traffic analysis plays an especially critical role in detecting network or link failures and predicting link congestion or bottleneck. Monitoring tools are the basic elements of this critical task. To achieve good monitoring, three aspects are important [17]:

1. *Network parameter design and monitoring technologies*

   - Network parameters are values that reflect current network status. The choice of these parameters is a precondition for good network management. Three types of parameters are essential for monitoring: i) *Network topology* parameters represent the number of nodes, link bandwidth, connections structures, and port status; ii) *Network traffic* parameters represent the number or speed of the packets that cross forwarding devices; and iii) *Network performance* parameters represent per-flow metrics such as throughput, delay, packet loss, etc. [19].

2. *Generic measurement framework*

- Several SDN measurement systems use traditional IP network traffic methods, randomly sampling local packets to obtain traffic statistics.

3. *Traffic analysis and prediction*

    - The goal is to identify anomalous traffic and analyze the possible presence of network congestion. The data provided will help improve traffic scheduling and management.

This thesis focuses on monitoring technologies, generic measurement framework and traffic analysis.

2.2 Big Data

The term Big data refers to various types of large and unstructured datasets that cannot be processed with conventional data processing systems in a reasonable time [37]. Size is the first characteristic used to understand big data [38], but it is not the only one. Laney [39] suggested three dimensions to characterize big data: *Volume*, *Variety*, and *Velocity* (*Three V's*). Volume refers to the huge amount of data produced by applications. Variety relates to the presence different data formats in a dataset. Different technologies enable the use of structured data (spreadsheets, relational databases), unstructured data (text, images, audio, video), and semi-structured data (XML). Velocity refers to the speed at which data are generated and analyzed. In addition to these three dimensions, other V's are also associated with big data: *Veracity*, *Variability*, and *Value* [38]. Veracity (introduced by IBM) refers to the presence of imprecise and uncertain data in some sources. Variability refers to the different flow rates of data. Value (introduced by Oracle) expresses the concept that data received in their original form generally presents low value relative to their volume.

2.2.1 Processing of Streaming Big Data

Batch and real-time are two types of data processing for big data. Choosing the more suitable type of processing for a specific application depends on the type and sources of data, the processing time needed, and whether immediate action is required [40]. Real-time processing involves continuous input, processing, and output of data and is needed when immediate action must be taken based on the processed data.

Streaming Big Data is characterized by data streams in which data are received as a continuous, infinite, rapid, unpredictable and time-varying sequence [23]. Some streaming Big Data applications are monitoring (network traffic, sensor networks, health care, etc.), surveillance, and financial transactions. For these types of applications, response time and throughput are critical. This means that most streaming Big Data applications have real-time requirements.

According to Stankovic et al. [41] there are eight misconceptions in real-time data management, of which one of the most important is "*real-time computing is fast computing*". Fast processing does not mean that real-time constraints are satisfied. Moreover, the meaning of "real-time processing" depends on the application. For some applications, real time means milliseconds, whereas for others, it means microseconds [42].

According to Shahrivari [43] the term "real-time" in Big Data is more related to interactivity than to millisecond response. Real-time query processing in a Big Data environment should return results in the order of seconds or minutes, as opposed to batch jobs that usually finish in hours or days.

The monitoring method proposed in this thesis provides results in order of seconds, and, therefore it can be considered as providing monitoring results in real-time.

2.2.2 MapReduce

MapReduce is a programming paradigm for processing large volumes of data [6, 27, 44] in parallel. The idea behind MapReduce is to distribute the data and the processing among several nodes for horizontal scalability. According to the name, the MapReduce model contains two phases:

- **Map phase**. In this phase, multiple map jobs read blocks of data and process them to produce key-value ($\langle K, V\rangle$) pairs. Several map jobs can be executed to generate intermediate $\langle K, V\rangle$ pairs.

- **Reduce phase**. In this phase, reducer jobs read the $\langle K, V\rangle$ pairs generated from multiple map jobs and aggregate or group them by key, producing the final output.

In the MapReduce flow (Figure 2.8), the data to be processed are first divided into splits, and the master node assigns these splits to map workers' nodes. Each worker processes its own splits to generate intermediate files with $\langle K, V\rangle$ pairs. The reduce workers, driven by the master node, process the intermediate $\langle K, V\rangle$ pairs according to their reduce functions to provide the final output.

Figure 2.9 shows how to perform a word count on a sequence of words using MapReduce. The goal is to find and count the number of occurrences of unique words in the sequence.

Initially, the sequence is divided into four splits. The map functions (M1...M4) act on each split to generate a $\langle K, V\rangle$ pair, where each word is the K and the numeric constant 1, the number of occurrences of the word, is the V. After the map phase, sorting and shuffling processes send

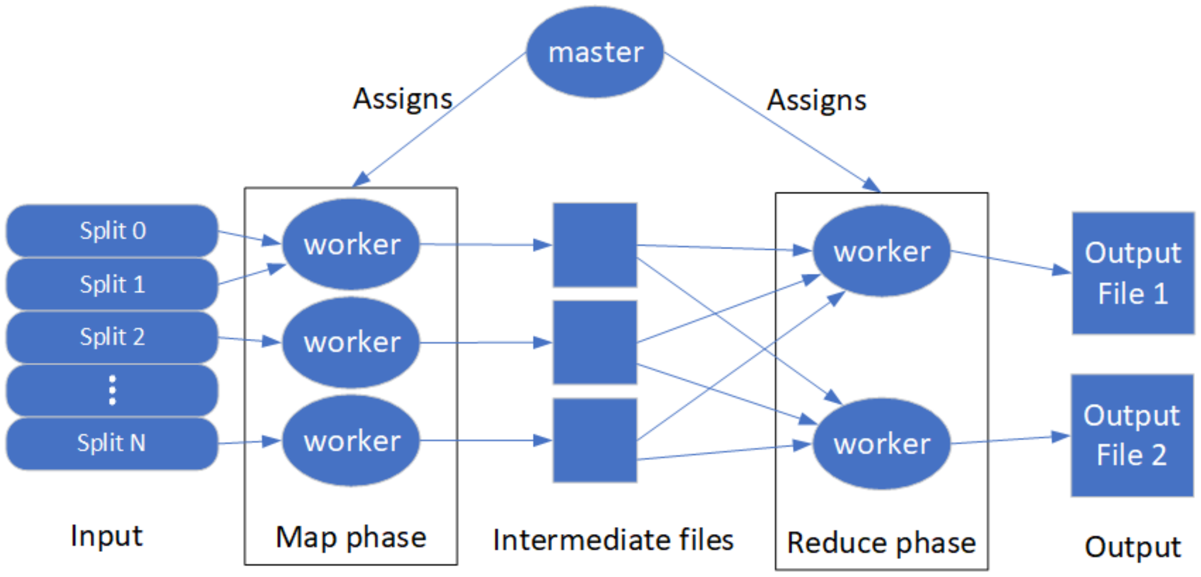


Figure 2.8: MapReduce flow [6].

the tuples with the same key to the corresponding reducer (R1...R4). Now each reducer has a list of unique keys with their corresponding values and adds all the values to get the final result. The final result is provided in the output file.

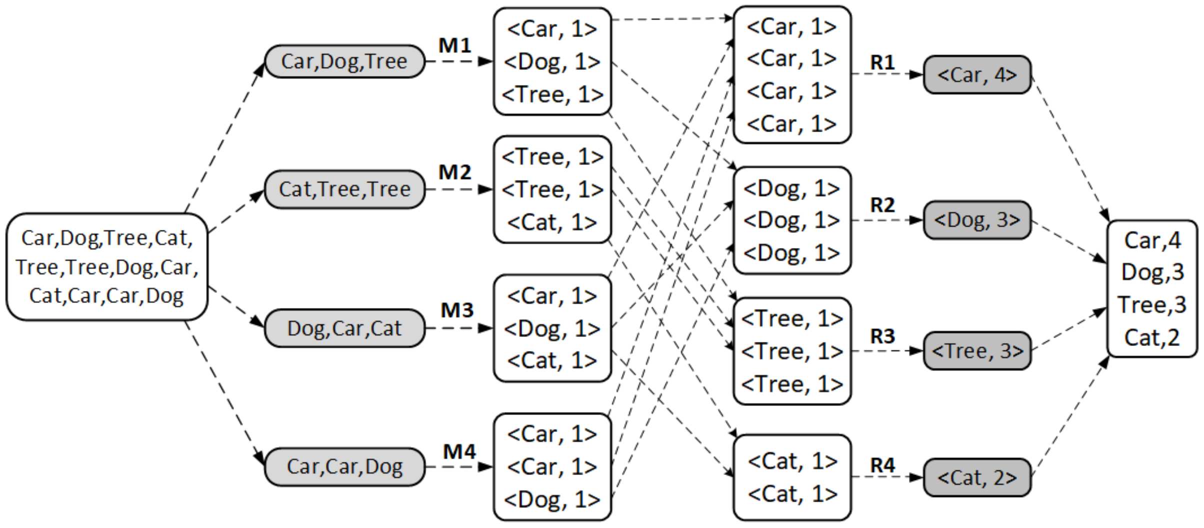

Car,Dog,Tree,Cat,
Tree,Tree,Dog,Car,
Cat,Car,Car,Dog
Car,Dog,Tree
Cat,Tree,Tree
Dog,Car,Cat
Car,Car,Dog
M1
M2
M3
M4
<Car, 1> <Dog, 1> <Tree, 1>
<Tree, 1> <Tree, 1> <Cat, 1>
<Car, 1> <Dog, 1> <Cat, 1>
<Car, 1> <Car, 1> <Dog, 1>
<Car, 1> <Car, 1> <Car, 1> <Car, 1>
<Dog, 1> <Dog, 1> <Dog, 1>
<Tree, 1> <Tree, 1> <Tree, 1>
<Cat, 1> <Cat, 1>
R1
R2
R3
R4
<Car, 4>
<Dog, 3>
<Tree, 3>
<Cat, 2>
Car,4
Dog,3
Tree,3
Cat,2

Figure 2.9: MapReduce example (Adapted from [7]).

The MapReduce paradigm [44] can process a significant number of real-world problems. A complex problem may involve several MapReduce operations and can be time- and resource-consuming. To tackle this problem, two flavours of MapReduce implementations are available, Hadoop and Spark. These tools differ from each other not only by their data flow operators [27] but also by their approach to intermediate files. Hadoop keeps the intermediate files on disk, whereas Spark keeps them in memory. This difference suggests the use of Hadoop for batch processing and Spark for real-time processing. Spark can also be used for stream processing using a framework called Spark Streaming. Spark streaming processes stream data in micro-batches, where every micro-batch contains the messages that arrived over the previous micro-batch period. Spark and Spark streaming processing are the basis of the MapReduce approach presented in this research.

2.2.3 Lambda Architecture

The Lambda Architecture (LA) is a Big Data system composed of a series of layers [8]. The idea behind the LA is to build a system that can process and produce different views of the data. A system built based on the LA can process stream and batch data. The LA consists of three layers: the speed layer, the batch layer, and the serving layer, as shown in Figure 2.10.

The speed layer receives a stream of data, processes them, and updates the real-time views. The speed layer performs incremental processing, does not keep historical records of the data, and processes the incoming data using stream processing platforms. In contrast, the batch layer repeatedly processes all the available data in the master dataset. Hence, when a batch job starts, the entire set of data in the master dataset will be reprocessed. The idea is that the next job will automatically process the data that do not arrive on time to be processed by the current job. As a result, each batch job replaces all the batch views. The serving layer stores the views produced by batch processing on the master dataset. The query processor collects data from the real-time and batch views to reply to users' queries.

The speed layer and the batch layer have different processing requirements, i.e. the speed layer processes a small amount of data with quick response, whereas the batch layer processes a large amount of data (historical data) with slower response. Consequently, they require different technologies to match their processing requirements. The serving layer demands technology for storing a large volume of data and producing a quick response to users' queries. The LA, being a design for Big Data processing, allows the use of various processing and storing technologies.

2.3 Summary

This chapter has introduced the main concepts used in this thesis. The initial section presented the concept of SDN, described this new approach for computer networking, and also described the activities of traffic engineering which are the focus of this thesis. Next, Big Data processing was presented, with a focus on the MapReduce approach and Lambda Architecture.

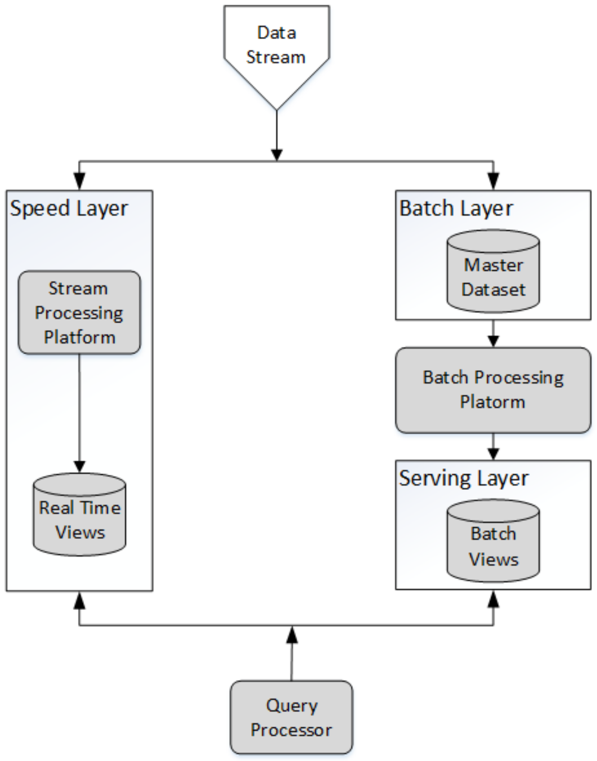


Figure 2.10: Lambda architecture (Adapted from [8]).

The next chapter provides a broad review of the academic literature related to the contribution of this research.

Chapter 3

Literature Review

This chapter presents research studies related to this thesis, which are divided into two categories: Traffic monitoring and Traffic matrix estimation.

3.1 Traffic Monitoring

For traditional IP networks, monitoring techniques such as NetFlow [45] and sFlow [46] are available. NetFlow, developed by Cisco, delivers statistics about individual IP flows by maintaining a flow cache that tracks statistics for each flow. It provides detailed data about each flow, such as source and destination IP addresses, byte count, packet count, and port number. sFlow uses time-based sampling to estimate the number of packets and bytes in each flow by multiplying the number of sampled packets and bytes by the sampling ratio.

Many traffic monitoring tools have been proposed for OpenFlow networks. OpenNetMon [19] was developed to determine whether end-to-end QoS requirements are met and to enable TE applications to compute appropriate paths. It is implemented as a module inside the SDN controller and polls edge switches to collect flow statistics at an adaptive rate to determine throughput, packet loss, and delay. OpenNetMon uses an adaptive rate for switch polling.

FlowSense [47] proposed a passive monitoring method by which the network informs the application of performance changes instead of polling the switches on demand. To achieve a low communication overhead, FlowSense uses `PacketIn` and `FlowRemoved` messages to estimate per-flow link utilization. FlowSense works best in networks with small flows.

Lavanya et al. [48] proposed a measurement framework where OF switches match packets against a small set of rules and update the counters for the highest-priority match. These rules are dynamically updated by the controller to identify large traffic aggregates quickly. Their solution programmed the switches to handle all incoming packets, to reduce controller overhead.

Planck [49], which is a network measurement architecture, used port mirroring to collect network traffic information at 280μs-7ms timescales on a 1 Gbps commodity switch and 275μs-4ms timescales on a 10 Gbps commodity switch. Port mirroring enables traffic monitoring by replicating the traffic going to an output port to a designated monitor port. Planck configured the switch such that the traffic going to multiple output ports was replicated to each monitor port and was used to drive a TE application to reroute congested flows in milliseconds.

Chowdhury et al. [50] proposed Payless, a monitoring framework for SDN. Payless was designed to use the OpenFlow controller's northbound API to collect flow statistics from the controller. To avoid continuous polling of switches for traffic statistics by the controller, Payless proposed a flow statistics collection algorithm that polls the switches at variable frequency. Payless sends, on average, 6.6 monitoring messages per second, in contrast with 13.5 messages per second for periodic polling. This algorithm achieves accuracy as high as continuous polling with lower network overhead. Furthermore, Payless provides a RESTful API for developing network monitoring applications. This API provides interfaces for collecting flow statistics at different levels (flow, packet, and port). The overhead imposed by Payless is low compared to controller overhead.

OpenSample [51] leverages sFlow packet sampling to provide near real-time measurements of both network load and individual flows. OpenSample captures packet header samples and uses TCP sequence numbers from these packets to reconstruct flow statistics. At the same time, OpenSample uses the packet samples to estimate port utilization.

OpenSketch [52] proposes a three-stage data plane pipeline for SDN. A measurement library for automatic data plane configuration is built. This library enables the development of new stream algorithms to monitor flows in commodity switch components.

An available bandwidth measurement application was proposed by Megyesi et al. [18]. Their application can travel the network topology and track bandwidth utilization over network links. As a result of this procedure, the application can calculate the available bandwidth between any two points in the network. By periodically polling the SDN controller, the application can calculate the current load on each link.

Luong et al. [20] proposed a solution for monitoring throughput of link traffic and a new forwarding algorithm for the control plane. These are based on two modules for the SDN controller. The *Throughput Monitor* module focusses on the number of packets and bytes that cross the switch. The *Packet Forwarding* module implements the forwarding algorithm. The statistics provided by the *Throughput Monitor* module are available for use by any monitoring application. In the proposed approach, throughput statistics are used on each port to provide aggregated statistics at different levels of resource monitoring. This makes it possible to provide this information to monitoring applications by persisting the computed aggregated values in a NoSQL database.

Sinha et al. [53] proposed an on-line per-flow and per-port monitoring and measuring approach to provide packet loss statistics in SDN. This study included experiments that monitored flows between two pairs of origin-destination hosts, unlike the present study that monitors traffic between all host origin-destination pairs in the topology.

Suárez-Varela and Barlet-Rois [54] presented a scalable flow-based monitoring solution for OF switches like NetFlow/IPFIX. Two sampling methods were designed to reduce both controller overhead and the number of entries in the flow tables. One basic difference between their solution and the present one is that the approach proposed here monitors ports instead of flows and does not use sampling methods.

SDN-Mon [55] was developed to monitor flows and free the controller from monitoring activity, enabling the controller to insert more general forwarding rules with wildcards in the switch. SDN-Mon contains two modules: a controller-side module, and a switch-side module. The controller-side module enabled flexible monitoring in the SDN controller. The switch-side module provided components to handle the monitoring functionality in the switches.

Suárez-Varela and Barlet-Rois [22] presented a flow monitoring and classification solution for OF switches. Their flow monitoring system uses a sample-based approach. A set of entries are installed in the switch to enable identification of the traffic to be sampled. When a packet arrives in the switch, a match operation is performed to check whether the packet is in one of the per-flow monitoring entries. If it matches, the packets and byte counters are updated. Suárez-Varela and Barlet-Rois [56] also presented an alternative monitoring method based on matching on ports for switches with no support for IP masks with suffixes. Their method consisted of installing a set of entries in the switch to enable direct discrimination of the traffic to be sampled. Consequently, only the first packets of these flows are sent to the controller, and the controller installs reactive specific flow entries to sample these flows.

Hartung and Körner [57] proposed SOFTmon, a traffic monitoring application that could provide charts and statistics up to flow level. SOFTmon supports flow monitoring only for the network layer, which means that a flow needs to have valid IPV4 entries in the source and destination addresses as well as Ethernet source and destination addresses to become a selectable item in SOFTmon.

Hendriks et al. [58] also used a per-flow statistical measurement approach. They evaluated the quality of measurements of OF devices from multiple vendors to show that it was impractical to deploy the OF device measurement-based approach in a network containing devices from multiple vendors.

FlowMon [59] provided a sample-and-fetch-based mechanism to detect large flows. FlowMon consists of two stages: a sampling stage and a counting stage. The sampling stage, using packet sampling, finds suspiciously large flows. The counting stage uses the flow table counting method to determine the truly large flows among the suspicious ones.

Afek et al. [60] developed a packet sampling mechanism and algorithms to detect large flows with a good and practical trade-off between switch and controller traffic. Their paper presented various methods to sample packets in an SDN switch that could be used independently of the large flow detection algorithm. The sample methods define the rate for creating counting rules in the switches and the rate for sampling packets from the switches.

Tahaei et al. [21] proposed an architecture to pull statistics using local controllers and forward them to an upper-layer application to aggregate counters and model a universal flow measurement in the network. According to their design, several controllers connect to the OF devices, and these controllers install wildcards for all requested flows in a single group. Afterwards, aggregated pull requests are used to collect statistics and send them to a coordinator for aggregation of network traffic.

Frunza et al. [61] developed and implemented a monitoring tool for SDN. Their monitoring station is not part of the SDN topology, i.e., it does not connect to the SDN controller, but rather is part of a traditional TCP/IP network. The monitoring station sends requests for statistics messages directly to the OF switch and receives the requested data through a tunnel. In this case, the monitoring station is acting as an SDN controller in the sense that the sent messages are OF messages and not API messages.

Rezende et al. [62] proposed SDNMon to monitor the data plane and to improve topology information at the control plane. The goal of SDNMon is to provide bandwidth and latency at two granularity levels, *Per-Port* and *Per-Flow*. SDNMon is implemented as an extension module of the SDN controller. Based on the network topology, SDNMon exploit threads to collect selected port and flow statistics and uses two monitoring approaches: (i) the sFlow protocol and (ii) a polling mechanism. SDNMon enables the implementation of other monitoring approaches through a *Monitoring Interface*.

Cheng et al. [63] proposed a compressive traffic monitoring method to collect real-time load information on network links in hybrid SDN. The idea behind their proposal was to collect the traffic information for a small subset of essential links and then to estimate the traffic information for the remaining links. They initially showed that traffic information in all links could be represented by what they called *basic links*. They created a method to identify the basic links in a network. Once the basic links had been identified, the next step was to connect the basic links to SDN switches and collect their traffic information from the SDN controller.

Lin et al. [64] proposed DTE-SDN, a TE engine to schedule the transfer of delay-sensitive traffic. DTE-SDN was implemented in the application layer using the northbound API to communicate with the SDN controller and contains three modules: (i) a topology-aware module that discovers the network topology, (ii) a monitoring module that, based on the network topology provided by the previous module, monitors the bandwidth and link delay of each network link, and (iii) a scheduling module that, based on the statistics collected by the monitoring module, performs multi-path routing of delay-sensitive traffic based on QoS metrics.

Shen [65] proposed a monitoring application, called SDN-monitor, to monitor selected OF switches with the aim of reducing resource consumption. His solution provides an algorithm divided into two phases. The first phase, the monitoring point selection phase, selects the switches to monitor by scanning all switches in the network and counting their flows. The algorithm sorts the switches in descending order according to their number of flows. Iteratively, the algorithm selects the switches with the highest number of flows, which collectively cover more flows. At the end of this process, the algorithm removes the covered flows from the unselected switches. The second phase, the flow re-routing phase, tries to combine the switches with the lowest re-routing cost to re-route the flows covered by the selected switches.

Wang and Su [66] proposed FlexMonitor, a flexible monitoring framework. FlexMonitor was implemented as an SDN controller module and contains four modules. The first module interprets requests from the upper applications and chooses an appropriate monitoring strategy based on application needs. The second module carries out monitoring tasks by deploying specific monitoring strategies in the network. The third module collects monitoring data from the network or hosts using various monitoring approaches. The last module aggregates the raw monitoring data coming from the previous module. FlexMonitor provides the following monitoring strategies: switch selection, OD host pair selection, and event definition.

Madanapalli et al. [67] designed and implemented a solution for detecting and monitoring elephant flows. Their solution contains three stages, of which the first one monitors traffic flows. The traffic flow monitoring stage is designed to be a ”bump-in-the-wire”[1] on the monitored link. A software inspection engine receives the packets that need to be inspected, thus protecting the SDN controller from overload, and the software inspection engine performs link monitoring.

Cohen and Moroshko [68] proposed a sampling-on-demand monitoring framework that enables the SDN controller to establish the sampling rate of each flow at every switch. The network operator sets this rate according to the monitoring goal. The proposed framework consists of three components. The first component, the *Sampling Management Module*, is deployed as an SDN controller module. The OF switches host the second component, which is the *Sampling Module*. The third component is a *Collecting Server* located in the network to collect and process the sampled packets. Their framework defines a new OF message called

[1]a communications device inserted into existing systems to enhance integrity, confidentiality, or reliability without altering the communications endpoints.

OFPT_RATE_MOD, which is sent by the *Sampling Management Module* to the switches. This message informs the switch of the sampling rate for each flow. The *Sampling Module* deployed in the switches samples the flows based on the specified rate, and the *Collecting Server* collects and processes the sampled packets.

Wang et al. [69] proposed Woodpecker, a system to detect and mitigate link-flooding attacks in SDN. To locate congested links, Woodpecker installs flow rules as measurement triggers in the OF switches. When a packet crosses the OF switch, the counters are updated, and the switch computes the byte rate of some specified flows based on the counter values. When the flow values activate the triggers, the switch sends a message to the controller.

Tsai et al. [70] proposed an adaptive flow measurement method to monitor SDN-based cellular core networks. Their proposed method starts observing the appearance and expiration of flows, and a table keeps track of the active flows. The next operation groups flows by their polling frequency, which can be set to static or dynamic. A log collector sends queries to collect flow status at a specific polling frequency and stores the retrieved data in a database. Lastly, a flow analyzer arranges and processes the stored data to provide useful information.

Tangari et al. [71] presented a self-adaptive and decentralized framework for resource monitoring in SDN. The framework defines a set of *Local Managers* (LMs), distributed over the network, with each LM monitoring a set of OF switches using the *Monitoring Module* (MM). Monitoring applications use the MM to add new monitoring requirements and to receive the measurement results. The MM uses a self-tuning adaptive monitoring approach to collect statistics from the switches. The goal of adaptive monitoring is to achieve a good trade-off between accuracy and resource consumption by continuously adapting the time T, i.e., the time between two consecutive measurements. The MM also stores the collected data in RAM and a database and provides a synchronization interface for the LM exchange messages.

CA Technologies [72] and ExtraHop [73] provide commercial SDN monitoring systems. The description of the features of their products provides a list of functionalities briefly highlighting some of the main monitoring components of the applications. The available documentation does not provide enough details to perform a comparison on how their products collect and generate the statistics on network resource utilization.

These previous efforts have provided a series of tools for SDN traffic monitoring. Despite their contribution, this research work has identified some gaps to be tackled. The tools presented so far are not based on a monitoring method, but on algorithms to solve specific monitoring problems such as monitoring flows and links. There is no process definition of how to collect, organize, and process the collected data. Some primary activities, such as storing the collected data for further processing and aggregating the generated statistics, are designed according to their proposed application. These tasks are part of all proposed algorithms but are performed in a variety of distinct ways. One of the main problems with this approach is the paucity of the results provided. The proposed algorithms cannot deliver monitoring at a fine-grained level, and the results do not provide many insights into network performance. Usually, one specific value is provided, such as throughput, link delay, or packet loss.

To overcome these problems, this research study has proposed a novel method for monitoring network traffic based on Big Data techniques. The proposed method establishes a process to collect counter data, store and process them to produce network statistics, and persist the obtained statistics.

One of the main benefits of Big Data techniques is that, unlike previous efforts that usually selected the resources to be monitored, the proposed approach focusses on collecting statistics from all active devices in the network. In this differentiated approach, once these statistics have been collected, a fine-grained statistical analysis of network resources, such as the ratio of each port to the switch load and the throughput capacity used on each port, path, and switch, is provided. This route-level aggregation provides an estimate of the network traffic matrix. All these throughput utilizations are estimated every three seconds; no other method has proven to be able to provide this information within this time constraint. Big Data streaming processing tools are used to provide near real-time statistics, which, to the best of our knowledge, is the first attempt to use this technology to generate this aggregation level.

This research study also identified that the values generated by previous monitoring applications were not used for traffic analysis, which provides a variety of information that can help to identify problems such as traffic congestion and trends. This research study also proposes a novel method for traffic analysis based on Big Data batch processing to scrutiny network resources utilization based on historical data.

3.2 Traffic Matrix Estimation

For traditional IP networks, several approaches have been used to estimate TM. Soule et al. [74] created a state space model to capture temporal and spatial correlations between a pair of hosts and Kalman Filters used to estimate and predict TMs. Xie et al. [75] proposed a Sequential Tensor Completion algorithm (STC) to recover missing Internet traffic data and also proposed a Reshape-Align scheme [25] to reshape inconsistent traffic matrices and align these matrices to form a tensor. Papagiannaki et al. [26] proposed a distributed measurement scheme based on limited use of flow measurement data and discussed the topic of traffic matrix estimation using direct measurement. They outlined the computation, communication, and storage overheads to generate traffic matrices at different granularity levels.

Yuan et al. [76] presented ProgME, a *Programmable MEasurement* architecture based on the concept of *flowset* (an arbitrary set of flows) to overcome the scalability challenges faced when measuring a large number of flows. ProgME can dynamically re-program the definitions of flow set based on user queries and can derive the traffic matrix. Guo et al. [77] proposed a framework to optimize the routing over multiple TMs in a hybrid SDN network. The basic idea was to cluster historical TMs and calculate the weight coefficient of every representative TM. After the weight coefficient calculation, an expected TM was calculated that combine the representative TM and the optimized OSPF weights over the expected TM.

The OF specification defines some flow counters that can be used to increase the accuracy of the TM estimate. Tootoonchian et al. [78] presented OpenTM, a traffic matrix estimator for OF networks. OpenTM was implemented as an application for the SDN controller to monitor active flows by polling the switches periodically. Their work also explored different algorithms for choosing which switch to query.

Yu et al. [79] proposed DCM, a per-flow monitoring system. DCM is based on bloom filters [80] and uses three steps to process packets in the data plane. The first step matches a packet with a wildcard monitor rule. The second step, called the admission bloom filter, represents the set of flows to be monitored, but does not have wildcard rules. The last step determines the corresponding monitoring actions for the flows in the admission bloom filter.

Polverini et al. [81] addressed the improvement of traffic matrix estimation. They developed a valid criterion based on the flow spread parameter, which is a way to find a subset of traffic to be measured to improve TM estimation. The flow spread parameter was used to characterize the weight of each flow traffic measurement in the TM estimation enhancement. Polverini et al. [82] also investigated an approach in which OF switches were introduced into a traditional IP network to understand how the new capabilities of these OF switches affected traffic matrix estimation. Their work showed that enhanced estimation accuracy was obtained by introducing only a few OF switches performing simple tasks.

Tian et al. [83] proposed a new framework for TM estimation in an SDN-based Ip network. Their work shows that if a flow's traffic can be derived from other flows in the network, adding a new entry for this flow for traffic measurement does not provide new information about the TM. Consequently, their traffic measurement scheme guarantees that when a controller selects a source-destination flow for measurement, the traffic for this flow cannot be derived from the existing flows in the flow table, improving the efficiency of the flow table.

OpenMeasure [84] provided an efficient inference framework based on adaptive measurement with on-line learning. OpenMeasure initially identifies the most informative flows and creates a set of rules to be installed in the switches. Using the controller's global view of the network, OpenMeasure dynamically decides what rules are to be installed and in which switches these rules will be deployed. The last action is to collect traffic statistics from switches to estimate the TM.

Maldoubi et al. [85] proposed SNIPER, a framework that combines SDN programmability with matrix completion techniques. SNIPER initially measures a subset of matrix entries for the attributes of interest (per-flow, size, delay, throughput, packet loss) directly, and then uses matrix completion techniques to estimate unobserved entries of the attributes of interest.

iSTAMP [86] used a well-compressed aggregated flow measurement and the K most informative flows to infer the TM. iStamp partitions the flow tables into two parts: the first for aggregate measurements, and the second for per-flow monitoring of selected flows. Using these two distinct parts, iStamp estimates the traffic matrix.

Gong et al. [87] proposed two strategies to design traffic measurement rules to be installed in the OF switch flow tables. They assumed that rules for routing flows in each SDN are aggregated whenever possible. These aggregated routing rules can be used to estimate the TM. In their work, the aggregated rules are disaggregated to improve estimation accuracy. Li et al. [88] developed a method to determine which flows are most informative and to construct a measurement flow set iteratively until an accuracy requirement is satisfied or a measurement resource constraint is reached.

Jiang et al. [89] developed an algorithm to estimate and recover the network traffic matrix at fine time granularity from sampled traffic traces. Their algorithm is based on fractal interpolation, cubic spline interpolation, and the weighted geometric average algorithm. Fractal interpolation describes the regularity of irregular systems. This theory was used to select the points and the time to obtain the samples and reconstruct the end-to-end network traffic. The curve provided by fractal interpolation is not smooth. Cubic spline interpolation is used to make the interpolation curve smoother. Because reconstruction errors are inevitable when using fractal and spline interpolation, the algorithm also uses the weighted geometric average algorithm to combine the previously mentioned interpolation approaches.

The previous studies mentioned earlier estimated the TM based on modelling and optimization procedures. For modelling, mathematical models are used to estimate the TM based on sampling mechanisms. This research has proposed an approach to the TM estimation problem using Big Data processing techniques. The MapReduce approach proposed here uses a direct measurement infrastructure to collect and process traffic data generated by the OF switches and, with the help of Big Data tools, tries to minimize the TM estimation error. To perform optimization, a search for the most informative flows is performed. The approach proposed here instead monitors the switch ports. Using this method, all traffic that crosses a link between an OD pair is used to estimate the TM.

3.3 Summary

This chapter has presented a review of academic research related to the contributions provided by this thesis. The first section reviewed the state of the art in research projects in the SDN traffic monitoring field. The last section reviewed research projects related to TM estimation because one of the contributions of this thesis is a new approach to TM estimation.

The next chapter describes the proposed Big Data traffic monitoring method.

Chapter 4

Big Data Traffic Monitoring Method

This chapter[1] presents the proposed Big Data trafffic monitoring method. It starts by establishing the resources monitored by the proposed method. Section 4.1 describes the proposed method with its associated activities using the BPMN notation on the process diagrams.

4.1 Introduction

An OF switch maintains several counters, each of which reflects its own traffic, and which are updated at each new packet. The Big Data traffic monitoring method probes the SDN controller periodically to obtain the counter values for the resources being monitored within the switch. Counters are maintained for several resources [91], but in the scope of this research, the following resources are monitored:

- Switch ports,
- Flow tables,
- Flow entries.

The basic values provided by the counters in an OF switch can be used to provide network bandwidth utilization at port and flow levels.

The Big Data traffic monitoring method includes three main activities, as depicted in Figure 4.1:

1. **Data Acquisition**. Loads network inventory and traffic data from the control and data planes by means of the SDN controller.

[1]The content of this chapter has been published as a journal paper [90].

2. **Data Aggregation**. Processes the acquired data, calculating network parameters [17] in near real time.

3. **Data Persistence**. Stores and provides network inventory and parameters to traffic analysis systems.

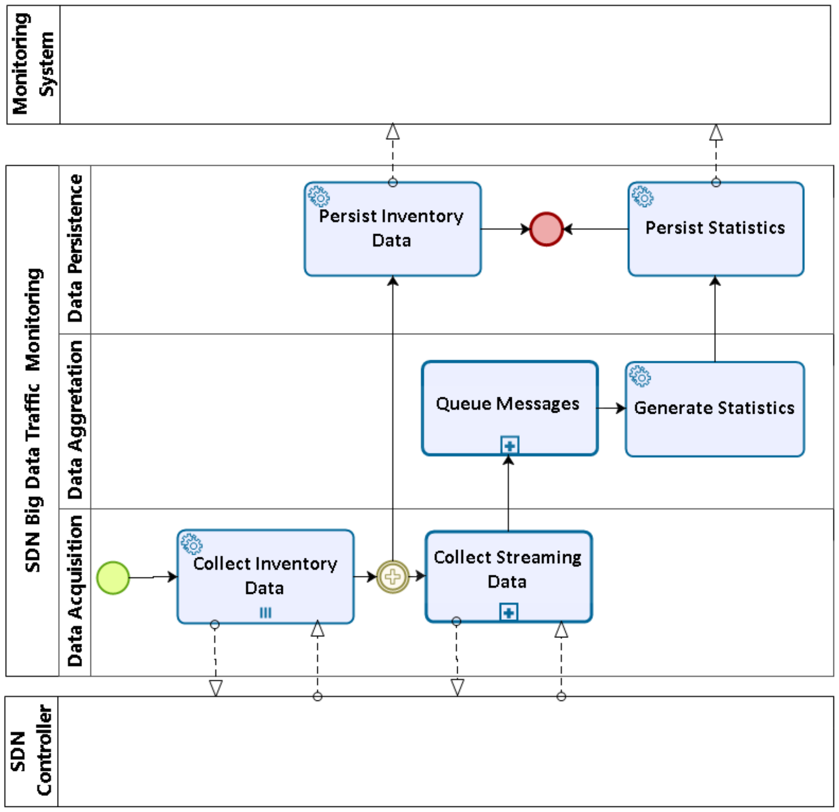


Figure 4.1: Big Data traffic monitoring method.

These three activities are further detailed in the following sub-sections.

4.2 Data Acquisition

The Data Acquisition activity collects all the data needed in the Big Data traffic monitoring method. The collected data can be categorized as inventory and streaming data. The inventory data populate a repository, whereas the streaming data are collected and immediately sent for further processing. Inventory data provides basic data about network resources and can be used for different types of functionalities depending on the implementation. For instance, to calculate the percentage of its own available bandwidth that a port is consuming, data about the available bandwidth are stored in the inventory data. Figure 4.2 shows the entity-relationship

(ER) model for the inventory data repository. Inventory data are kept in the *Host*, *Interface*, *Switch Port*, *Switch*, *FlowTable*, and *Flow* entities. Inventory data can be defined as data that are less prone to changes over time because they are related to the basic identification and configuration of the network hardware (hosts and switches). Streaming data are related to the counters provided by the OF devices. According to the OF specification, these counters are updated at every packet that crosses the switch, and therefore, their rate of change is higher than that of inventory data. If these counters are used for monitoring or any other type of analysis, their values must be collected periodically. This requirement suggests that these data should not be persisted in any repository and, once collected, should be immediately forwarded for processing.

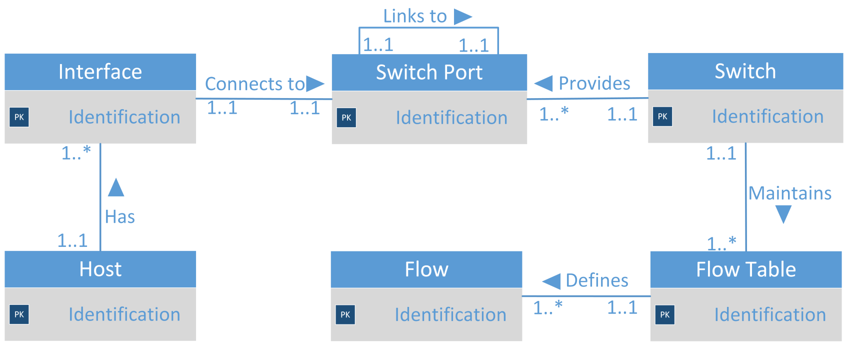


Figure 4.2: ER model for inventory data.

The description of the entity types depicted in Figure 4.2 is given below :

- **Host**. Identifies all computers connected to a network.
- **Interface**. Describes all the network interfaces available in a network.
- **Switch Port**. Describes all the ports available in a switch.
- **Switch**. Represents all the switches in a network.
- **Flow Table**. Represents all flow tables available in a switch.
- **Flow**. Describes all flows created in a switch.

The model presented in Figure 4.2 identifies the following relationships between entities:

- **Has**. A *Host* can have installed and configured several network *Interfaces* to communicate over the network. An *Interface* is connected to only one host. It is worth mentioning that, even though the multiplicity of this relationship is defined as 1 x N, the actual number of interfaces that a host can have installed is bounded by hardware limitations.

- **Connects to**. An *Interface* is connected to only one *Port*, and a *Port* can be connected to only one *Interface*.
- **Links to**. This recursive relationship implies that a *Port* can be connected to only one other *Port*.
- **Provides**. A *Switch* can provide one or several *Switch Port*s to the network, and a *Switch Port* can be provided by only one *Switch*.
- **Maintains**. A *Switch* can maintain one or several *Flow Table*s, and a *Flow Table* is maintained by only one *Switch*.
- **Defines**. A *Flow Table* defines one or several *Flow*s, and a *Flow* is defined by only one *Flow Table*.

The attributes that describe each entity occurrence in the *Host* and *Interface* entities are not defined by the OF specification. These attributes are SDN controller-dependent and hence are not described in this section. The only attribute defined for these entities is the *identification*, which uniquely identifies one occurrence of an entity. This attribute is set as primary key in both *Host* and *Interface* entities. Table 4.1 provides the list of attributes defined for the *Switch* entity according to the OF specification.

Switch	
Attribute	**Description**
Identification	Switch unique identification in the network
MAC address	Switch's MAC Address
Buffers	Maximum packets buffered at once
Tables	Number of flow tables supported by the switch
Capabilities[1..9]	Capabilities supported by the switch
	Ex. Provide Flow Statistics
	Provide Port Statistics

Table 4.1: Attributes for the switch entity.

The list of attributes defined for the *Switch Port* entity in the inventory category is shown in Table 4.2.

The basic attributes for the *Flow Table* and *Flow* entities are their identification. Any other attributes provided by the SDN controller can be added.

- **Collect Inventory Data**

  The goal of this task is to collect an inventory of the network. In an OF network, the inventory is composed of hosts and switches. This task also collects information about

Switch Port	
Attribute	**Description**
Identification	Port unique identification within the switch
Hardware Address	Port MAC Address
Name	Human-readable name
Configuration	Port administrative settings
State	Port internal state
Current Feature	Current port speed and duplexity
Supported Features	Supported port speed and duplexity
Peer Features	Speed and duplexity advertised by the peer
Advertised Features	Port advertised speed and duplexity
Current Speed	Current bit rate in kbps
Maximum Speed	Maximum bit rate in kbps

Table 4.2: Attributes for the switch port entity.

the interconnections between hosts and switches. These interconnections are represented by the relationships *Connects to* and *Links to*, as presented in Figure 4.2.

The diagram presented in Figure 4.2 shows the relationship *Provides* between *Switch* and *Switch Port* entities. This relationship implies the dependency of the *Switch Port* entity on the *Switch* entity. The same applies to the relationship *Maintains* between *Switch* and *Flow Table* entities. The relationship *Defines* between *Flow Table* and *Flow* implies dependency of the *Flow* entity on the *Flow Table* entity. Based on these relationships, when this task collects switch inventory data, it also collects port inventory data, flow tables, and flow identifications. The pseudocode for this task is provided in Algorithm 4.1.

Algorithm 4.1: Collect Data algorithm

```
1 Procedure CollectData()
2     while TRUE do
3         CollectHostInventory()
4         CollectSwitchInventory()
5         CollectLinkInventory()
6         update Host, Switch, Switch Port, Flow Table, Flow repositories
7         sleep(n units of time)
8     end
```

This task regularly probes the SDN controller to update network inventory (lines 3–5). This periodic operation will keep the repository updated with the most recent network configuration. The time interval between two consecutive updates is specified by the variable *n* (line 7).

- **Collect Streaming Data**

The Collect Streaming Data activity, shown in Figure 4.3, uses the controller's northbound APIs to collect data about the counters of the monitored resources in each switch. This activity defines six tasks: namely, *Start Collect Counters*, *Collect Port Counters*, *Collect Flow Table Counters*, *Collect Flow Counters*, *Format Data*, and *Send Data*, which are described in the following sub-sections.

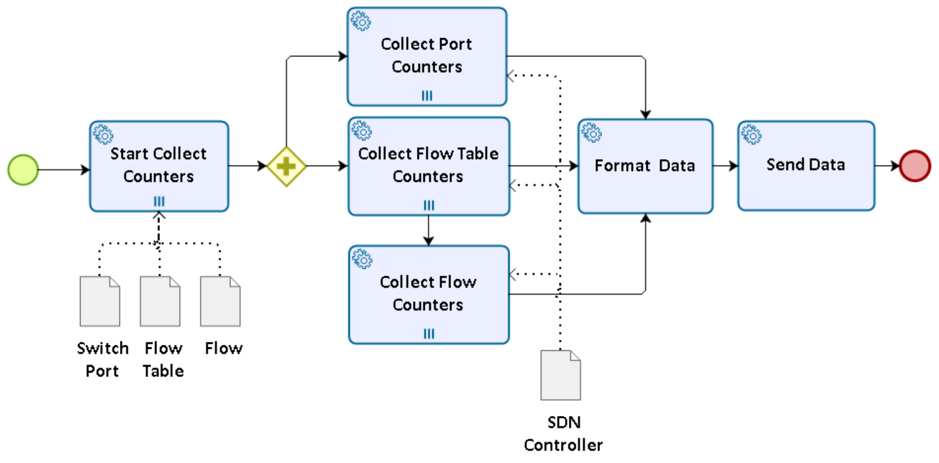


Figure 4.3: Collect streaming data.

- **Start Collect Counters**

  This task periodically queries the inventory to start collecting counters for ports and flow tables. For each port and flow table, a collect task is started. The number of counter collectors is given by Equation 4.1:

$$C = \sum_{S=1}^{n} (p + ft), \tag{4.1}$$

  where C is the number of collectors, n the number of switches, p the number of ports in the switch, and ft the number of flow tables in the switch.

  The pseudocode for this task is shown in Algorithm 4.2. As this task continually reads the inventory repository, it is expected that between two or more readings, the same port and flow table will be in the repository and that only one task has to be initiated for collecting the counters. To avoid more than one task collecting counters for the same resource, lines 5 and 11 provide an **if** statement to ensure this constraint. For instance, if port *eth0* is present in two or more updates of the repository, only one task will be collecting its counters. The same is valid for flow tables.

- **Collect Port Counters**

Algorithm 4.2: Start Collect Counters Algorithm

```
Procedure StartCollectCounters()
    while TRUE do
        read Switch Port repository
        for each switch port do
            if collect counters not started then
                start collect port counters task
            end
        end
        read Flow Table repository
        for each flow table do
            if collect counters not started then
                start collect flow table counters task
            end
        end
        sleep (n units of time)
    end
```

This task collects counter data for switch ports. The collected attributes are described in Table 4.3.

Port Counters	
Attribute	**Description**
Timestamp	Instant when the reading was collected
Packets Received	Number of received packets
Packets Transmitted	Number of transmitted packets
Bytes Received	Number of received bytes
Bytes Transmitted	Number of transmitted bytes
Collision Count	Number of collisions
Over RunError Received	Number of packets with RX overrun
Drops Transmitted	Number of packets dropped by TX
Drops Received	Number of packets dropped by RX
Frame Error Received	Number of frame alignment errors
CRC Error Received	Number of cyclic redundancy check errors
Seconds	Number of seconds the port has been installed
Nanoseconds	Number of nanoseconds in a second

Table 4.3: Attributes for port counters.

The pseudocode for this task is provided in Algorithm 4.3.

The *CollectPortCounters(switch,port)* (line 3) function probes the SDN controller every *n* (line 5) time units to obtain the counter value for the port. The switch and port parameters identify the specific port for the SDN controller.

Algorithm 4.3: Collect Port Counters Algorithm

```
1 Procedure CollectPortCounters()
2     while TRUE do
3         CollectPortCounters(switch,port)
4         send counters to Format Data
5         sleep(n units of time)
6     end
```

- **Collect Flow Table Counters**

  This task collects flow table counter data. The collected attributes are described in Table 4.4.

Flow Table Counters	
Attribute	**Description**
Timestamp	Instant in time when the reading was collected
Active entries	Number of active flows
Packet Lookups	Number of packets looked up in the table
Packet Matches	Number of packets that matched in the table

Table 4.4: Attributes for flow table counters.

Because there is a dependency of the entity *Flow* on the *Flow Table* entity (relationship *Defines* in Figure 4.2), this task starts a counter collector for each flow. The number of flow counter collectors is given by Equation 4.2.

$$Cf = \sum_{S=1}^{n} \sum_{ft=1}^{m} f, \tag{4.2}$$

where Cf is the number of flow counter collectors, n is the number of switches, m is the number of flow tables in the switch, and f is the number of flows in the flow table.

Algorithm 4.4 provides the pseudocode for this task.

The *CollectFlowTableCounters(switch,flow table)* (line 3) function probes the SDN controller every n (line 10) time units to obtain the counter values for the flow table. The switch and flow table parameters identify the specific flow table for the SDN controller.

- **Collect Flow Counters**

  This task collects flow counter data. The collected attributes are described in Table 4.5.

  The pseudocode for this task is shown in Algorithm 4.5.

Algorithm 4.4: Collect Flow Table Counters Algorithm

```
Procedure CollectFlowTableCounters()
    while TRUE do
        CollectFlowTableCounters(switch,flow table)
        send counters to Format Data
        for each flow in the flow repository do
            if collect counters not started then
                start collect flow counters task
            end
        end
        sleep(n units of time)
    end
```

Flow Counters	
Attribute	**Description**
Timestamp	Instant in time when the reading was collected
Received Packets	Number of received packets
Received Bytes	Number of received bytes
Seconds	Number of seconds that the flow has been installed
Nanoseconds	Number of nanoseconds in a second

Table 4.5: Attributes for flow counters.

Algorithm 4.5: Collect Flow Counters Algorithm

```
1 Procedure CollectFlowCounters()
2     while TRUE do
3         CollectFlowCounters(switch,flow table,flow)
4         send counters to Format Data
5         sleep(n units of time)
6     end
```

The *CollectFlowCounters (switch, flow table, flow)* (line 3) function probes the SDN controller every *n* (line 5) time units to obtain the counter values for the flow. The parameter switch, flow table, and flow identify the specific flow for the SDN controller.

- **Format Data**

All collected counter data are used for further processing. It is expected that the format in which the data are collected is not the format in which they will be processed. It is also expected that the layout in which they are collected is not the layout needed for processing. This task formats the data before they are sent for processing. The format is implementation dependent.

The pseudocode for this task is shown in Algorithm 4.6.

Algorithm 4.6: Format Data Algorithm

```
1 Procedure FormatData()
2     while TRUE do
3         receive counters data
4         format data
5         forward data to the send data task
6         sleep(n units of time)
7     end
```

- **Send Data**

  This task receives the formatted data from the *Format Data* task, tags it as a switch port, flow table, or flow counters and forwards it to the *Queue Messages* sub-process.

  As shown in Figure 4.4, a switch has ports and flow tables, and each flow table has flows. For each port, flow table, and flow, the counter collectors' tasks (*Collect Port Counters*, *Collect Flow Table Counters*, *Collect Flow Counters*) probe the SDN controller to obtain the data. This procedure is present in line 3 in algorithms 4.3, 4.4, and 4.5. The same procedure is also present in lines 3, 4, and 5 in Algorithm 4.1. The procedure in the counter collectors' tasks is more computationally intensive for two main reasons: (1) the number of items being monitored is much grater, and consequently, (2) the number of requests to the SDN controller is also much greater. The rate of requests can be higher or lower depending on the time interval between requests.

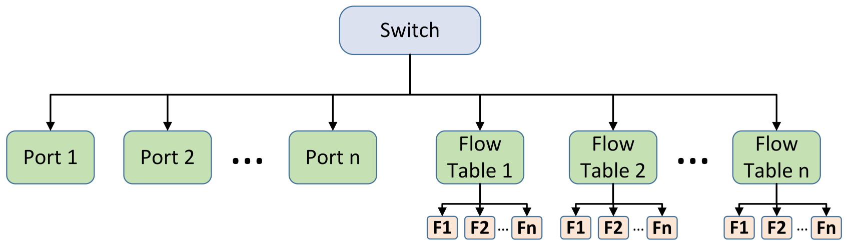


Figure 4.4: Switch hierarchy.

The time interval between requests is defined by the *sleep* statement present in all algorithms mentioned in the previous paragraph. The value for *n* in Algorithm 4.1 must be greater than the one in the counter collectors. As mentioned in Section 4.2, inventory data tend to be relatively constant over time. The number of hosts and their configuration do not change very frequently, and the same applies to switches and network connections. On the other hand, counter data constantly change due to the number of packets that traverse the switch. The number of requests sent to the SDN controller in one iteration of the while loop in Algorithm 4.1 is three. This number assumes that the SDN controller provides one request for each resource type (Hosts, Switches, or Links), but this number depends on the northbound API provided by the controller. For instance, one controller may return all the data with only one request. When the controller provides several options for requesting the data, the choice of which option to use may be based on the request latency. A request that returns all the data may have higher latency, whereas a request for each resource type may have lower latency, but more requests are required. This trade-off must be addressed based on the chosen SDN controller.

4.3 Data Aggregation

This activity receives the data collected by the *Data Acquisition* activity, processes it generating the expected statistics and sends the outcome to be stored. The process executed by this activity defines the sub-processes *Queue Messages* and *Generate Statistics*, which are described next.

4.3.1 Queue Messages

The goal of this sub-process is to guarantee that the messages received from the *Data Acquisition* activity are stored and available for further processing as they are requested. Messages are stored and forwarded in the same order that they arrive. The tasks shown in Figure 4.5 operate based on this requirement.

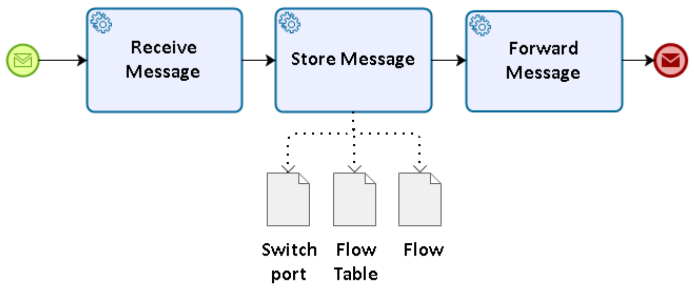


Figure 4.5: Queue messages process.

- **Receive Message**

  This task receives the messages coming from the *Data Acquisition* activity. The arriving messages refer to switch port, flow table, and flow counters and must be identified as such and forwarded for storage.

- **Store Message**

  This task stores all messages according to their tag (port, flow table, or flow) and provides them as requested with the guarantee that messages are being provided according to their arrival order and no message is missing.

- **Forward Message**

  The goal of this task is to retrieve and forward the stored messages as they are requested for further processing.

4.3.2 Generate Statistics

Statistics are periodically generated based on the available counter data stored by the previous sub-process. This task requests these data (messages), processes them by computing the expected network parameters, and sends the results to storage. The pseudocode for this task is presented in Algorithm 4.7.

Algorithm 4.7: Generate Statistics Algorithm

```
Procedure GenerateStatistics()
    while TRUE do
        request data
        ComputeStatistics()
        send statistics
        sleep(n units of time)
    end
```

Messages are processed in message sets. Each message set is composed of all messages that arrived in the *Queue Messages* sub-process during one statistical computation. Figure 4.6 shows the *Generate Statistics* procedure. During processing of Batch 1, the messages that compose Batch 2 arrive in the *Queue Messages* sub-process. At the end of Batch 1 processing, the computed statistics are sent for storage. During processing of Batch 2, the messages that compose Batch 3 are still arriving. This is a continuous process. The *Compute Statistics* function computes statistics and aggregates them according to its purpose, which can be, for instance, aggregation by switch, port, and flow table. The aggregated element and the network parameters are application-dependent.

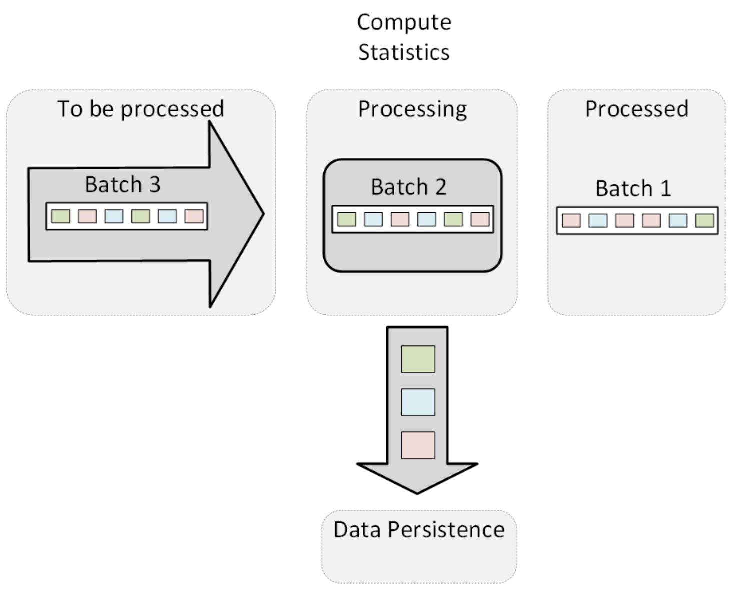


Figure 4.6: Generate statistics procedure.

4.4 Data Persistence

The collected inventory and generated network parameters are persisted in this activity. These data can be used by any monitoring, traffic analysis, or visualization system.

- **Persist Inventory Data**

  This task persists the inventory of the network. These data can be used to recreate the network topology by identifying all its components.

- **Persist Statistics**

  The network parameters generated by the *Data Aggregation* activity are persisted by this activity. It is a complex task to synchronize the moment when the system requests these parameters and the moment when they are available. To overcome this situation, the network parameters are persisted in a repository and are available when a consuming system needs the latest computed values. The premise of a real-time monitoring system significantly increases the speed at which these parameters are updated. Any client system needs to request the stored values at a rate that is compatible with the rate that the parameters are updated in the repository, with the risk of missing some updates.

4.5 Summary

This chapter presents the Big Data traffic monitoring method and describes its three main activities. Section 4.2 introduces the activity of collecting inventory and traffic data along with a persistence model of the collected data. Section 4.3 details the data aggregation activity. Section 4.4 describes the persistence of the collected and generated data from Sections 4.2 and 4.3.

The next chapter presents an implementation to validate the proposed monitoring method. The chapter describes the implementation components and presents the component deployment diagram.

Chapter 5

Implementation

This chapter presents an implementation of the Big Data traffic monitoring method and its components. Section 5.1 provides the design of the interconnection between the components and some details of the SDN controller used to manage the network. The components are also described, and the deployment diagram of the components is provided, along with the environment that receives the deployment. The deployment diagram also identifies the Big Data tools used in the implementation.

5.1 Big Data Traffic Monitoring Implementation

This section describes a Big Data traffic monitoring implementation based on the method proposed in Chapter 4. The designed components implement each activity and each task of the Big Data traffic monitoring method presented in Figure 4.1. It is worth mentioning that the proposed *Collect Flow Table* and *Collect flow* components have not been implemented because the primary focus of the present implementation is on port throughput monitoring.

The implementation components are depicted in the component diagram presented in Figure 5.1. The assembly connectors specify the following links:

- **Inventory**. The *collect streaming data* component requires an interface to obtain the topology data, and the SDN controller provides this interface.
- **Counters**. The *collect streaming data* component requires an interface to obtain the Counters, and the SDN controller provides the required interface.
- **Add message**. The *collect streaming data* component needs an interface to send the collected inventory and counter data to the queue, and the *queue messages* component provides the necessary interface.

- **Get message**. The *generate statistics* component retrieves the latest messages added to the queue, and the *queue messages* component provides the appropriate interface.

- **Insert**. After generating statistics, the *generate statistics* component inserts them to the *persist statistics* component using the interface provided.

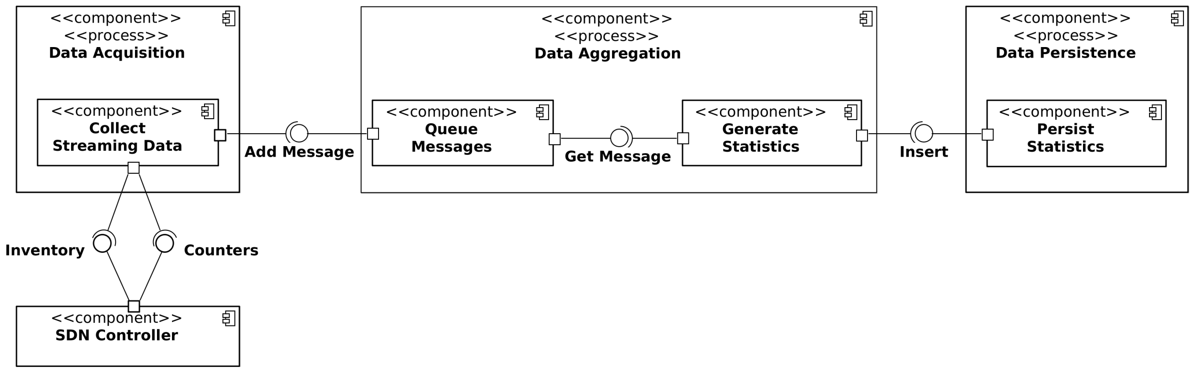

<<component>> <<process>> Data Acquisition
<<component>> Collect Streaming Data
Add Message
<<component>> <<process>> Data Aggregation
<<component>> Queue Messages
Get Message
<<component>> Generate Statistics
Insert
<<component>> <<process>> Data Persistence
<<component>> Persist Statistics
Inventory
Counters
<<component>> SDN Controller

Figure 5.1: Big Data traffic monitoring components.

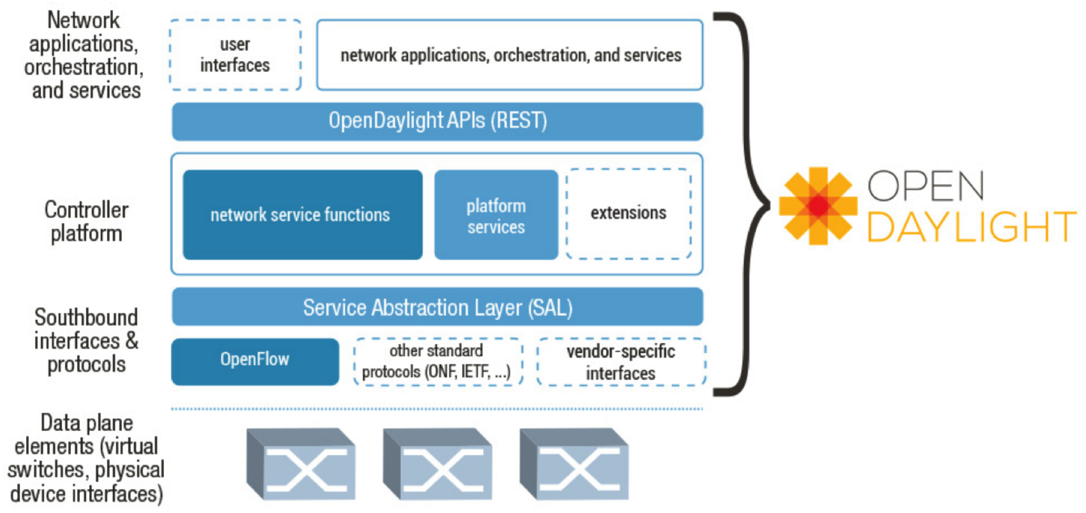
The following sections describe each component presented in Figure 5.1.

5.1.1 SDN Controller

The SDN controller used for the Big Data traffic monitoring implementation is OpenDaylight [9, 20, 22]. This controller is an open-source platform hosted by the Linux foundation and sponsored by industry leaders such as IBM, Cisco, HP, Microsoft, Red Hat, and VMware. These companies provide economic and engineering support for the development of the platform. The OpenDaylight architecture is presented in Figure 5.2.

OpenDaylight is an application that runs on any operating system with a Java Virtual Machine (JVM) and provides a multi-layer architecture. The primary layer is the *Controller Platform*, which manages switch flow traffic using flow tables. On the *Southbound Interfaces and Protocols*, OpenDaylight can support multiple standard protocols defined by standardization organizations such as IETF and ONF. Many contributors can add new protocols as modules due to the multi-layer architecture. The *Service Abstraction Layer* (SAL) links new modules dynamically. The OSGI framework provides a dynamic link between SAL and evolving Southbound protocols.

The controller supports the OSGI framework and bidirectional REST for the Northbound API. This feature enables new modules to be added to the *Controller Platform* with their respective APIs for network applications.



Network applications, orchestration, and services
user interfaces
network applications, orchestration, and services
OpenDaylight APIs (REST)
Controller platform
network service functions
platform services
extensions
Southbound interfaces & protocols
Service Abstraction Layer (SAL)
OpenFlow
other standard protocols (ONF, IETF, ...)
vendor-specific interfaces
Data plane elements (virtual switches, physical device interfaces)
OPEN DAYLIGHT

Figure 5.2: OpenDaylight architecture [9].

The Big Data traffic monitoring implementation belongs to the *Network applications, orchestration, and services* layer, and makes use of the REST APIs provided by OpenDaylight. The required and provided interfaces are *Inventory* and *Counters* (Figure 5.1), which are implemented by the REST APIs in the controller.

5.1.2 Data Acquisition

This component implements the data acquisition activity (Figure 4.1) and provides the collect streaming data component to implement the collect inventory data and collect streaming data sub-processes.

Collect Streaming Data

This component collects inventory and counters data using the *Inventory* and *Counters* interfaces provided by OpenDaylight. The algorithms described in section 4.2 are implemented in this component.

Algorithm 4.1 updates three repositories: *Hosts*, *Switches*, and *Links*. Internally each of these repositories is stored in main memory as a *HashMap*, as shown in Figure 5.3.

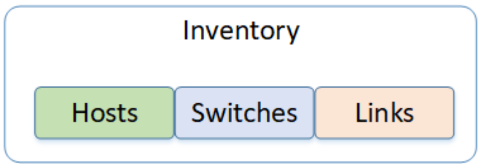


Figure 5.3: Repositories.

OpenDaylight provides a series of attributes for *Hosts*, and Table 5.1 lists the ones stored in the repository.

Attribute	**Description**
Identification	Host unique identification in the network
<List of Interfaces>	List of all network interfaces

Table 5.1: Host attribute description.

The *List of interfaces* attribute is a list that identifies all the network interfaces that are configured in a host. Table 5.2 lists the attributes that identify a single network interface.

Attribute	**Description**
IP	IP address configured for the network interface
MAC	MAC address of the network interface

Table 5.2: Network interface attribute description.

For the *Switch* repository, in addition to the attributes listed in Table 4.1, OpenDaylight provides the attributes described in Table 5.3. Table 4.2 lists the attributes defined for each port.

Attribute	**Description**
<List of ports>	List of ports available in the switch
<List of flow tables>	List of flow tables available in the switch
<List of flows>	List of current flows in the switch

Table 5.3: Switch attribute description.

The attributes provided for *Links* are listed in Table 5.4.

Links can be established between hosts and switches and between switches. For each link, OpenDaylight identifies the termination points. If the source/destination node is a host, the termination point provides the network card. If the source/destination node is a switch, the termination point provides the port number.

Attribute	**Description**
Identification	Link unique identification
Source Node	Source point of communication
Source Node Termination Point	Termination point in the source node
Destination Node	Destination point of communication
Destination Node Termination Point	Termination Point in the destination node

Table 5.4: Link attribute description.

The following procedures are executed after the repositories are populated:

1. The inventory data are formatted and sent to the *Queue Messages* component.

2. For every switch port, flow table and flow, counters are collected. This procedure follows the switch hierarchy presented in Figure 4.4 in the following way:

   - Collects counters for each switch port (Algorithms 4.2 and 4.3). Table 4.3 lists the attributes.
   - Collects counters for each flow table (Algorithms 4.2 and 4.4). Table 4.4 lists The attributes.
     - Collects counters for each flow in the flow table (Algorithms 4.4 and 4.5). Table 4.5 lists the attributes.

   Threads are used to collect the counters mentioned above. For each element (port, flow, or flow table), one thread is started. The total number of threads is given by Equation 5.1:

$$t = \sum_{S=1}^{n} p + ft + f \tag{5.1}$$

   where t is the total number of threads, n is the number of switches, p is the number of ports in the switch, ft is the number of flow tables, and f is the total flows.

Each message has a timestamp added to identify the time when the message was collected. The messages related to counters have a switch identification field added.

Figure 5.4 shows the packages that implement the *Collect Streaming Data* component.

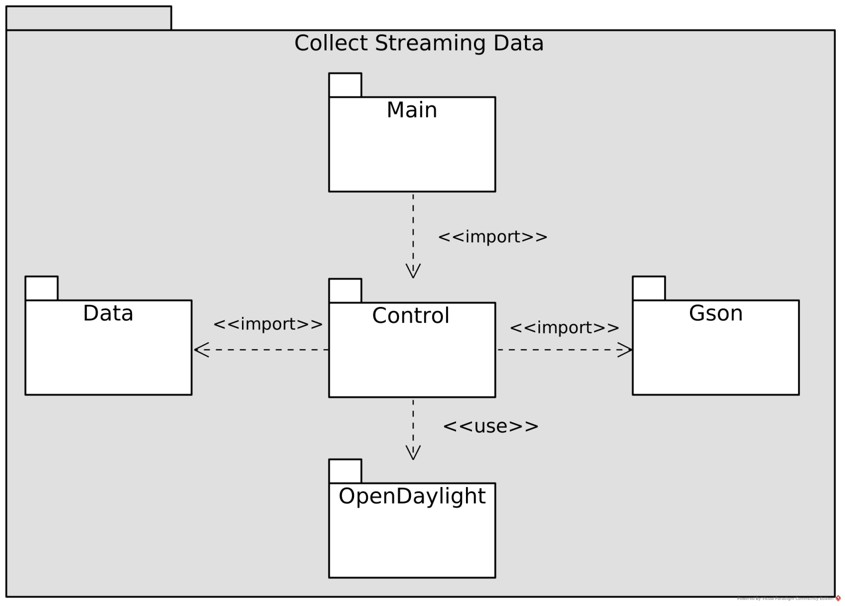


Figure 5.4: Collect streaming data package.

The *Main* package sets the initial configuration of the implementation and starts the collect data manager in the *Control* package. The *Control* package sends requests to OpenDaylight via REST APIs (Links *Inventory* and *Counters* in Figure 5.1) to populate the three repositories. The inventory data are immediately formatted and sent to the *Queue Messages* component. For the counter data, the manager starts all the threads, and each thread sends the collected counters to the *Queue Messages* component. These values are not persisted in the repositories. All the collected data are sent using the *Add Message* link (Figure 5.1).

OpenDaylight replies to requests by sending messages in JSON format. The *Gson* package provides all the classes that represent each of the response messages. These messages are instantiated in the *Data* package for further formatting and sending.

5.1.3 Data Aggregation

The *Data Aggregation* component receives the messages from the *Collect Streaming Data* component and generates statistics that describe the network traffic.

Queue Messages

Apache Kafka manages the messages generated by the *Collect Streaming Data* component. In Kafka's terms, the *Collect Streaming Data* component is defined as a *producer*. Because Kafka works with the concept of topics, the following topics were created to accommodate the incoming messages:

1. **Host**. Messages related to host description (Table 5.1).
2. **Switch**. Messages related to switch description (Table 4.1).
3. **Link**. Messages related to link description (Table 5.4).
4. **SwitchPort**. Messages related to port description (Table 4.2).
5. **Traffic**. Messages related to switch port counters (Table 4.3).
6. **FlowTable**. Messages related to flow table description (Table 4.4).
7. **Flow**. Messages related to flow counters (Table 4.5).

Generate Statistics

This component contains two packages, as shown in the package diagram presented in Figure 5.5.

The *Inventory* package processes the inventory messages coming from the topics *Host*, *Switch*, *SwitchPort*, *Switch*, and *Link*.

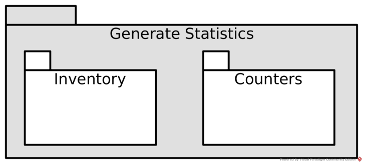


Figure 5.5: Generate statistics package.

The *Counters* package processes the streaming counter messages and generates the statistics. Apache Spark streaming is used to process these messages coming from the *Traffic*, *FlowTable*, and *Flow* topics.

An OF switch does not provide the throughput for each port or flow, this value can be calculated using the values present in the message by means of the formula in Equation 5.2, where T is the throughput, BR is the number of bytes received, BT is the number of bytes transmitted, and S the number of seconds:

$$T = \frac{(BR + BT)}{S} \tag{5.2}$$

A *driver program* (*Counters* package) launches the Map/Reduce operations to compute throughput with the following keys:

- <Switch, Port, Timestamp>. The reducer generates the throughput for each port.
- <Switch, Timestamp>. The reducer generates the throughput for each switch.
- <Timestamp>. The reducer generates the throughput for the entire network.
- <Switch, Port, FlowTable, Flow, Timestamp>. The reducer generates the throughput for each flow.
- <Switch, Port, FlowTable, Timestamp>. The reducer generates the throughput for each flow table.

5.1.4 Data Persistence

This component persists the statistics generated by the *Data Aggregation* component.

Persist Statistics

The Elasticsearch database stores all the generated statistics. This component consists of a series of classes used as an interface between the *Generate Statistics* component and the database. Using the basic concepts of Elasticsearch, the following indexes were created: *Host*, *Link*, *Switch*, *SwitchPort*, *FlowTable*, *Flow*, and *Traffic*, *SwitchStats*, and *NetworkStats*. These classes provide the methods to carry out all the necessary manipulation on the listed indices. These classes are implemented in the *Counters* package (Figure 5.5).

5.1.5 Environment

Four machines were used to deploy the implementation. They are described in Table 5.5.

Id	**CPU**	**RAM**
Server 1	2 x Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz	96 GB
Server 2	2 x Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz	96 GB
Server 3	2 x Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz	96 GB
Server 4	Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz	16 GB

Table 5.5: Hardware environment.

Servers 1, 2, and 3 are connected to an HP MSA 2040 SAN 12 TB storage system. The storage was divided into four logical partitions of 2.5 TB each.

All servers run Linux Ubuntu Server 16.04, and the same is valid for all configured containers. A feature called *Multipath* was configured in the servers, enabling them to have multiple I/O paths to storage, which enables parallel access to the four configured partitions.

5.1.6 Deployment

All packages developed in the implementation were coded using the Java language. The following Java Archives (JAR) were built:

- CollectStreamingData.jar. This file contains the implementation of the *Collect Streaming Data Package* (Figure 5.4).
- Inventory.jar. This file contains the implementation of the *Inventory* package (Figure 5.5).
- Counters.jar. This file contains the implementation of the *Counters* package (Figure 5.5).

From the description above, it is important to note that only one JAR file was generated for the *Collect Streaming Data* package, but that for the *Generate Statistics* package ,the *Inventory* and *Counters* files were generated. This difference in the number of files is because the *Counters* file is a Spark application, but the *Inventory* file is not.

The entire implementation was deployed according to the diagram shown in Figure 5.6. The Linux Server 1 hosts the *CollectStreamingData* JAR file and the Kafka containers; Linux server 2 hosts the *Inventory* JAR file and the Elasticsearch containers; Linux Server 3 hosts

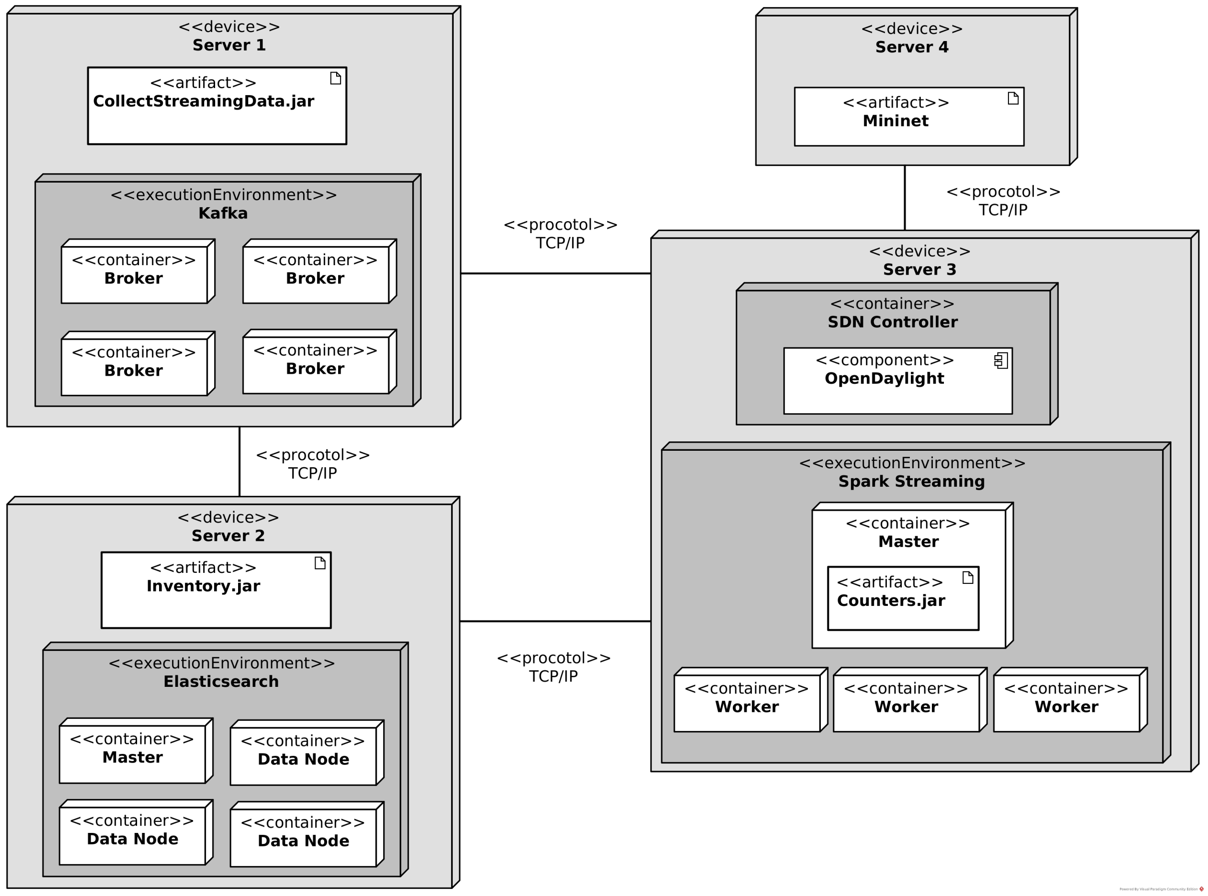


Figure 5.6: Big Data traffic monitoring deployment.

OpenDaylight and the Spark streaming containers; and Server 4 hosts the SDN network simulator (Mininet).

As mentioned in section 5.1.5, all servers (except Server 4) can connect in parallel to the four partitions in storage. This feature enables each container to access all partitions in parallel allowing the use of containers to simulate a cluster of servers.

5.2 Summary

This chapter has described the components of the Big Data traffic monitoring method implementation and has also presented the deployment environment. The deployment diagram provides the Big Data tools used in the implementation: Apache Kafka as the *Queue Message* component, Apache Spark Streaming as the *Generate Statistics* component, and Elasticsearch as the *Persist Statistics* component.

The implementation of the Big Data traffic monitoring method provided a considerable number of statistics reflecting network traffic as highlighted in Chapter 8. One of these statistics is provided by TM, which is one of the main contributions of this thesis. The next chapter introduces the proposed MapReduce method for TM estimation.

Chapter 6

MapReduce Traffic Matrix Estimation Method

This chapter presents the proposed MapReduce method to estimate the TM. The estimation of the TM is in the Big Data traffic monitoring method's implementation scope, as described in section 6.1. The chapter starts by describing the estimation method's MapReduce functions. The implementation of the proposed method requires the definition of some basic structures, which are presented in section 6.2. Section 6.2.1 describes the algorithms that implement the TM estimation method.

6.1 MapReduce Design

The MapReduce approach consists of four map functions and one reduce function, as shown in Figure 6.1.

The *Map Raw Data* function receives raw data as messages and generates a *key/value* map, where the timestamp of each message is the *key* and the data needed to calculate the throughput are the *value*.

The *Sort By Key* function sorts the keys generated by the *Map Raw Data* function in ascending order. This step is necessary to calculate the current link throughput. The current link throughput is used to calculate the current throughput between OD pairs of hosts.

The map functions shuffle the generated maps before the next function processes them. The *Sort By Key* function generates a map that must be processed in ascending order by timestamp. The function *Coalesce* prevents the shuffling in the map produced by the *Sort By Key* function.

The *Generate Final Map* function performs two tasks: (i) computing the current throughput in every link, and (ii) computing the accumulated throughput in every link.

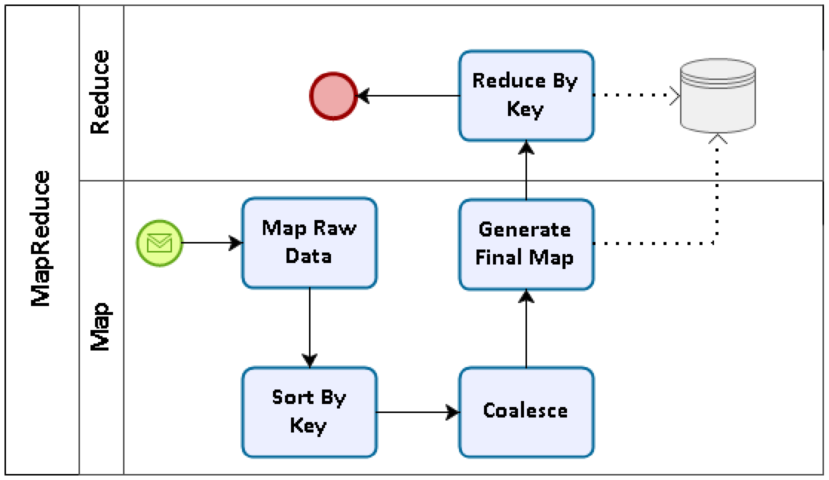


Figure 6.1: MapReduce design.

The *Reduce by Key* function, which generates and persists the TM, processes the final map.

6.2 Basic Definitions

The first goal of the MapReduce approach is to identify and generate the *key*s that will be aggregated and subsequently reduced. This procedure is part of the Map portion of the proposed design (Figure 6.1). The proposed approach establishes that the *key*s will be composed of all links in the route between OD pairs. To achieve this goal, all the necessary elements that will be used to generate the *key*s must be identified. This section introduces the basic definitions of all these elements and the structures that relate them.

The definitions are as follows:

- Set $N = \{x_1, x_2, \cdots, x_n\}$. This set represents the nodes network.
- Set $H = \{h_1, h_2, \cdots, h_m \mid h_i \in N\}$. $H \subset N$. This set represents all hosts in the network.
- Set $S = \{s_1, s_2, \cdots, s_k \mid s_i \in N\}$. $S \subset N$. This set represents all switches in the network. Each switch s_i has several ports, which can be represented as $s_{i1}, s_{i2}, \cdots, s_{il}$.
- Set $U_1 = \{(h_i, s_j) \vee (s_j, h_i) \mid h_i \in H \wedge s_j \in S\}$. $U_1 \subset H \times S$. This set represents host/switch connections .
- Set $U_2 = \{(s_i, s_j) \mid s_j \in S \wedge s_j \in S \wedge i \neq j\}$. $U_2 \subset S \times S$. This set represents switch/switch connections.
- Set $U = U_1 \bigcup U_2$.

- The network is represented by a graph G given by (N, U).
- $L_1 = \{ (h_i, s_j, s_{ja}) \vee (s_j, h_i, s_{ja}) \}$. This set represents the links between host and switch ports.
- $L_2 = \{ (s_i, s_{ia}, s_j, s_{jb}) \mid i \neq j \}$. This set represents the links between two switch ports.
- $L = L_1 \bigcup L_2$.
- $HC = \{ (h_i, h_j) \mid (h_i, h_j) \in H \times H \wedge i \neq j \}$. This set represents the combinations of OD hosts.
- A path P from $h_i \rightarrow h_j$ is a sequence $\langle h_i, s_1 \rangle$,$\langle s_1, s_2 \rangle$,$\langle s_2, s_3 \rangle$,...,$\langle s_{k-1}, s_k \rangle$,$\langle s_k, h_j \rangle$ such that each switch in the sequence $s_1, s_2, \cdots, s_k$ is distinct.
- $LP = \{(s_{ij}, (h_1, h_2), \cdots, (h_n, h_m)) \mid i = 1 \cdots n, j = 1 \cdots l\}$. This set represents all paths that include the switch port s_{ij}.

To generate the TM, some basic prerequisites are required:

1. The sets H, S, and the graph G are given.
2. The set HC is built using a function $f : H \rightarrow HC$. The cardinality of HC is given by the combination of all elements of H (n) taken from r, as shown in Equation 6.1:

$$CardHC = \binom{n}{r} = \frac{n!}{(n-r)! \cdot r!} \tag{6.1}$$

   Because each element in HC is a binary relation between members of H, the value for r is 2.

3. The set L is given.

The LP set provides, for each switch port, the list of OD hosts for which this port is part of the path. To build this set, the first step it to build the HC set, which contains all the possible combinations of pairs of OD hosts. For each pair in HC, the path in P is retrieved. Each path between the members of an OD pair consists of three types of links: (1) a link between the origin host and the first switch, expressed as $\langle h_i, s_1 \rangle$, (2) links between switches, expressed as $\langle s_i, s_j \rangle$, and (3) a link between the last switch and the destination host, expressed as $\langle s_k, h_j \rangle$. For each type of pair, the set L is searched to find the ports used in the link. For link types (1) and (3), a match is sought for elements in L_1. Elements in L_1 are described as (h_i, s_j, s_{ja}) or (s_j, h_i, s_{ja}). When a match is found, the switch port (s_{ja}) is returned. For link type (2), the same procedure is followed, with elements in L_2 searched for matching. In this case, when a match (s_i, s_j) is found, both ports(s_{ia}, s_{jb}) are returned. For each returned port, a pair $(\langle port \rangle, (O, D))$ is added or updated in *LP*.

6.2.1 Implementation

The data available from the *Queue Messages* activity (Figure 4.1) represent the traffic information on the network. These data, combined with the elements of the *LP* set, are used to generate the *key*s in the Map portion of the MapReduce design (Figure 6.1). This section presents the generation of the *LP* set and the procedure that combines the collected traffic data with the *LP* set elements to generate the final map. As shown in Figure 6.1, four map functions are executed in the map part.

The *Map Raw Data* function receives traffic statistics as messages; these messages have the layout provided in Figure 6.2. The timestamp field is in the Unix format, where the last three digits refer to the millisecond of the data collection time. the approach used here discards the last three digits because it provides the throughput in seconds. This operation changes the timpestamp 1522598210107 to 1522598210 and makes it possible to group statistics collected in the same second.

$$\underbrace{1522598210107}_{\text{timestamp}}\ \underbrace{\text{S2}}_{\text{switch id}}\ \underbrace{\text{S2:eth4}}_{\text{switch port}}\ \underbrace{3368}_{\text{bytes received}}\ \underbrace{3368}_{\text{bytes transmitted}}\ \underbrace{3}_{\text{second}}\ \underbrace{458000000}_{\text{nanosecond}}$$

Figure 6.2: Port statistics message layout.

The map generated by the *Map Raw Data* is sorted by the *Sort By Key* function. The sorted key is composed of the *timestamp* and *switch port* fields.

The *Generate Final Map* function provides two algorithms that are needed to generate the final map: The Generate *LP* set algorithm (Algorithm 6.1) generates the LP set, and the Generate $\langle Key, Value \rangle$ pairs algorithm (Algorithm 6.2), for every port statistics message, calculates the throughputs of the link that include this port and generates the $\langle Key, Value \rangle$ for all the OD pairs that include the port. The following sections describe these algorithms.

Generate *LP* Set Algorithm

Algorithm 6.1 shows the steps to generate the *LP* set. Topology information is collected from the OpenDaylight SDN controller to build the sets H, S, U, and L. The response comes in a JSON format, but is converted to an internal CSV format. Figure 6.3 shows the topology used in the experiments, and Figure 6.4 shows its representation in OpenDaylight.

Algorithm 6.1: Generate *LP* set

```
Input  : $H, S, L, P$
Output: $LP$
globals: hosts_list // Elements of H
         hosts_destination // Elements of H
         path // path between two hosts
         ports[2] = { null ,null } // link
hosts_list ← H
hosts_destination ← H

foreach $h_i$ in hosts_list do
    remove $h_i$ from hosts_destination
    foreach $h_j$ in hosts_destination do
        $HC \leftarrow \langle h_i, h_j \rangle$
    end
end

foreach $\langle h_i, h_j \rangle$ in HC do
    path ← getPath($h_i$, $h_j$)
    foreach element $\langle \beta_0, \beta_1 \rangle$ in path do
        ports = { null ,null }
        ports = getPorts($\langle \beta_0, \beta_1 \rangle$)
        if ports[0] not null then
            if ports[0] not in LP then
                LP.add(ports[0], $\langle h_i, h_j \rangle$)
            else
                LP.update(ports[0], $\langle h_i, h_j \rangle$)
            end
        end
        if ports[1] not null then
            if ports[1] not in LP then
                LP.add(ports[1], $\langle h_i, h_j \rangle$)
            else
                LP.update(ports[1], $\langle h_i, h_j \rangle$)
            end
        end
    end
end

Function getPath($h_i$, $h_j$) : P
    return $\langle h_i, s_1 \rangle$, $\langle s_1, s_2 \rangle$,$\langle s_2, s_3 \rangle$,...,$\langle s_{k-1}, s_k \rangle$,$\langle s_k, h_j \rangle$;
end

Function getPorts($\langle \beta_0, \beta_1 \rangle$) : linkports[2]
    linkports[2] = { null ,null }
    if $\langle \beta_0,\beta_1 \rangle \in L_1$ then
        linkports[0] = $L_1.s_{ia}$
    else
        if $\langle \beta_0,\beta_1 \rangle \in L_2$ then
            linkports[0] = $L_2.s_{ia}$
            linkports[1] = $L_2.s_{jb}$
        end
    end
    return linkports
end
```

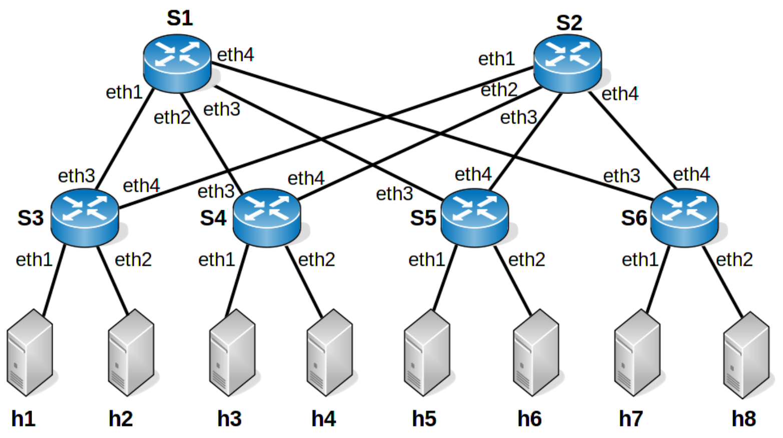


Figure 6.3: Network topology.

Figure 6.4 identifies the elements of the set H, which contains all hosts in the network, as fields starting with the string ”*host*” and identifies the elements of the set S, which contains all switches in the network, as fields starting with the string ”*Sn*”, where n identifies the switch number. Figure 6.4 also identifies the elements of U and L.

```
"1516218618282;S1:eth2;S1:eth2;S1;S4:eth3;S4",
"1516218618282;S2:eth1;S2:eth1;S2;S3:eth4;S3",
"1516218618282;S1:eth1;S1:eth1;S1;S3:eth3;S3",
"1516218618282;S1:eth4;S1:eth4;S1;S6:eth3;S6",
"1516218618282;S2:eth3;S2:eth3;S2;S5:eth4;S5",
"1516218618282;S1:eth3;S1:eth3;S1;S5:eth3;S5",
"1516218618282;S2:eth2;S2:eth2;S2;S4:eth4;S4",
"1516218618282;S3:eth4;S3:eth4;S3;S2:eth1;S2",
"1516218618282;S4:eth3;S4:eth3;S4;S1:eth2;S1",
"1516218618282;S6:eth1/host:07;S6:eth1;S6;host:07;host:07",
"1516218618282;S3:eth3;S3:eth3;S3;S1:eth1;S1",
"1516218618282;S2:eth4;S2:eth4;S2;S6:eth4;S6",
"1516218618282;S5:eth4;S5:eth4;S5;S2:eth3;S2",
"1516218618282;S6:eth3;S6:eth3;S6;S1:eth4;S1",
"1516218618282;S5:eth3;S5:eth3;S5;S1:eth3;S1",
"1516218618282;S4:eth4;S4:eth4;S4;S2:eth2;S2",
"1516218618282;S5:eth1/host:05;S5:eth1;S5;host:05;host:05",
"1516218618282;S6:eth4;S6:eth4;S6;S2:eth4;S2",
"1516218618282;S5:eth2/host:06;S5:eth2;S5;host:06;host:06",
"1516218618282;S4:eth2/host:04;S4:eth2;S4;host:04;host:04",
"1516218618282;S3:eth2/host:02;S3:eth2;S3;host:02;host:02",
"1516218618282;host:08/S6:eth2;host:08;host:08;S6:eth2;S6",
"1516218618282;S6:eth2/host:08;S6:eth2;S6;host:08;host:08",
"1516218618282;host:01/S3:eth1;host:01;host:01;S3:eth1;S3",
"1516218618282;host:03/S4:eth1;host:03;host:03;S4:eth1;S4",
"1516218618282;host:02/S3:eth2;host:02;host:02;S3:eth2;S3",
"1516218618282;S4:eth1/host:03;S4:eth1;S4;host:03;host:03",
"1516218618282;host:05/S5:eth1;host:05;host:05;S5:eth1;S5",
"1516218618282;host:04/S4:eth2;host:04;host:04;S4:eth2;S4",
"1516218618282;host:07/S6:eth1;host:07;host:07;S6:eth1;S6",
"1516218618282;S3:eth1/host:01;S3:eth1;S3;host:01;host:01",
"1516218618282;host:06/S5:eth2;host:06;host:06;S5:eth2;S5";
```

Figure 6.4: OpenDaylight topology representation.

Figure 6.5 shows the type of lines, from Figure 6.4, used to extract the elements of U_1. The *host* and *switch* fields form the elements of U_1.

$$\underbrace{\text{1516218618282}}_{\text{timestamp}}\ \underbrace{\text{host:01/S3:eth1}}_{\text{link id}}\ \underbrace{\text{host:01}}_{\text{host port}}\ \underbrace{\text{host:01}}_{\text{host}}\ \underbrace{\text{S3:eth1}}_{\text{switch port}}\ \underbrace{\text{S3}}_{\text{switch}}$$

Figure 6.5: Line for extracting U_1 elements.

Figure 6.6 shows the type of lines, from Figure 6.4, used to extract the elements of U_2. The *switch origin* and *switch destination* fields form the elements of U_2.

$$\underbrace{\text{1516218618282}}_{\text{timestamp}}\ \underbrace{\text{S1:eth2}}_{\text{link id}}\ \underbrace{\text{S1:eth2}}_{\text{switch origin port}}\ \underbrace{\text{S1}}_{\text{switch origin}}\ \underbrace{\text{S4:eth3}}_{\text{switch destination port}}\ \underbrace{\text{S4}}_{\text{switch destination}}$$

Figure 6.6: Line for extracting U_2 elements.

The elements of L are directly related to the elements of U because they provide a critical piece of information, the port number in each switch used in a connection. The port number is identified as a string of the form "*Sn:ethm*", where S stands for switch, n identifies its number, *eth* indicates a switch port, and m identifies the port number. The port number is present in Figure 6.5 (*switch port*) and Figure 6.6 (*switch origin port* and *switch destination port*). Therefore, for each element of U_1 (h_i, s_j), the *switch port* (s_{ja}) is added to form the elements of L_1. Figure 6.7 shows an example of an L_1 element. The host port is not used in L_1 because it is assumed that only one network interface is available in the host.

```
[host:07, S6, S6:eth1]
```

Figure 6.7: L_1 element.

The elements of L_2 are generated in the same way; for each element of U_2 (s_i, s_j), the ports in the connection (s_{ia},s_{jb}) are then identified. Figure 6.8 shows an example of al L_2 element.

```
[S6, S6:eth3, S1, S1:eth4]
```

Figure 6.8: L_2 element.

The graph G is built using GraphStream [92], which is a dynamic graph library in Java. The approach used here extracts the nodes (hosts (H), switches(S), and links(U)) and feeds the graph with these values.

Lines 8–13 from Algorithm 6.1 generate the HC set. For each OD pair (h_i, h_j) in HC, the path connecting the two hosts is returned by the function *getPath(h_i, h_j)* (Lines 15–16). In the topology presented in Figure 6.3, if an attempt is made to find the paths between any two hosts, loops can be found in the network graph. To overcome this issue, the topology was divided into two parts: the first with S1 as the root, and the second with S2 as the root, as shown in Figures 6.9 and 6.10, respectively. With this approach, the solution presented here provides the TM between all possible combinations of OD traversing S1 or S2.

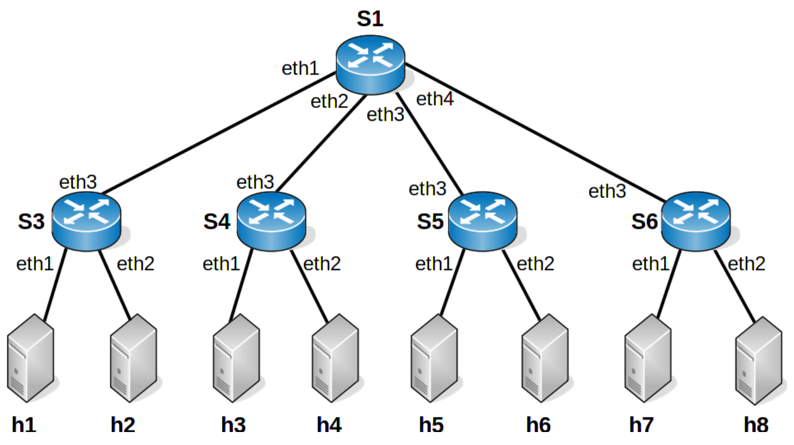


Figure 6.9: Topology traversing S1.

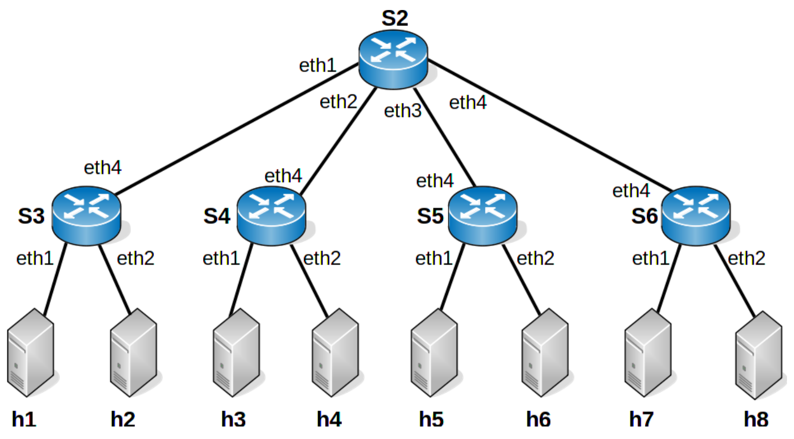


Figure 6.10: Topology traversing S2.

As an example, Figure 6.11 shows the return of the *getPath(h_i, h_j)* function if (h_7, h_5) and (h_8, h_1) are passed as parameters in two separate calls. For (h_7, h_5), the graph with S1 (Figure 6.9) as the root is used. For (h_8, h_1), the graph with S2 (Figure 6.10) as the root is used.

```
[host:07, S6, S1, S5, host:05]
[host:08, S6, S2, S3, host:01]
```

Figure 6.11: Paths between (h_7, h_5) and (h_8, h_1).

The *getPorts(⟨β_0, β_1⟩): linkports[2]* function (lines 41–52) returns the $\langle\beta_0, \beta_1\rangle$ link ports. If the $\langle\beta_0, \beta_1\rangle$ link is an element of the L_1, the function returns only the switch port (lines 43–45). If the $\langle\beta_0, \beta_1\rangle$ link is an element of the L_2, the function returns both switch ports (lines 46–49).

This function searches the set *L* to provide these data. Using them, the elements of *LP* can be generated (lines 17–34). For each value returned from the *getPorts()* function, an entry in *LP* is added. At the end of this procedure, each switch port *LP* will have the OD pairs that include the switch port. Figure 6.12 provides an example giving the OD pairs that include the S3:eth3 switch port. The first line identifies the switch port. All the others have the following format: "root/origin/destination". On the second line, S1 is the root, "host:01" is the origin host, and "host:03" is the destination host.

```
S3:eth3
    S1/host:01/host:03
    S1/host:01/host:04
    S1/host:01/host:05
    S1/host:01/host:06
    S1/host:01/host:07
    S1/host:01/host:08
    S1/host:02/host:03
    S1/host:02/host:04
    S1/host:02/host:05
    S1/host:02/host:06
    S1/host:02/host:07
    S1/host:02/host:08
```

Figure 6.12: Port origin/destination pairs.

As previously mentioned, the approach used here divides the network topology into two parts to avoid loops in the network graph. Loops in the network are not the only problem faced when traversing a network graph. Another problem is to find the shortest path between an OD pair of hosts. Even though the Dijkstra algorithm solves this problem, it is still possible to find more than one shortest path between an OD pair of hosts. In this case, one of the shortest paths can be selected for monitoring. The MapReduce approach proposed here enables the monitoring of more than one shortest path. The solution is to add a different identifier to the generated OD definition in *LP*. For instance, in a network with two shortest paths from host **Ha** to host **Hb**, where the first shortest path includes switch port **SAp1** and the second shortest path includes switch port **SBp2**, the entries in *LP* for **SAp1** and **SBp2** can differentiate the two shortest paths by creating OD definitions like those presented in Figure 6.13. The presence of the strings **PATH1** and **PATH2** differentiates the OD definitions for the two shortest paths.

6.2.2 Generate $\langle key, value \rangle$ Pairs Algorithm

The goal of this algorithm is to generate the final map. This map contains the traffic data for every link in the network. The link is not part of the *key*; only the OD pairs that are connected

```
SAp1
          PATH1/Ha/Hb
SBp2
          PATH2/Ha/Hb
```

Figure 6.13: *LP* entries for two paths between the same OD.

through this link compose the key. The map consists of OD pairs as the *key* and the traffic data of the link as the *value*. The pseudo-code provided in Algorithm 6.2 describes the Generate $\langle key, value \rangle$ pairs algorithm.

For each port statistics message (Figure 6.2), the message of the port pair with the same timestamp is retrieved (Line 8). A port and its pair are an element of the set L. If the port is an element of L_1, the port and its pair have the same value because in L_1 the link is formed between a host port and a switch port, but the host port is not monitored. For L_2 elements, (s_i, s_{ia}, s_j, s_{jb}), if the fields **second** and **nanosecond** are exactly the same for (s_{ia}, s_{jb}), the value of **BR** for s_{ia} should be the same as the value of **BT** for s_{jb}, and the other way around. However, collecting statistics for (s_{ia}, s_{jb}) at the same **second** and **nanosecond** is an unrealistic scenario. Hence, one of the messages has the most up-to-date statistics for the link. To use the most up-to-date statistics, the message with the larger value of (**BR** + **BT**) represents the throughput of the link (Line 10). Figure 6.14 provides an example. The ”S1:eth1” and ”S3:eth3” ports form a link, and the statistics of the ”S3:eth3” port are used as the throughput of the link in that timestamp.

For each message (Figure 6.2), two calculations are performed:

1. **Current throughput in every link**. The current throughput is calculated according to Equation 6.2 [18]:
$$T_i(t) = \frac{c_i(t) - c_i(t-T)}{T} \tag{6.2}$$
where $T_i(t)$ is the current throughput of link i at instant t, $c_i(t)$ is the statistics value (*Bytes Received (BR) + Bytes Transmitted (BT)*) at instant t, and T is the polling interval (Line 11). Lines 9 and 13 respectively retrieve and update $c_i(t-T)$ for each link.

2. **Accumulated throughput in every link**. The accumulated throughput is calculated according to Equation 6.3 (Line 12):
$$T_i(t) = \frac{(BR + BT)}{S} \tag{6.3}$$
where $T_i(t)$ is the accumulated throughput of link i at instant t and S is the number of seconds for which the link is active.

The final step in Algorithm 6.2 is map generation. The *LP* set is queried to provide the

```
1522598234107; S1; S1:eth1; 776966; 69572908; 27; 802000000
1522598234107; S3; S3:eth3; 75257774; 837686; 28; 773000000
```

Figure 6.14: Collected statistics on link S1:eth1/S3:eth3.

Algorithm 6.2: Generate $\langle key, value \rangle$ pairs

```
Input  : Ports Statistics Messages, LP,L
Output: ⟨key, value⟩
1  globals: path_list // List of all paths
2  locals:  previousStat // Previous Statistics
3           throughput // Accumulated throughput
4           currThroughput // Current throughput
5           pair // pair of the current port
6           interval = 3 // interval between readings
7  foreach message do
8      pair = getPairStatistic(L.getPair(message.switchport),message.timestamp)
9      previousStat = getLinkStat(message.switchport, pair)
10     if ((message.BT + message.BR) > (pair.BT + pair.BR))) then
11         currThroughput = ((message.BT + message.BR) - previousStat) / interval
12         throughput = (message.BT + message.BR) / message.second
13         updateLinkStat(message.switchport,pair,(message.BT + message.BR))
14         path_list = LP.getListPaths(message.switchport)
15         foreach element pl in path_list do
16             key = concatenate(pl, message.timestamp)
17             value = (throughput,currThroughput)
18             generate ⟨key, value⟩
19         end
20     end
21 end
```

paths that include the selected port (Line 14). For every returned path, a $\langle key, value \rangle$ pair is generated (Lines 15–19). Table 6.1 provides an example of the generated $\langle key, value \rangle$ pairs for the S3-eth3 port in Figure 6.12. The *key* is composed of the path and timestamp, and the *value* contains the calculated current (C_v, Equation 6.2) and accumulated (A_v, Equation 6.3) throughputs.

Key	**Value**	
	Current	**Accumulated**
S1/host:01/host:03 1522598234	C_v	A_v
S1/host:01/host:04 1522598234	C_v	A_v
S2/host:02/host:05 1522598234	C_v	A_v
S2/host:02/host:06 1522598234	C_v	A_v
S1/host:01/host:04 1522598234	C_v	A_v
S2/host:02/host:05 1522598234	C_v	A_v
S1/host:01/host:03 1522598234	C_v	A_v
S2/host:02/host:06 1522598234	C_v	A_v
S2/host:02/host:05 1522598234	C_v	A_v
$\cdots\ \cdots$ 1522598234	C_v	A_v

Table 6.1: Generated $\langle key, value \rangle$ pairs.

For all port statistics in the same timestamp, the same type of $\langle key, value\rangle$ pairs as those generated for the S3-eth3 port and shown in Table 6.1 are generated.

The *Reduce By Key* function aggregates *values* with the same *key*. Table 6.2 shows the reduce operation applied to some of the $\langle key, value\rangle$ pairs presented in Table 6.1. The TM consists of records composed of the fields provided by Table 6.2 (Key, Current, Accumulated).

Key	**Value**	
	Current	**Accumulated**
S1/host:01/host:03 1522598234	$\sum C_v$	$\sum A_v$
S1/host:01/host:04 1522598234	$\sum C_v$	$\sum A_v$
S2/host:02/host:05 1522598234	$\sum C_v$	$\sum A_v$
S2/host:02/host:06 1522598234	$\sum C_v$	$\sum A_v$
$\cdots\cdots$ 1522598234	$\sum C_v$	$\sum A_v$

Table 6.2: Reduce by key applied to $\langle key, value\rangle$ pairs.

6.3 Summary

This chapter has presented the proposed MapReduce approach to estimate the TM and started by delineating the process for obtaining the TM. The process consists of four *Map* functions and one *Reduce* function. Furthermore, the chapter described the basic structures used as an input to the algorithms that estimate the TM. Finally, this chapter presented the layout of the network topology provided by the SDN controller and the two algorithms that implement the described process.

Chapters 4, 5, and this chapter focus on the description of the Big Data traffic monitoring method and its implementation. This thesis also proposes a traffic analysis method, and the next chapter presents this method.

Chapter 7

Big Data Traffic Analysis

Traffic analysis processes a large amount of data to deliver a wide range of insights about network usage. Processing a large amount of data is inherently a batch process. The lambda architecture presented in Subsection 2.2.3 of this thesis defines a dedicated branch for batch processing. Figure 2.10 shows the batch layer, which defines a *Master Dataset* that holds stream data to be processed by batch jobs. The batch jobs run in the *Batch Processing Platform* to feed the *Serving layer* with *batch views*. The *batch views* are the source of data for traffic analysis and monitoring applications.

In this chapter, a batch-processing traffic analysis method is proposed to run on the *Batch Processing Platform* and generate batch views that reflect different analyses of network traffic and provide historical data. The traffic analysis involves the following levels of aggregations: *switch port*, *switch*, *link*, *path*, and *network*. The MapReduce approach is recommended to generate these different aggregation levels because of its ability to process a large amount of data using parallel and distributed execution.

7.1 Traffic Analysis Method

Figure 7.1 depicts the advanced traffic analysis method as an activity diagram. The flow of control presented in the activity diagram is established based on data dependency among the activities to provide different levels of aggregation.

The *Store Streaming Data* activity receives raw data coming from the SDN controller, which provide the starting point for statistics calculations. The *Calculate Switch Port Traffic Statistics* activity calculates the basic level of statistics, which is the port traffic statistics. Port traffic statistics enable several levels of aggregation to be generated, such as switch traffic, network traffic, link traffic, and path traffic. The *Generate Switch Traffic Statistics* activity aggregates switch statistics by timestamp. The switch statistics involve aggregating the traffic in the switch ports. The *Generate Network Port Statistics* activity aggregates network statistics based on port statistics; these data are necessary to generate port traffic analysis. Even though the *Generate Switch Traffic Statistics* and the *Generate Network Port Statistics* activities use the same data source, they proceed in parallel because they output different levels of aggregation. The *Generate Network Switch Statistics* activity aggregates network statistics based on switch statistics. The values generated up to this point provide statistics calculated for switches, ports, and the network and enable traffic analysis to be generated. The *Generate Switch Traffic Analysis* activity provides a traffic analysis of individual switches. The *Generate Switch Port Traffic Analysis* activity provides a traffic analysis of each port in every switch. Because a pair of switch ports forms a link, the activity *Generate Link Traffic Analysis* depends on the analysis provided by the *Generate Switch Port Traffic Analysis* activity. This same level of dependency also applies to the *Generate Path Traffic Analysis* activity, because the paths between hosts are composed of links, and consequently to generate path traffic analysis, link traffic analysis must be provided in advance.

The following sections detail the activities and objects presented in the activity diagram.

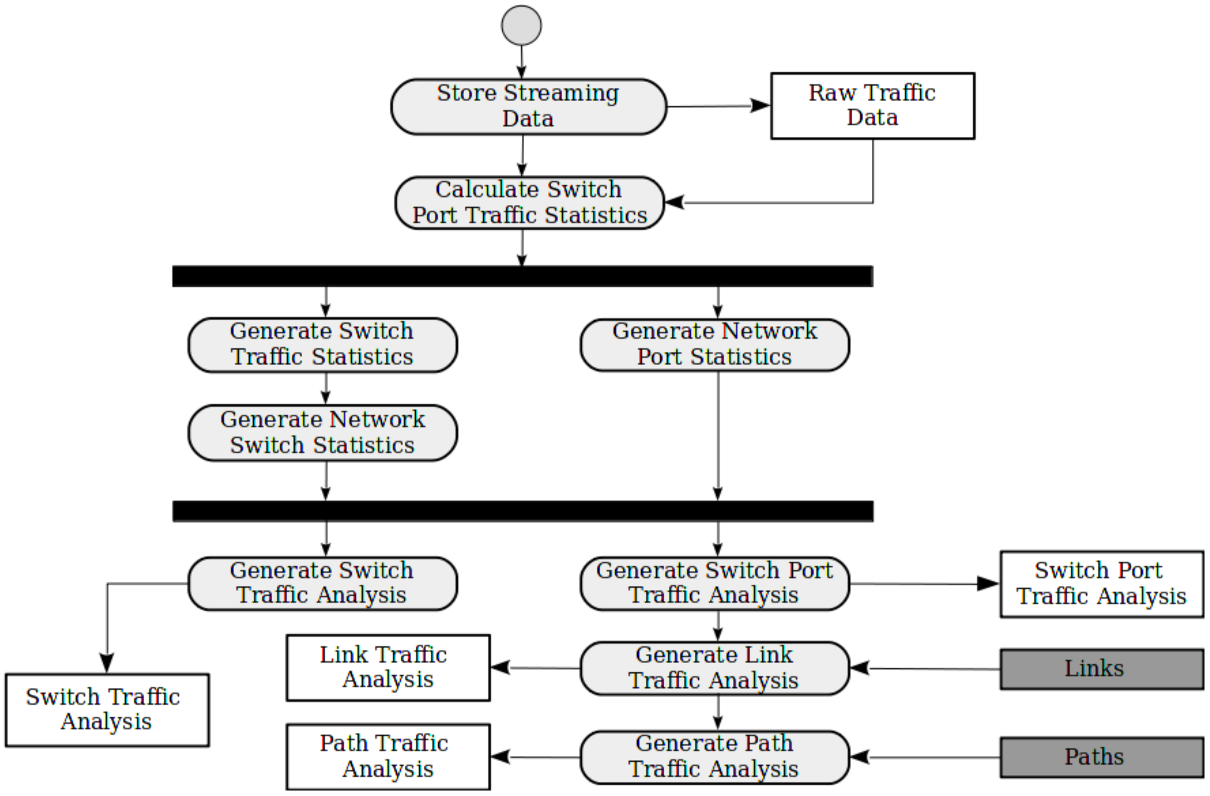

Store Streaming Data
Raw Traffic Data
Calculate Switch Port Traffic Statistics
Generate Switch Traffic Statistics
Generate Network Port Statistics
Generate Network Switch Statistics
Generate Switch Traffic Analysis
Generate Switch Port Traffic Analysis
Switch Port Traffic Analysis
Switch Traffic Analysis
Link Traffic Analysis
Generate Link Traffic Analysis
Links
Path Traffic Analysis
Generate Path Traffic Analysis
Paths

Figure 7.1: Traffic analysis method.

7.1.1 Store Streaming Data

This activity collects raw traffic data coming from the SDN controller and stores them in the *raw traffic data* object flow. The *Master Dataset* is the physical repository of the *raw traffic data* object flow. The batch job processes the full content of the *Master Dataset*. Therefore, this activity continuously appends new data, and never removes data. The data coming from the SDN controller are not in the format in which they will be processed, and this activity formats these incoming data for further processing.

7.1.2 Calculate Switch Port Traffic Statistics

Figure 7.2 shows the layout of the collected raw data stored by the *Store Streaming Data* Activity. This layout differs from the one presented in Figure 6.2 because it adds the *packets received* (PR) and *packets transmitted* (PT) attributes. The values provided represent counter values for each switch port.

$$\underbrace{514516101215}_{\text{timestamp}}\ \underbrace{\text{S2}}_{\text{switch id}}\ \underbrace{\text{S2:eth4}}_{\text{switch port}}\ \underbrace{100}_{\text{packets received}}\ \underbrace{272}_{\text{packets transmitted}}\ \underbrace{11623}_{\text{bytes received}}\ \underbrace{27251}_{\text{bytes transmitted}}\ \underbrace{3}_{\text{second}}\ \underbrace{458000000}_{\text{nanosecond}}$$

Figure 7.2: Traffic statistics message layout.

The timestamp attribute follows the same rule as presented in Section 6.2.1. Table 7.1 describes the throughput statistics that this activity generates for each port.

All other succeeding activities use the output generated in this activity to perform their particular levels of aggregation. Figure 7.3 provides an example of the output row after the calculations presented in Table 7.1 have been applied to the values in Figure 7.2.

$$\underbrace{514516101}_{\text{timestamp}}\ \underbrace{\text{S2}}_{\text{switch id}}\ \underbrace{\text{S2:eth4}}_{\text{switch port}}\ \underbrace{100}_{\text{PR}}\ \underbrace{272}_{\text{PT}}\ \underbrace{11623}_{\text{BR}}\ \underbrace{27251}_{\text{BT}}\ \underbrace{33.3}_{\text{PRT}}\ \underbrace{90.66}_{\text{PTT}}\ \underbrace{124}_{\text{PTr}}\ \underbrace{3874.33}_{\text{BRT}}\ \underbrace{9083.66}_{\text{BTT}}\ \underbrace{12958}_{\text{BTr}}$$

Figure 7.3: Output generated by the calculate switch port traffic statistics activity.

7.1.3 Generate Switch Traffic Statistics

This activity performs MapReduce operations to generate switch statistics. The generated statistics are necessary to calculate the switch traffic and perform the switch port traffic analysis. Figure 7.3 shows the layout of the data processed by this activity. The composite key that aggregates switch statistics at every timestamp contains the following attributes:

Calculated Throughput	**Equation**
Packet Received Throughput (PRT)	$$\frac{PR}{S} \tag{7.1}$$
Packet Transmitted Throughput (PTT)	$$\frac{PT}{S} \tag{7.2}$$
Packet Throughput (PTr)	$$\frac{(PT + PR)}{S} \tag{7.3}$$
Bytes Received Throughput (BRT)	$$\frac{BR}{S} \tag{7.4}$$
Bytes Transmitted Throughput (BTT)	$$\frac{BT}{S} \tag{7.5}$$
Bytes Throughput (BTr)	Equation 6.3

Table 7.1: Calculated port throughputs.

- $\langle switchid, timestamp \rangle$

The following statistics are generated using this key:

- **Summation**. Each item in Table 7.1 has its values totalized. For instance, the equation $TPRT = \sum_{p=1}^{n} PRT$ summarizes the Packet Received Throughput (PRT), the equation $TPTT = \sum_{p=1}^{n} PTT$ summarizes the Packet Transmitted Throughput (PTT), and so on. Packets and bytes received, and packets and bytes transmitted are also summarized.

- **Mean**. The mean of every item mentioned in the previous *summation* paragraph is also calculated using the mean Equation 7.6:

$$M = \frac{1}{n} * \sum_{i=1}^{n} p_i \tag{7.6}$$

  where M is the mean, n is the number of ports in the switch, and p represents the base value for the mean (PR, PT, BR, BT, PRT, PTT, PTr, BRT, BTT, Btr) in switch port i.

- **Standard Deviation**. The standard deviation of every item mentioned in the previous *summation* paragraph is also calculated using the standard deviation equation 7.7:

$$\sigma = \sqrt{\frac{\sum (x - M)^2}{n}} \tag{7.7}$$

where σ is the standard deviation, x is the individual value of each item mentioned in the previous *summation* paragraph (PR, PT, BR, BT, PRT, PTT, PTr, BRT, BTT, and Btr), M is the mean calculated in the previous **Mean** part for PR, PT, BR, BT, PRT, PTT, PTr, BRT, BTT, and Btr, and n is the number of items.

7.1.4 Generate Network Switch Statistics

The statistics generated in this activity are an aggregation of the values produced by the previous activity (Section 7.1.3). Using switch-aggregated statistics enables the proposed method to provide switch traffic analysis based on network traffic statistics.

This activity generates the same type of statistics as presented in Section 7.1.3 (summation, mean, and standard deviation) with the difference that the generated values refer to the entire network. The key used to aggregate these values contains the following attribute:

- $\langle timestamp \rangle$

7.1.5 Generate Network Port Statistics

This activity also generates the same type of statistics as those generated in Section 7.1.4 and even uses the same key, the *timestamp*. The network statistics aggregation in this activity is based on port statistics and is needed to generate switch port traffic analysis.

Even though this activity and the activity presented in Section 7.1.4 perform aggregation using the same key, the main difference between them is the mean calculation. Table 7.2 provides an example of this difference. Table 7.2a shows the mean calculation based on port statistics, and Table 7.2b shows the mean calculation based on switch statistics. Even though the totals are the same, the means are different. This is why the proposed method defines two distinct activities to generate network statistics.

7.1.6 Generate Switch Traffic Analysis

This activity provides a z-score switch traffic analysis. A z-score tells how many standard deviations a value is away from the mean, and in which direction [93]. A positive z-score

Timestamp	Switch	Port	Throughput
1522598234	S2	S2:eth2	12590
1522598234	S1	S1:eth1	3450
1522598234	S3	S3:eth4	2550
1522598234	S2	S2:eth1	5620
1522598234	S3	S3:eth2	9740
1522598234	S4	S4:eth1	15420
1522598234	S5	S5:eth2	11560
1522598234	S4	S4:eth3	12590
1522598234	S1	S1:eth2	8456
1522598234	S2	S2:eth3	7458
1522598234	S5	S5:eth1	19854
		Total	109288
		Mean	9935.27

(a) Mean for port statistics

Timestamp	Switch	Throughput
1522598234	S2	25668
1522598234	S1	11906
1522598234	S3	12290
1522598234	S4	28010
1522598234	S5	31414
	Total	109288
	Mean	21857.6

(b) Mean for switch statistics

Table 7.2: Mean statistics

indicates that a value is above the mean, whereas a negative z-score indicates that a value is below the mean. The z-score is calculated using Equation 7.8:

$$z = \frac{x - \mu}{\sigma} \tag{7.8}$$

where z is the z-score, x is the value of one specific value, μ is the mean of all values, and σ is the standard deviation of all values.

A value with a z-score less than -3 or greater than +3 is called an *outlier* [93] and may be an indication of abnormal behaviour. The traffic behaviour in any switch can be analyzed by calculating the z-score of the following parameters:

- Packets Received (PR), Packets Transmitted (PT), Bytes Received (BR), Bytes Transmitted (BT), Packets Received Throughput (PRT), Packets Transmitted Throughput (PTT), Packets Throughput (PTr), Bytes Received Throughput (BRT), Bytes Transmitted Throughput (BTT), Bytes Throughput (BTr).

A *Join* operation with the results of the activities described in Sections 7.1.3 and 7.1.4 generates the **Switch Traffic Analysis** object, as shown in Figure 7.4.

The **Switch Traffic Analysis** object stores the values generated in this section, Section 7.1.3, and Section 7.1.4. For any application that needs to retrieve the statistical and historical behaviour of every switch in the network, the **Switch Traffic Analysis** object is the source of these data. Each record in this object, in addition to the switch statistics, also contains the network statistics. Consequently, every record replicates network statistics. Even though this

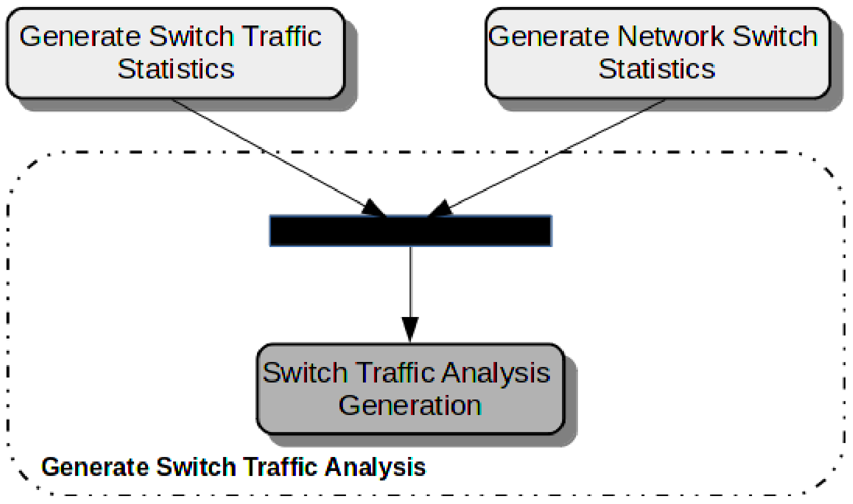

Generate Switch Traffic Statistics
Generate Network Switch Statistics
Switch Traffic Analysis Generation
Generate Switch Traffic Analysis

Figure 7.4: Switch analysis generation.

approach demands more storage space, any additional analysis can benefit from it because with one read operation, an application can retrieve switch and network statistics, immediately speeding up the analysis process. Table 7.3 shows an example of the partial contents of the **Switch Traffic Analysis** object.

1522598234	S1 statistics columns	S1 traffic analysis columns	network statistics columns
1522598234	S2 statistics columns	S2 traffic analysis columns	network statistics columns
1522598234	S3 statistics columns	S3 traffic analysis columns	network statistics columns
1522598234	S4 statistics columns	S4 traffic analysis columns	network statistics columns
1522598235	S1 statistics columns	S1 traffic analysis columns	network statistics columns
1522598235	S2 statistics columns	S2 traffic analysis columns	network statistics columns
1522598235	S3 statistics columns	S3 traffic analysis columns	network statistics columns
1522598235	S4 statistics columns	S4 traffic analysis columns	network statistics columns
... ...			

Table 7.3: Example of the switch traffic analysis object.

7.1.7 Generate Switch Port Traffic Analysis

This activity generates the z-scores of switch ports. The z-scores of each port within the switch context and the network context are calculated. Executing a *Join* operation using the output of sections 7.1.2, 7.1.3, and 7.1.5 enables the z-score calculation for all parameters listed in Section 7.1.6. Figure 7.5 shows the *Join* operation.

The **Switch Port Traffic Analysis** object stores the result of the *Join* operation. Table 7.4 shows a partial example of the content of the **Switch Port Traffic Analysis** object.

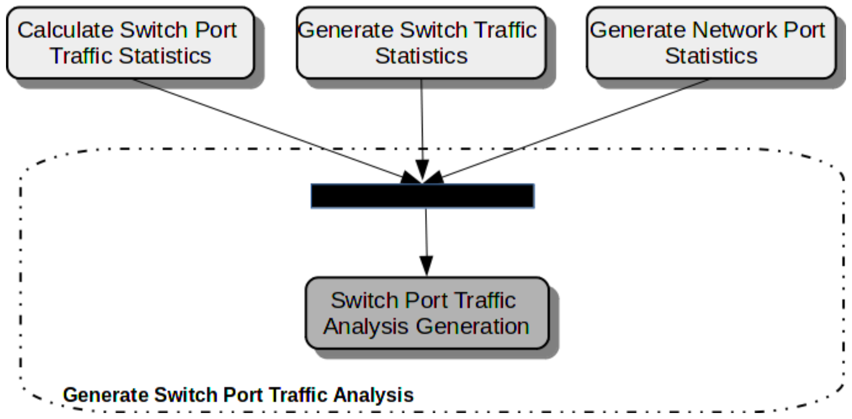

Calculate Switch Port Traffic Statistics
Generate Switch Traffic Statistics
Generate Network Port Statistics
Switch Port Traffic Analysis Generation
Generate Switch Port Traffic Analysis

Figure 7.5: Port analysis generation.

1522598234	S1:eth1 statistics columns	S1:eth1 traffic analysis columns	S1 traffic statistics columns	network statistics columns
1522598234	S2:eth3 statistics columns	S2:eth3 traffic analysis columns	S2 traffic statistics columns	network statistics columns
1522598234	S1:eth3 statistics columns	S1:eth3 traffic analysis columns	S1 traffic statistics columns	network statistics columns
1522598234	S3:eth2 statistics columns	S3:eth2 traffic analysis columns	S3 traffic statistics columns	network statistics columns
1522598234	S4:eth4 statistics columns	S4:eth4 traffic analysis columns	S4 traffic statistics columns	network statistics columns
1522598235	S1:eth1 statistics columns	S1:eth1 traffic analysis columns	S1 traffic statistics columns	network statistics columns
1522598235	S2:eth3 statistics columns	S2:eth3 traffic analysis columns	S2 traffic statistics columns	network statistics columns
1522598235	S1:eth3 statistics columns	S1:eth3 traffic analysis columns	S1 traffic statistics columns	network statistics columns
1522598235	S3:eth2 statistics columns	S3:eth2 traffic analysis columns	S3 traffic statistics columns	network statistics columns
1522598235	S4:eth4 statistics columns	S4:eth4 traffic analysis columns	S4 traffic statistics columns	network statistics columns
... ...				

Table 7.4: Example of the switch port traffic analysis object.

7.1.8 Generate Link Traffic Analysis

Every link in the network is composed of two ports belonging to different switches unless the switch port is connected to a host. In this case, the link is the switch port. The **Links** object (Figure 7.1) is a container for the links formed in the network. The procedure to select a port from one of the two ports in a link is similar to the procedure presented in Subsection 6.2.2, i.e., selecting the link with the maximum value of $BR + BT$. A *Join* operation is executed between the output of the **Generate Switch Port Traffic Analysis** activity and the **Links** object. This operation selects the maximum value for $(BR + BT)$ of each link at every timestamp and stores the result in the **Link Traffic Analysis** object, as shown in Figure 7.6. The layout of the object **Link Traffic Analysis** is the same as that presented in Table 7.4 with the addition of the link ID.

7.1.9 Generate Path Traffic Analysis

The goal of this activity is to generate the traffic analysis of all OD pairs of hosts. This activity uses the **Paths** object, which contains the set of all links between the OD pairs of hosts. The procedure to build this set is presented in Algorithm 6.1. A *Join* operation is executed between

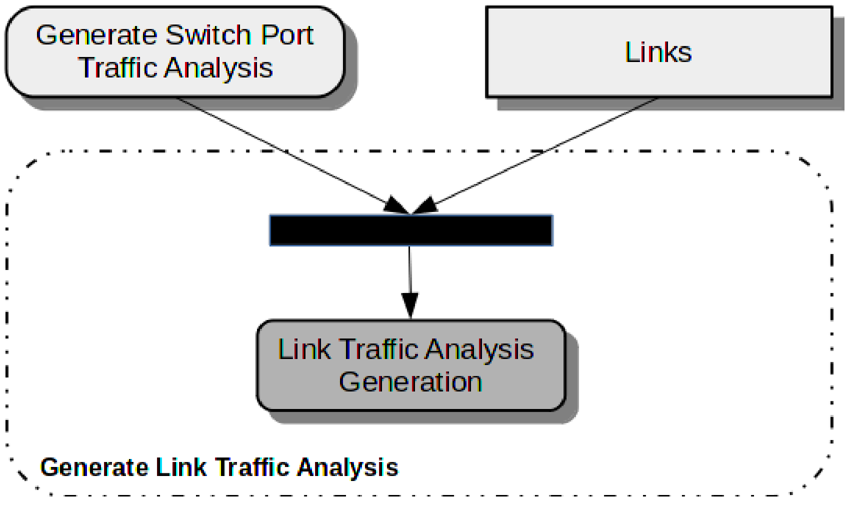

Generate Switch Port Traffic Analysis
Links
Link Traffic Analysis Generation
Generate Link Traffic Analysis

Figure 7.6: Link analysis generation.

the output of the *Generate Link Traffic Analysis* activity and the **Paths** object to populate the **Path Traffic Analysis** object, as shown in Figure 7.7.

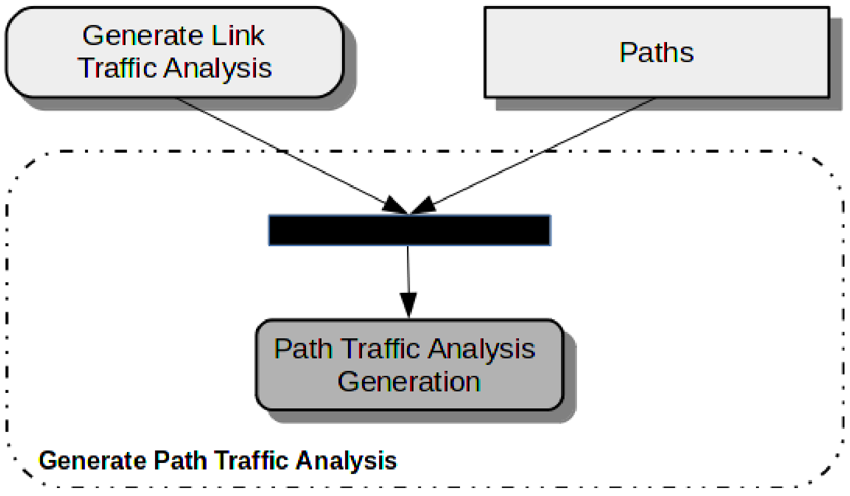

Generate Link Traffic Analysis
Paths
Path Traffic Analysis Generation
Generate Path Traffic Analysis

Figure 7.7: Path analysis generation.

Table 7.5 shows an example of the partial content of the **Path Traffic Analysis** object. As the **Paths** object is built by Algorithm 6.1, the idea of the *root* path is used in this activity, and the *root* path attribute is part of the path identification.

7.2 Summary

This chapter has described a Big Data traffic analysis method based on MapReduce operations to generate several levels of traffic statistics and traffic analysis aggregation. Besides the ca-

1522598234	S1:h1-h7	S1:h1-h7 statistics columns	S1:h1-h7 traffic analysis columns	path statistics columns
1522598234	S1:h2-h5	S1:h2-h5 statistics columns	S1:h2-h5 traffic analysis columns	path statistics columns
1522598234	S2:h1-h7	S2:h1-h7 statistics columns	S2:h1-h7 traffic analysis columns	path statistics columns
1522598234	S2:h4-h6	S2:h4-h6 statistics columns	S2:h4-h6 traffic analysis columns	path statistics columns
1522598234	S2:h3-h8	S2:h3-h8 statistics columns	S2:h3-h8 traffic analysis columns	path statistics columns
1522598235	S1:h1-h7	S1:h1-h7 statistics columns	S1:h1-h7 traffic analysis columns	path statistics columns
1522598235	S1:h2-h5	S1:h2-h5 statistics columns	S1:h2-h5 traffic analysis columns	path statistics columns
1522598235	S2:h1-h7	S2:h1-h7 statistics columns	S2:h1-h7 traffic analysis columns	path statistics columns
1522598235	S2:h4-h6	S2:h4-h6 statistics columns	S2:h4-h6 traffic analysis columns	path statistics columns
1522598235	S2:h3-h8	S2:h3-h8 statistics columns	S2:h3-h8 traffic analysis columns	path statistics columns
… … … … … … …				

Table 7.5: Example of the path traffic analysis object.

pability for parallel and distributed execution, the MapReduce approach facilitates the process of value aggregation by a diverse number of keys, enabling the generation of statistics and analysis at several levels of aggregation.

The proposed method is made up of a series of activities and objects, and an activity diagram describing activity flow control.

The **Raw Traffic Data**, **Switch Traffic Analysis**, **Switch Port Traffic Analysis**, **Link Traffic Analysis**, and **Path Traffic Analysis** objects are persisted in secondary memory. These objects not only provide traffic analysis and statistics, but they also provide a rich source of historical information that can be used to verify traffic trends in links, paths, switches, and ports and help in anomaly detection and network provisioning.

So far, this thesis has described the methods and implementations proposed for Big Data traffic monitoring and traffic analysis. The next chapter provides the experimental results obtained by implementing the methods proposed in Chapters 4, 6, and this chapter.

Chapter 8

Experimental Results

This chapter presents an evaluation of the methods proposed in this thesis. Section 8.1 describes the results of the Big Data traffic monitoring method experiments. Section 8.2 describes the results of the MapReduce TM estimation experiments. Section 8.3 provides a performance evaluation of Spark streaming for running the Big Data traffic monitoring implementation. A point to reinforce is that the implementation of the MapReduce TM estimation is part of the implementation of the Big Data traffic monitoring method. Section 8.4 describes the results of the Big Data traffic analysis experiments.

8.1 Big Data Monitoring Results

Mininet was used to create the network topology and run the experiments. Using Iperf3, TCP flows were created according to the scheduling diagram shown in Figure 8.1. This diagram shows the traffic generated between pairs of hosts and the respective configured bandwidth. Some flow table entries were added to change the default route configuration provided by OpenDaylight. These flow entries define the IP routing, as shown in Table 8.1.

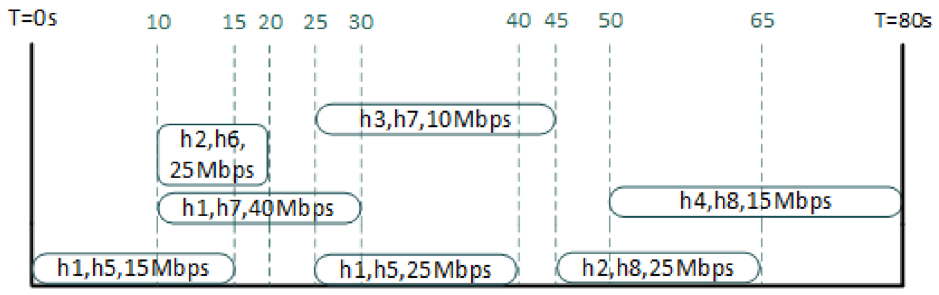


Figure 8.1: Monitoring scheduling diagram.

The experimental implementation probed the SDN controller every three seconds to collect traffic statistics. The collected data were processed by Spark streaming, and all statistics were persisted in Elasticsearch every three seconds. Even though the implementation provided all statistics in real-time, the graphics were generated off-line. The graphics provided in this chapter show statistics for selected resources and elements, but the implementation provided statistics for all monitored resources and elements. Except when explicitly stated, the statistics generated by this implementation denotes cumulative throughput. The scheduling diagram shown in Figure 8.1 displays the bandwidth selected for every unit of traffic generated. The bandwidth value is expressed in Mbits per second.

Switch	Source IP	Destination IP	Action
S1	10.0.0.1	10.0.0.5	output:eth3
	10.0.0.2	10.0.0.8	output:eth4
	10.0.0.3	10.0.0.7	output:eth4
S2	10.0.0.1	10.0.0.7	output:eth4
	10.0.0.2	10.0.0.6	output:eth3
	10.0.0.4	10.0.0.8	output:eth4
S3	10.0.0.1	10.0.0.5	output:eth3
	10.0.0.1	10.0.0.7	output:eth4
	10.0.0.2	10.0.0.6	output:eth4
	10.0.0.2	10.0.0.8	output:eth3
S4	10.0.0.3	10.0.0.7	output:eth3
	10.0.0.4	10.0.0.8	output:eth4
S5	*	10.0.0.5	output:eth1
	*	10.0.0.6	output:eth2
S6	*	10.0.0.7	output:eth1
	*	10.0.0.8	output:eth2

Table 8.1: IP routing table.

To validate the statistics provided, the packets on ports in the topology were captured using tcpdump. Using tshark and a python script, the throughput was calculated at every three seconds for each port and switch; the results were then plotted as graphics. This plot was intended to show that the statistics calculated in this study followed the same pattern as those calculated using tshark. With tshark, the statistics at every three seconds were calculated, and the counter values provided by the OF standard included seconds and nanoseconds, which did not enable exact matching of values at the nanosecond level. Hence, it cannot be expected that the values calculated using tshark and those provided by the present implementation will be the same.

8.1.1 Switch and Port Throughput

Figure 8.2 shows the evolution over time of the throughput in switch S6 and in each of its ports (eth1–eth4) during the simulation period. The graphic at the bottom provides the throughput for the switch.

The throughput for the switch was not originally provided by the SDN controller and is an aggregation of the value of its ports. The expression for this aggregation is shown in Equation 8.1:

$$S = \sum_{n=1}^{N} T_n, \tag{8.1}$$

where S is the switch throughput, T_n is the port n throughput, and N is the number of ports in the switch.

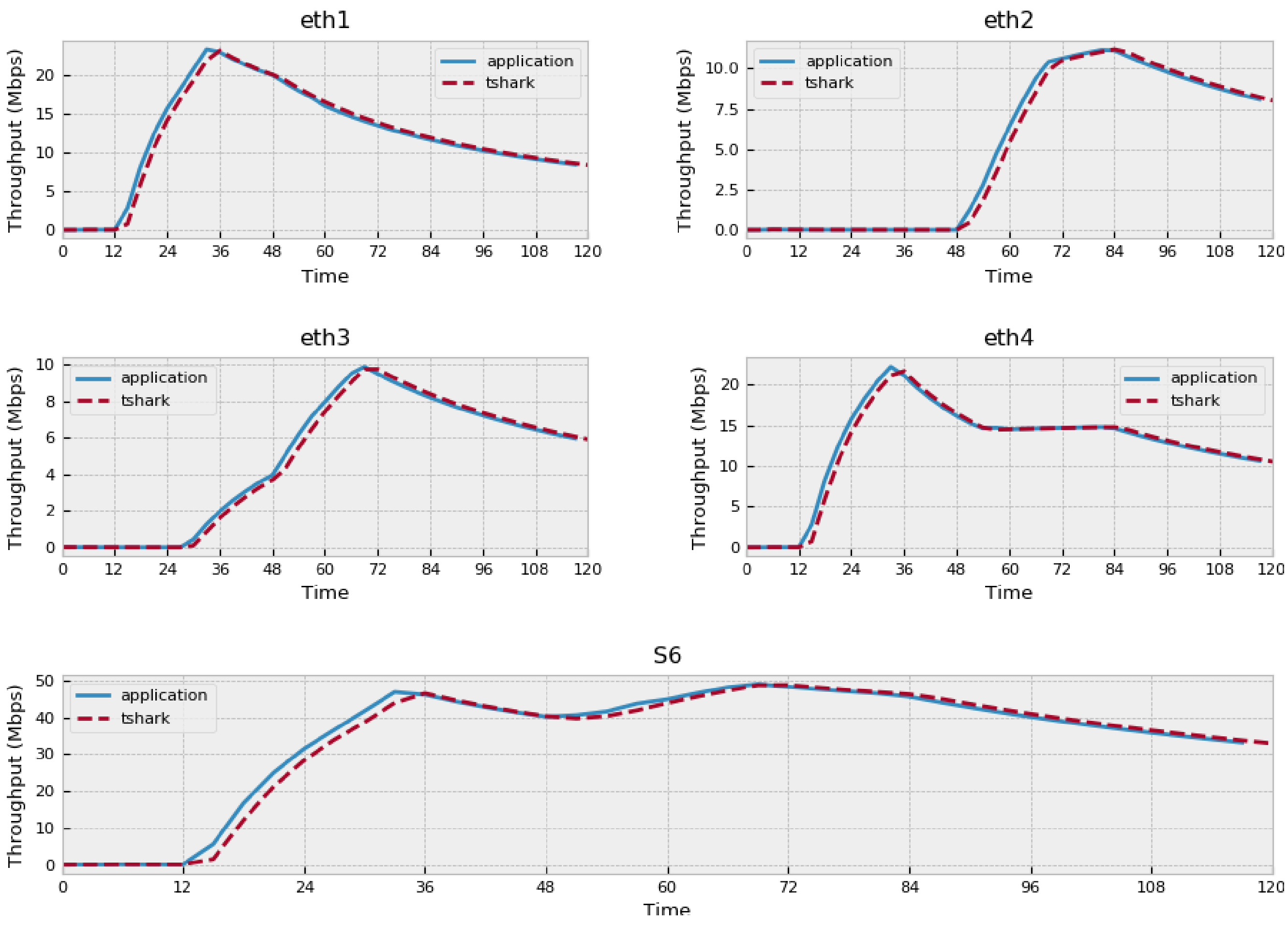


Figure 8.2: Switch and port throughput.

According to the schedule in Figure 8.1, a 20s burst of traffic is started from h1 to h7 at instant 10s. This is reflected on port eth1 where an increase in the throughput from instant 10 to 30 was observed. This increase continued until instant 30 because another burst of traffic at instant 25 was started from h3 to h7. The eth2 port reflects both the 20s traffic started from h2 to h8, and the 30s traffic started from h4 to h8. The eth3 port shows the 20s traffic between h3 and h7 and the 20s traffic from h2 to h8. The eth4 port shows the 20s traffic from h1 to h7 and the 30s traffic from h4 to h8.

8.1.2 Switch Throughput Breakdown

Another statistic provided by the present implementation represents the contribution (percentage) of the port throughput to the switch throughput, as can be seen in Figure 8.3. The graph shows the evolution over time of the contribution of each port to the switch load. For each port, the percentage was calculated according to the expression in Equation 8.2:

$$P = \frac{T_p}{T_s} * 100, \tag{8.2}$$

where T_p is the port throughput and T_s the switch throughput.

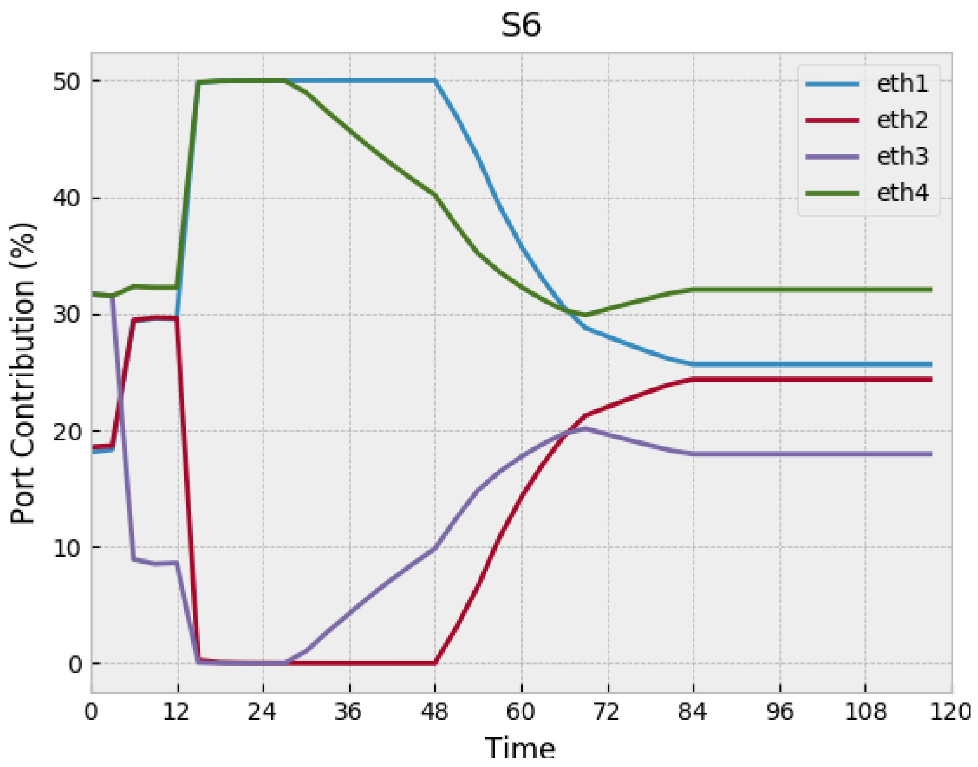


Figure 8.3: Port contribution to switch throughput.

Figure 8.2 shows that traffic starts after 10 seconds in S6. The percentage presented in Figure 8.3 from 0 to 10s is related to packets from other protocols such as MDNS, LLDP, ARP, and ICMP because the simulation included first pinging all machines to test connectivity. Figure 8.3 shows that from instant 10 to 25s, ports eth1 and eth4 generated all the traffic in the switch, which can be confirmed in Figure 8.2. Also, the eth2 port started its contribution to the switch load after instant 50s.

8.1.3 Switches Throughput

The throughput of every switch in the topology was calculated based on Equation 8.1. This procedure enabled tracking of the throughput evolution of all switches at the same time. The network topology contained six switches (S1–S6). Figure 8.4 shows the measured throughput and the tshark validation for switches S1, S3, and S4. In the first 15s, switches S1 and S3 have an increase in their throughput. S1 is in the route of the traffic from h1 to h5 for 15s. After 15s, S1 throughput decreased for 10s, but started to increase after 25s because of the rise in traffic from h1 to h5 and after 45s due to the (increased) traffic from h2 to h8 for 20s. S3 throughput grew faster because it was driven by the traffic from h1 to h5 and from h1 to h7 at 10s and again by the traffic from h1 to h5 at 25s. The traffic in S4 only started at 25s from h3 to h7 and saw a slight increase at 50s due to the traffic between h4 and h8.

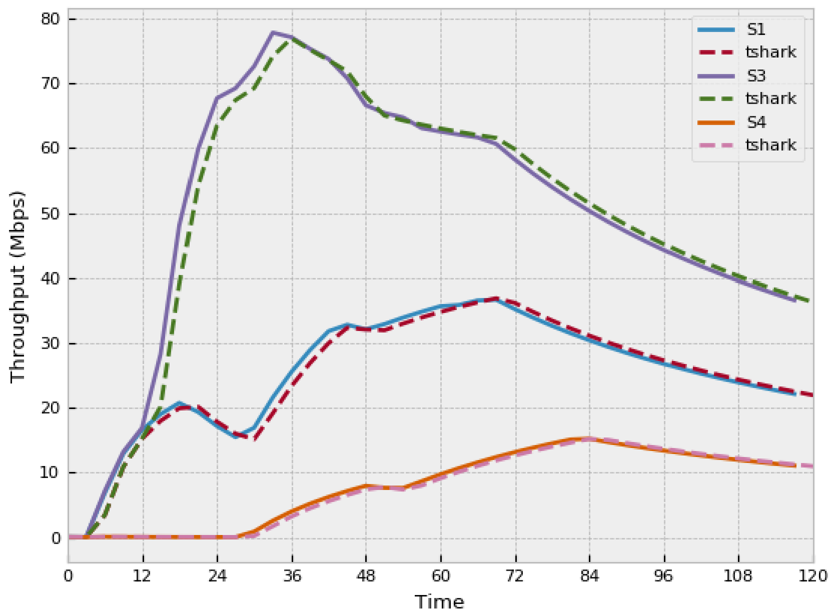


Figure 8.4: Switch throughput.

8.1.4 Port Capacity Usage

This study also examined the percentage of its own maximum throughput capacity that was used by each port. These statistics are shown in Figure 8.5. The percentage was calculated according to Equation 8.3:

$$P = \frac{T_p}{P_c} * 100, \tag{8.3}$$

where T_P is the port throughput and P_C the port throughput capacity. The P_C variable is provided by the *Switch Port* and is described in Table 4.2 as *Current Feature*. The value provided by OpenDaylight for the study environment was 1 Gbps. Because this value was too large for the experiments, a value of 100 Mbps was used.

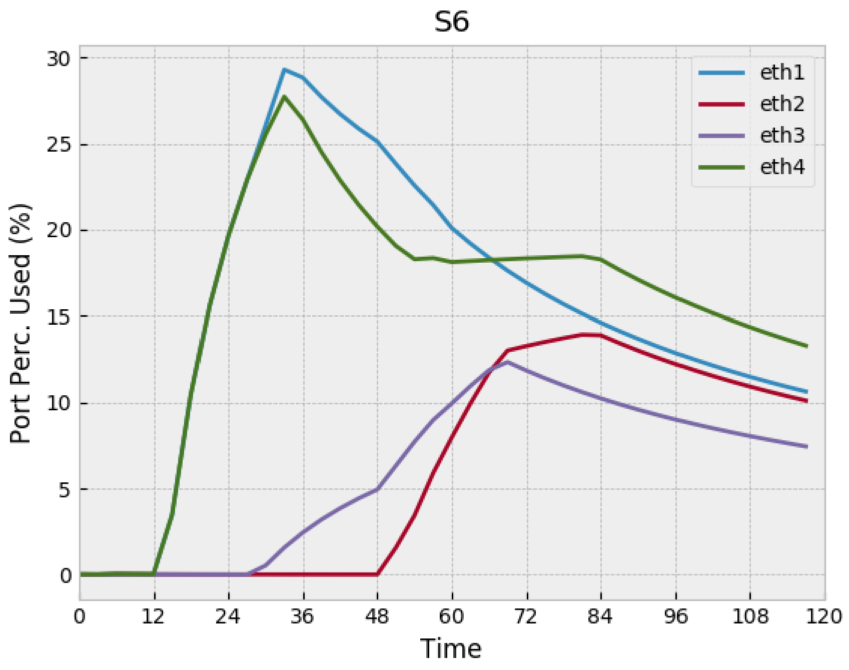


Figure 8.5: Port percentage capacity used.

Figure 8.5 shows the values of P calculated for all ports (eth1–eth4) in switch S6. The eth1 port reached its peak of utilization between 25 and 30s because of the traffic from h1 to h7 (10s) and h3 to h7 (25s). The eth2 port took more time to reach its peak because the traffic that traversed it only started at 45s (h2 to h8 and h4 to h8). The percentage of use for eth3 was affected by the traffic from h3 to h7 and from h2 to h8. Finally, eth4 had almost the same percentage of use as eth3 due to the traffic from h1 to h7 and h4 to h8.

8.1.5 Links Throughput

Because a network is composed of various links, the present implementation provides the throughput for every link in the topology. This graphic is shown in Figure 8.6. Links are formed between hosts and switches and between ports in a switch. If a link is between a host and a switch, the implementation simply collects the throughput of the switch port because the SDN controller does not provide statistics for the network interface attached to the host. If a link is established between switch ports, the implementation collects throughput from both ports. As shown in Table 4.3, the OF protocol defines the fields *Bytes Received* and *Bytes Transmitted* for port counters. If the implementation collects these counters for two ports that form a link at the same *Second* and *Nanosecond* (also defined in Table 4.3), these values are supposed to be the same, i.e., the *Bytes Received* of one port should be the same as the *Bytes Transmitted* of its partner and vice versa. Hence, the implementation only needs the counters of one of the ports in the link. Because the present implementation runs different threads to collect counters for each port, it is expected that the readings between pairs will not collect the same *Second* and *Nanosecond*. The following approach is used in this situation for a link formed by S_{a_i} and S_{b_j}:

- Define S_{a_i} as port i of Switch a.
- Define S_{b_j} as port j of Switch b.
- Define BR_{a_i} and BT_{a_i} as *Bytes Received* and *Bytes Transmitted* respectively for the reading of S_{a_i}.
- Define BR_{b_j} and BT_{b_j} as *Bytes Received* and *Bytes Transmitted* respectively for the reading of S_{b_j}.
- if $((BR_{a_i} + BT_{a_i}) > (BR_{b_j} + BT_{b_j}))$, use the reading of S_{a_i}; otherwise, use the reading of S_{b_j}.

This approach provides the most recent statistics about link throughput.

Figure 8.6 shows the calculated throughput in the links formed by S2:eth4 and S6:eth4, S1:eth1 and S3:eth3, S2:eth3 and S5:eth4, and S1:eth2 and S4:eth3. The throughput in link S2:eth4/S6:eth4 increases during the traffic from h1 to h7, experiences a small decrease, and then stops decreasing because of the traffic between h4 to h8. The throughput behaviour observed in link S1:eth1/S3:eth3 is driven by the traffic between h1 and h5 and between h2 and h8. The traffic from h2 to h6 is the only instance that uses link S2:eth3/S5:eth4. Link S1:eth2/S4:eth3 is used by the traffic from h3 to h7.

8.1.6 Current Link Throughput

The current throughput of a link is calculated according to equation 6.2. Figure 8.7 shows the current estimated throughput at every $t - T$ time interval in the links formed by S2:eth4 and

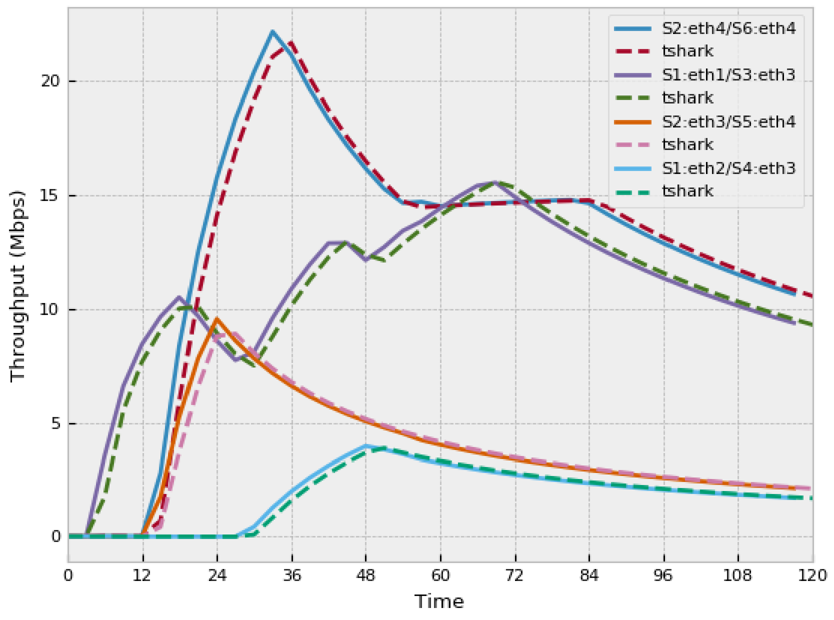


Figure 8.6: Links throughput.

S6:eth4, S1:eth1 and S3:eth3, and S2:eth3 and S5:eth4. The link S2:eth4/S6:eth4 is used by the traffic from h1 to h7 and from h4 to h8. S1:eth1/S3:eth3 carries the traffic from h1 to h5 (at 0 and 25s) and from h2 to h8. The traffic from h2 to h6 at instant 10s drives the throughput in S2:eth3/S5:eth4.

8.2 MapReduce TM Estimation Results

This section provides the implementation results of the MapReduce TM estimation method.

8.2.1 Host-to-Host Throughput

The OD pair accumulated throughput is also an aggregation of the values of individual links between the hosts. The MapReduce approach described in Chapter 6 provides an estimate of the throughput between all possible combinations of OD traversing S1 or S2 according to Table 8.2.

Figure 8.8 shows the estimated throughput between the following pairs of hosts: h1–h5 (root S1), h3–h8 (root S2), and h2–h4 (root S2). Table 8.1 provides the rules for verifying which ports are used by each OD pair. For instance, the following links connect h3 to h8 using S2 as a root: S4:eth1, S4:eth4/S2:eth2, s2:eth4/S6:eth4, and S6:eth2. The OD pair h1–h5 has

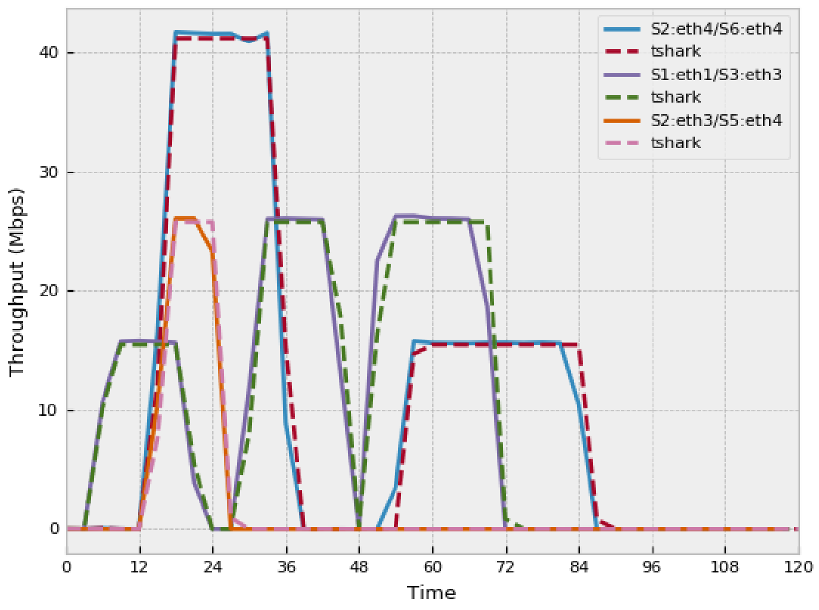


Figure 8.7: Links current throughput.

S_1	S_2
$h_1 - h_2$	$h_1 - h_2$
$h_1 - h_3$	$h_1 - h_3$
$h_1 - h_4$	$h_1 - h_4$
$\cdots$	$\cdots$
$h_7 - h_8$	$h_7 - h_8$

Table 8.2: Origin-Destination roots.

two specific traffic states defined at instants 0 and 25s. No traffic occurred between h3 and h8, meaning that the links that connect them were used to estimate throughput. In this case, the following traffic flows were used in throughput estimation: h1–h7, h3–h7, and h4–h8. The same situation applied to the OD pair h2–h4. In this case, the traffic between h1–h7, h2–h6, and h4–h8 generated the throughput for this OD pair.

8.2.2 Host-to-Host Current Throughput

As shown in Table 6.2, the current throughput of an OD pair is an aggregation of the values of individual links between the hosts. Figure 8.9 shows the estimated current throughput between h3 and h8 (root S2). The first four top graphics show the throughput on each link that connects the hosts, and the graphic on the bottom shows the aggregation of the individual links. The links between h3 and h8 are: S4:eth1, S2:eth2/S4:eth4, S2:eth4/S6:eth4, and S6:eth2. The only traffic flows in S4:eth1 is that from h3 to h7, which starts at 25s. The S2:eth2/S4:eth4 link

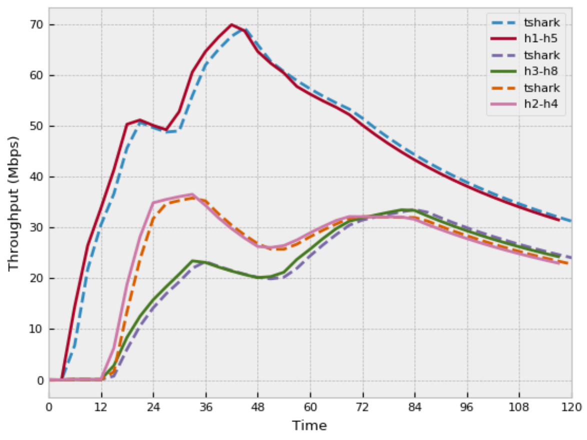


Figure 8.8: Origin-Destination throughput.

is used by traffic from h4 to h8, which starts at instant 50s. The traffic from h1 to h7 at instant 10s and from h4 to h8 at instant 50s drives the S2:eth4/S6:eth4 link throughput. The S6:eth2 link carries traffic from h2 to h8 at instant 45s and from h4 to h8 at instant 50s.

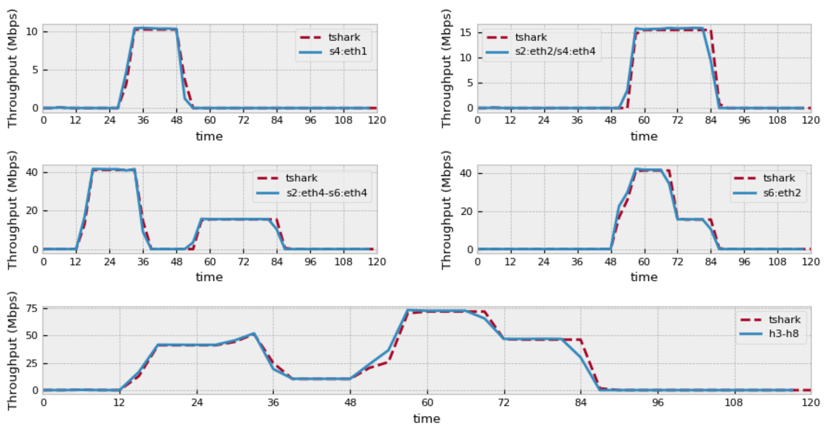


Figure 8.9: Origin-Destination current throughput.

8.3 Performance Evaluation

The proposed MapReduce approach was executed inside an Apache Spark Streaming environment. A performance evaluation of the execution of the proposed approach based on several metrics produced by the Spark Streaming monitoring system is now presented.

Figure 8.10 shows the record input rate statistics. The average value of 7.62 records per second is shown on the left side of the figure. The graph in the centre shows the evolution of the input rate over the simulation period. The first observed spike from left to right is related to the time that Spark takes to set up the environment, which causes an accumulation of records. Subsequent spikes are produced by the records arriving every 3 seconds. Spark streaming processes the input records in batches, and the histogram on the right side shows the distribution of batches/records per second.

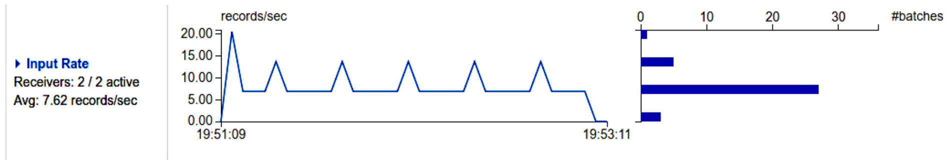


Figure 8.10: Record input rate.

Scheduling delay is the time between the instant that a collection of jobs for a batch was submitted and the instant that the first job was started. The scheduling delay the present simulation is shown in Figure 8.11. According to the graph in the centre, the first value for the scheduling delay was over 2 seconds because of the environment set-up, but the following batches had values near zero, making the overall average 173 ms. The histogram graph on the right shows that most of the batches had nearly zero scheduling delay time.

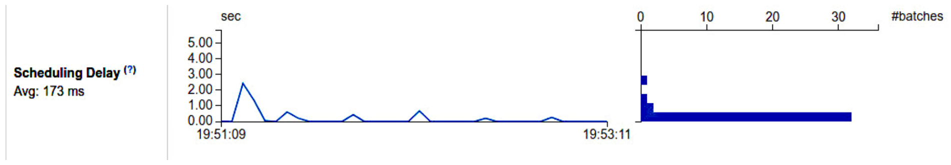


Figure 8.11: Scheduling delay.

Figure 8.12 shows the processing time for each batch of data. The average processing time in the present simulation was 2.5 seconds. According to the graph in the centre and the histogram in Figure 8.12, the most batches finished their execution in less than the time defined to process each batch (3.5 seconds, as shows by the dotted line labelled *stable*). The reason why this time is not the same for all batches is that the record input rate (Figure 8.10) is not the same. The record input rate is related to the Apache Kafka record processing time and affects the time that all MapReduce functions (Figure 6.1) take to run in a single batch. The time needed to persist each set of calculated statistics is already included in the batch processing time.

According to Spark Streaming documentation, the total delay parameter shows whether or not the system is stable. The system is considered stable if it can keep up with its record input rate. If the delay is maintained to be comparable to the batch size, the system is considered stable, which is true in the approach proposed here as shown in Figure 8.13.

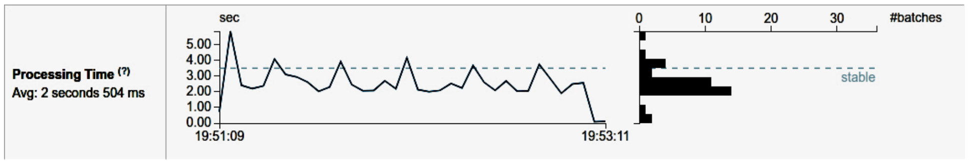


Figure 8.12: Processing time.

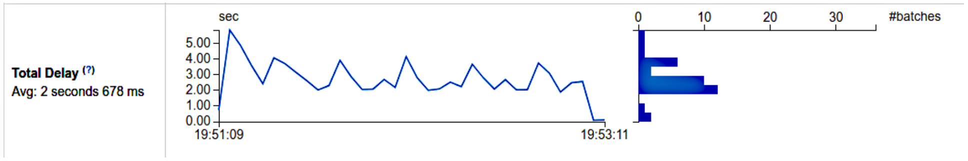


Figure 8.13: Total delay.

8.4 Big Data Traffic Analysis Method Results

This section provides the Big Data traffic analysis method (chapter 7) implementation results. The implementation followed the design guidelines presented in the right branch of the Lambda Architecture, as shown in Figure 2.10. The *Batch Layer* stored the object *Raw Traffic Data* (Figure 7.1), which contains the counter values collected from the SDN controller. The *Batch Processing Platform* runs all activities presented in Figure 7.1, except for the *Store Streaming Data* activity that processes the values stored in the *Batch Layer* and generated the traffic analysis. The *Serving Layer* stores the generated traffic analysis using the *Switch Traffic Analysis*, *Switch Port Traffic Analysis*, *Link Traffic Analysis*, and *Path Traffic Analysis* objects (Figure. 7.1).

Figure 8.14 depicts the traffic scheduling diagram for the experiments. Iperf3 generated the TCP flows and, Table 8.1 shows the route configuration entry flows that define the following roots for host-to-host communication: h1–h5, h3–h7, and h2–h8 use S1 as root, and h2–h6, h1–h7, and h4–h8 use S2 as root. Figure 6.3 presents the network topology.

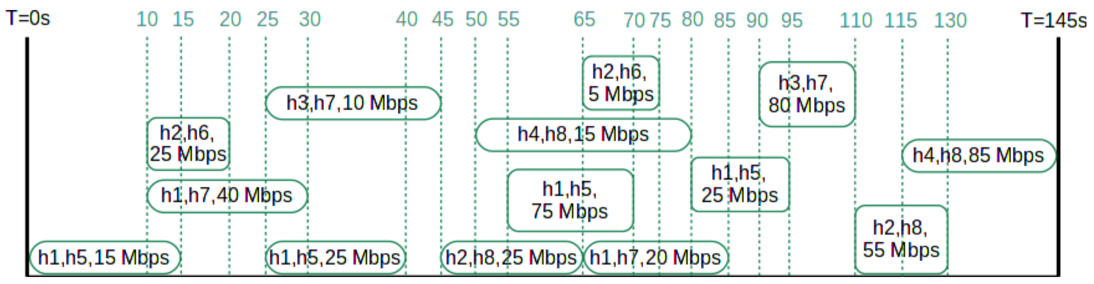
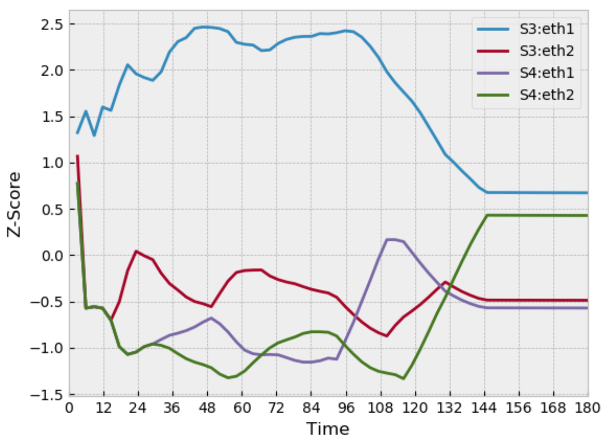


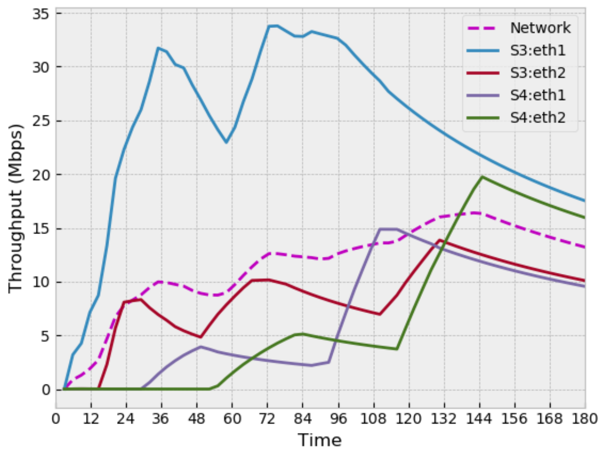
Figure 8.14: Batch scheduling diagram.

8.4.1 Switch Port Traffic Analysis

Figure 8.15 presents the z-score evolution in ports S3:eth1, S3:eth2, S4:eth1, and S4:eth2. The calculated z-score refers to the accumulated throughput in the port based on the average accumulated network throughput. Figure 8.15a provides the calculated z-score, and Figure 8.15b shows the ports' accumulated throughput and the network average accumulated throughput and also helps to show the consistency between the ports' calculated z-scores and their throughput.



(a) Switch ports Z-Score.



(b) Network throughput average.

Figure 8.15: Switch port traffic analysis.

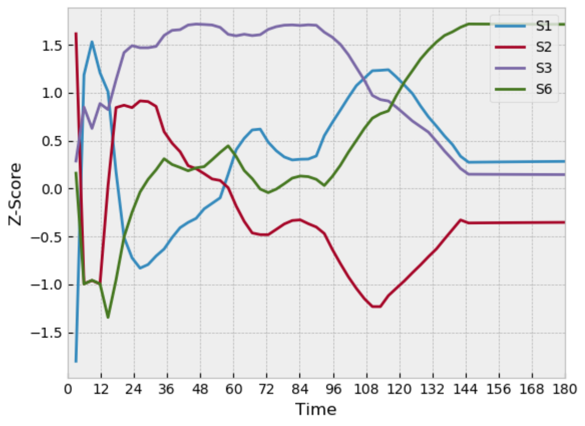
According to Figure 8.14, three bursts of traffic involving h1 (Figure 6.3) take place in the first 25 seconds. Figure 8.15a shows the z-score for S3:eth1. From instant 0 to 10s, the z-score has a high value because it is the only load on the network. Between 10 and 15s, the z-score increases because of the burst of traffic between h1–h7. After instant 15s, the z-score decreases for two reasons: first, the traffic between h1–h5 finishes, and second, a burst of traffic between h2–h6 increases the average network throughput and also changes the network throughput standard deviation. The S3:eth1 z-score keeps dropping up to instant 65s, when it increases because new bursts of traffic starts from h1–h5 at instant 55s, from h1–h7 at instant 65s, and from h1–h5 at instant 80s.

S3:eth2 has an increase in its z-score value at instant 15s because of the burst of traffic between h2 and h6 and also after instant 48s because of the traffic between h2–h8. Even though a burst of traffic occurs between h2 and h6 at instant 65s, the z-score of S3:eth2 decreases because this traffic throughput is 5 Mbps, which is much less than other throughputs in the same simulation period. S4:eth1, most of the time, presents negative z-scores because of the low throughput in the traffic burst from h3 to h7 at instant 25s and only reaches a positive value when its throughput goes above the network average value with the traffic burst from h3–h7 at instant 90s. S4:eth2 also has negative z-scores most of the time because of a burst of traffic at instant 50s between h4 and h8, but has a positive z-score because of another traffic burst between h4 and h8 at instant 115s and also because most of the other ports are decreasing their accumulated throughput, making the traffic in S4:eth2 go above the network average.

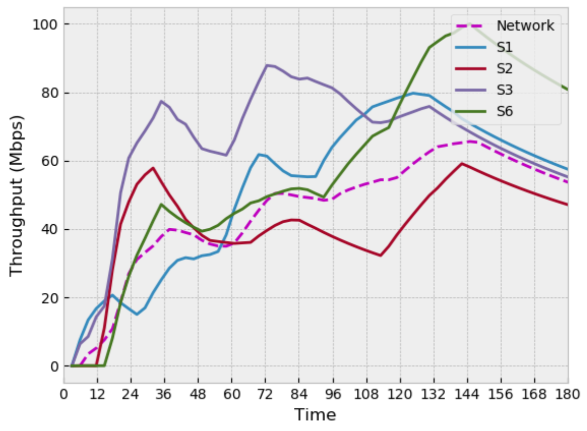
A benefit of the analysis provided by Figure 8.15 is that the port throughput presented in the graphs do not suggest anomalous behaviour according to the z-score *outlier* definition (Section 7.1.6).

8.4.2 Switch Traffic Analysis

Figure 8.16 shows the z-score changes in the switches compared to the network throughput average. Figure 8.16a shows the switches' z-scores, and Figure 8.16b shows their accumulated throughput and the average network accumulated throughput, and also helps to show the consistency between the switches' calculated z-scores and their throughput. unlike the previous section where the source for the average network throughput was the ports' accumulated traffic throughput, the source for the average network throughput in this section is the accumulated switch traffic throughput.



(a) Switch Z-Score.



(b) Network throughput average.

Figure 8.16: Switch traffic analysis.

Figure 8.16a shows that the S1 z-score increases from 0 to 10s because of the traffic between h1 and h5. The z-score decreases until instant 25s, when another burst of traffic starts from h1–h5 and increases until instant 70s because of the burst of traffic between h2–h8 at instant 45s and h1–h5 at instant 55s. After instant 70s, the z-score value decreases, but rises again after instant 80s because of the traffic between h1–h5, h3–h7 at instant 90s and between h2–h8 at instant 110s.

The S3 z-score value from 0 to 10s is greater than zero because of the burst of traffic between h1–h5. At instant 10s, the S3 z-score increases because of the bursts of traffic between h1–h7, between h2–h6, and between h1–h5 at instant 25s. At instant 40s, the S3 z-score decreases because all traffics ceases, but rises around instant 50s because of the bursts of traffic between h2–h8 at instant 45, between h1–h5 at instant 55s, and between h2–h6 at instant 65s. Around instant 80s, the z-score decreases because the remaining traffic in S3 is between h1–h5 and stabilizes between 0 and 0.5 because of the traffic between h2–h8 at 110s.

The burst of traffic between h2–h6 and h1–h7 at instant 10s increases the S2 z-score. At instant 30s, the z-score decrease because traffic ceases between h1–h7. Despite the burst of traffic between h4–h8 at instant 50s, the z-score value continues to drop because of the increase in the network switch average. The bursts of traffic between h2–h6 and h1–h7 increase the S2 z-score, but at instant 80s, it decreases again because all traffic has ceased. At instant 115s, the z-score increases again because of the burst of traffic between h4–h8.

The S6 switch deals with all traffic involving hosts h7 and h8. The bursts of traffic from h1–h7 at instant 10s, h3–h7 at instant 25s, h2–h8 at instant 45s, and h4–h8 at instant 50s make the z-score increase until instant 55s. After instant 55s, the z-score shows a slight decrease, most likely because of the traffic between h1–h5 that increases the average network throughput and then increases again because all the bursts of traffic after instant 90s have hosts h7 and h8 as their destination.

8.4.3 Link Traffic Analysis

Besides the z-scores provided in the previous section, the data generated by the proposed Big Data traffic analysis method make it possible to analyze a variety of traffic phenomena, such as the distribution of traffic throughput in the network links. This analysis helps activities such as load balancing and fault tolerance.

Figure 8.17 shows the distribution of network traffic throughput, identifying the minimum, median, and maximum values and the first and third quartile distributions. The median is the number such that 50% of the ordered data is less than or equal to that number. The first quartile is the number such that 25% of the data is less than or equal to that number, and the third quartile is the number such that 75% of the data is less than or equal to that number [94]. The symbol ▲ inside the boxes represents the average value of the data set for each link, and the symbol ○ represents *outliers*.

According to Figure 8.17, among the links present in the graphic, link S1:eth1/S3:eth3 presents the highest accumulated throughput value, but its average value is nearly the same as link S2:eth4/S6:eth4. S2:eth2/S4:eth4 presents the lowest accumulated throughput because traffic between h4–h8 crosses the link and has only two bursts of traffic (instants 50s and 115s). Traffic between h4–h8 and h1–h7 crosses the link S2:eth4/S6:eth4, meaning that 25% of the

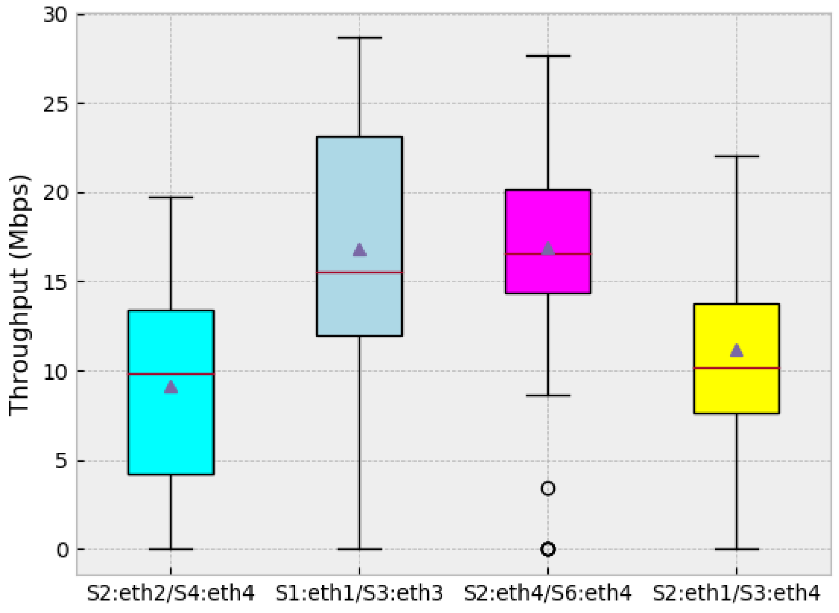


Figure 8.17: Links throughput distribution.

accumulated throughput lies between 8 and 14 Mbps. The link S2:eth1/S3:eth4 deals with the traffic between h1–h7 and h2–h6 and presents a better distribution of the accumulated throughput.

8.4.4 Path Traffic Analysis

Path traffic analysis provides the average accumulated throughput between each host OD pair according to the scheduling diagram presented in Figure 8.14. Equation 8.4 calculates the average for each pair of host OD pair:

$$Avg = \frac{1}{N}\sum_{N=1}^{N} T_n, \tag{8.4}$$

where Avg is the average throughput, N is the number of calculated throughputs, and T_n is the throughput of the n-th calculated throughput for the path.

Figure 8.18 shows the calculated average throughput. The path between h1 and h7 has the highest average throughput. The following analysis explains this result: the path between h1–h7 crosses links S3:eth1, S3:eth4/S2:eth1, S2:eth4/S6:eth4, and S6:eth1. Every unit of traffic crossing these links adds load to the link and consequently adds traffic between h1–h7. Table 8.3 shows the traffic that adds load to the links between h1–h7 (root S2).

Table 8.3 shows that, besides the traffic between h1–h7, the traffic from h1–h5, h2–h6, h3–h7, and h4–h8 will add load to the path between h1–h7, and Figure 8.14 shows that the bursts

Traffic	**Link**
h1–h7	All in the path
h1–h5	S3:eth1
h2–h6	S3:eth4/S2:eth1
h3–h7	S6:eth1
h4–h8	S2:eth4/S6:eth4

Table 8.3: Links h1–h7.

of traffic between the mentioned links have high throughput and confirms the result presented in Figure 8.18.

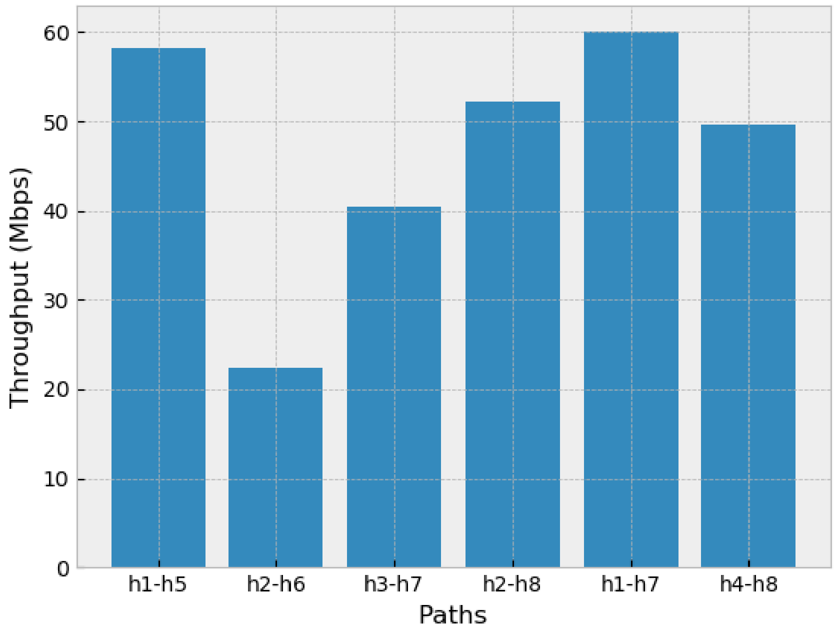


Figure 8.18: Path average throughput.

The path between h2–h6 has the lowest throughput average and crosses links S3:eth2, S3:eth4/S2:eth1, S2:eth3/S5:eth4, and S5:eth2. Table 8.4 shows the traffic and links that add load to the path between h2 and h6 (root S2). Figure 8.14 shows the bursts of traffic on the links between h2 and h6, and the combination of few links and low throughput confirms the obtained result.

Traffic	**Link**
h2–h6	All in the path
h1–h7	S3:eth4/s2:eth4
h2–h8	S3:eth2

Table 8.4: Links h2-h6.

8.5 Summary

This chapter has presented the experimental results obtained by implementing of the methods proposed in this thesis, namely, Big Data traffic monitoring method, the MapReduce TM estimation method, and the Big Data traffic analysis method. Furthermore, the chapter has presented a performance evaluation of Spark streaming to show the stability of the system when processing the counte values that were received in a streaming fashion and generating the statistics provided in Sections 8.1 and 8.2.

The results described in this chapter show that the proposed methods can deliver insights into network traffic, but that the implementation has some limitations. The traffic bandwidth used in the experiments has an Mbps scale, and the three-second interval to provide the statistics seems appropriate. For traffic bandwidth with a Gbps scale, the interval becomes too long because traffic congestion or imbalance can occur and not be promptly detected. The implementation needs improvement to be able to collect and provide traffic statistics within an interval of less than three seconds.

The experiments used one SDN controller, but the proposed methods can be extended to use more than one. This approach impacts *data acquisition* (Figure 4.1) because more than one controller must be polled, which affects the time between traffic data requests. The time interval between sequential data collection needs to be increased to accommodate all data collection threads. The time to process the collected data may not be affected if more than one processing cluster is available to generate the statistics.

The main goal of this chapter was to demonstrate the ability of the proposed methods to provide fine-grained statistics on network resources. Even though the statistics provided in Sections 8.1 and 8.2 are depicted in separate graphics, they were all generated in the same time interval, every three seconds, which proves the efficiency of the proposed monitoring method. The results depicted in Section 8.4 also show the efficiency of the proposed traffic analysis method. It was not the intention of this work to prove that the chosen statistics were the most appropriate for traffic analysis, but that the proposed methods enable the use of different statistics that can be application-dependent.

Despite the number of results presented in this chapter, the ideal scenario to provide a complete evaluation of the proposed methods would be an environment where the results provided in this chapter were used by other TE activities to make decisions about new policies and to deploy these policies in the network. The new configuration provided by the new policies would be evaluated to measure the benefits of the data provided by the proposed methods.

The next chapter concludes this thesis by reviewing its contributions and suggesting future directions in this research area.

Chapter 9

Conclusions

This thesis has investigated the problem of traffic monitoring in SDN and provided a series of contributions that can leverage TE activities. Among TE activities (Figure 2.7), traffic monitoring and analysis provide information about the behaviour of network parameters in the network components. For instance, to detect the need for switch load balancing, the amount of traffic crossing that switch is critical information. The more information that is available, the better will be the result of the action to solve the problem. Building on a number of studies tackling the traffic monitoring problem, this thesis has investigated and proposed methods to address following problems:

- Traffic Monitoring
  - The lack of a monitoring method that enables fine-grained monitoring of network resources. Fine-grained monitoring is beneficial, not only for network operators but also for algorithms trying to solve problems such as switch load balancing, controller load balancing, and fault tolerance for the control and data planes.
- Traffic Analysis
  - The lack of a traffic data analysis method to reveal network traffic trends and behaviour over time. unlike monitoring, data analysis deals with historical network data and consequently requires a different approach in terms of organizing and processing traffic data.

These problems are closely related because the monitoring activity collects network data, calculates the network parameter(s), and hands them over to an application or a network administrator. The traffic data analysis activity will use these same collected data, but will focus on dissecting their evolution over time to provide more advanced insights into network traffic.

Consequently, both activities use the same data, but with distinct goals. This thesis significantly advances the TE activity of traffic monitoring and analysis by proposing methods that can manage both problems using Big Data tools that standardize both processes. These methods enable designers to focus on the type of monitoring, e.g. network parameter(s), and the analysis desired instead of dealing with problems like how to aggregate data at different levels of the hierarchy and how to distribute the tasks that process the aggregations.

9.1 Contributions

This thesis has delivered the following contributions toward solving previously mentioned problems:

- Traffic Monitoring
  - This research has proposed a Big Data traffic monitoring method that considers the collection and processing of counter values related to flows, flow tables, and ports. Because the counters are collected in a streaming fashion, Big Data streaming processing tools can deal with the problem of temporarily storing and providing counter values for further processing.
  - The implementation of the proposed Big Data traffic monitoring method using Big Data tools delivered statistical information such as the ratio of each port in the switch load and the throughput capacity used on each port, path, and switch. One of the most significant benefits of using Big Data tools is that during implementation, the focus is on the procedure to calculate the statistics, not on how to aggregate the calculated statistics per resource and parallelize the processing.
  - A critical source of information about the traffic in the network is the TM. The TM is generated during the calculation of statistical information. This thesis has provided a MapReduce approach to estimate the TM in real time based on the traffic crossing the links between the OD pairs of hosts. In addition to the TM estimate based on the accumulated throughput in each link, this research has also delivered the TM estimate obtained between two sequential traffic data collections, which is called the current TM throughput. The estimate was by calculating the current throughput in each link and then aggregating the calculated values for each pair of OD hosts.
- Traffic Analysis

  - This research has proposed a traffic data analysis method based on Big Data techniques. Because a large amount of data is required to perform traffic analysis, the proposed method is designed to execute batch processing. The method establishes a hierarchical data dependency starting from the basic statistics as calculated in the Big Data traffic monitoring method, namely, the ratio of each port in its switch load and the throughput capacity used on each port, path, and switch. Using these basic statistics, the method starts to build up historical and more advanced traffic analyses. The hierarchical data dependency is generic enough to be used for different analyses as provided in the experiments described in Section 8.4. The contribution of the proposed method is the standardized flow of steps to carry out traffic analysis and analyze its data dependencies.

As a result of the contributions described above, two other contributions can be highlighted:

- The methods proposed in this thesis were implemented using direct measurement, an approach that was not considered feasible because of the lack of infrastructure to collect and process traffic data and the amount of data to be collected and processed. With the delivered implementation, this research opens the possibility of direct measurement because of the use of Big Data tools that were designed to deal with this type of approach.

- Both proposed methods have the same source of data, i.e., counter data collected from the SDN controller. In this case, it is beneficial that the Big Data processing system for both methods can use this source of data without the problem of collecting the same data twice and the risk of data inconsistency. This research uses Lambda Architecture to carry out both streaming and batch processing to solve the issue of redundant data collection and inconsistency. The Lambda Architecture was designed for this specific purpose and enables a diverse collection of Big Data tools to be combined to process the incoming data.

9.2 Future Work

This research study has made significant contributions, but traffic monitoring and analysis in SDN have several open challenges. TE can significantly benefit from the advances in the traffic monitoring and analysis area, but both traffic monitoring and analysis are in their early stages in SDN.

Figure 9.1 shows the TE scope and highlights the topics covered by this research study, namely *Traffic Analysis/Characterization* (**Monitoring**, **Traffic Analysis**). The topics of *check-*

ing network invariants and *debugging programming errors* are outside the scope of this research. At this stage, the *Topology Update* activity does not benefit from the contributions of this thesis, but the *Flow Management* and *Fault Tolerance* activities do benefit from the methods proposed in this thesis.

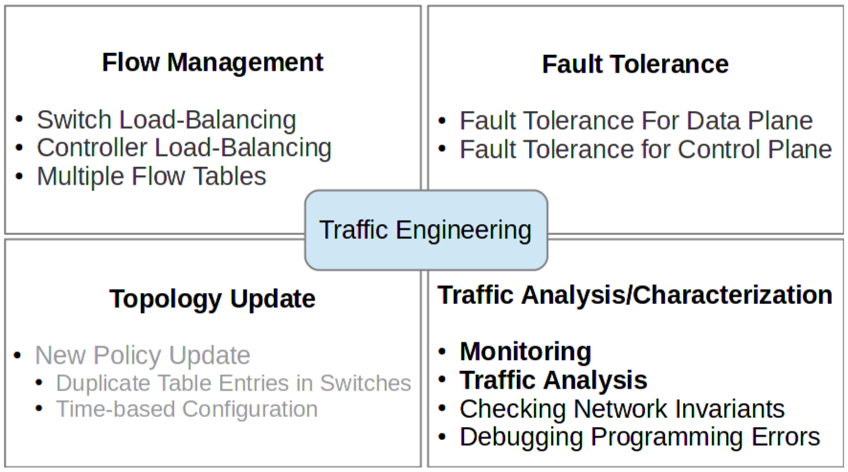

Flow Management
• Switch Load-Balancing
• Controller Load-Balancing
• Multiple Flow Tables
Fault Tolerance
• Fault Tolerance For Data Plane
• Fault Tolerance for Control Plane
Traffic Engineering
Topology Update
• New Policy Update
• Duplicate Table Entries in Switches
• Time-based Configuration
Traffic Analysis/Characterization
• Monitoring
• Traffic Analysis
• Checking Network Invariants
• Debugging Programming Errors

Figure 9.1: TE scope.

The following sections propose future research topics that can extend this work.

9.2.1 Traffic Engineering

- Several research topics in the TE area propose algorithms to solve problems such as switch load balancing and fault tolerance. These algorithms do not take into account the historical behaviour of resources such as links, ports, and switches. Using the results provided by the monitoring methods proposed in this researchn new algorithms for TE problems can be suggested to obtain better results.

- New research areas are using machine learning algorithms to perform tasks such as traffic prediction, traffic classification, and anomaly detection in SDN. These research topics deal with a small range of features collected from the network, usually values collected from the counters and traffic throughput. New machine learning algorithms can be suggested using the insights delivered by the proposed methods by using them as a new set of features.

9.2.2 Traffic Monitoring and Analysis

- The first task in the proposed methods is to collect counters from the SDN controller. Threads collect these counters in a round-robin fashion, i.e, for each network resource, one thread collects the counter values and is run again only after all the other threads collect their resource values. An adaptive algorithm can be designed based on the evolution of traffic in that specific resource, enabling the collection procedure to adapt itself according to the network traffic. Eventually, some resources can receive more traffic and have priority in the collection task.

- The implementation of the monitoring methods provides results within a three second range. For networks with high bandwidth availability, this time range may not be ideal because of the rate at which packets are created. An investigation on how to decrease this time range while keeping the same consistent results would be beneficial for high-speed networks. Decreasing the time range is quite challenging because the proposed method uses threads to send requests to the SDN controller. By decreasing the time interval between counter readings, the number of requests sent by the threads to the controller per unit of time will increase.

- Figure 6.2 shows the layout of the messages containing counter values. The field *second* is of particular interest because it is used to calculate traffic throughput, and its regular change keeps the changes in the statistics smooth, especially when calculating the current values according to Equation 6.2. The idea is to collect values at a regular interval of seconds, for instance, at second 3, 6, 9, 12, 15, and so on. During the experiments, because of the round-robin nature of the thread scheduling, some reading sequences did not provide a regular period, resulting in, for instance, a sequence of seconds of 3, 6, 12, 18, and so on. The uneven gap between the readings makes the transition change abruptly and not be smooth. part of the counters' collectors task should be to detect this gap and create a record for the gap according to some rules using, for example, the average values between the current reading and the last reading.

- The *Data Acquisition* component (Figure. 5.1) collects data using the northbound APIs available in the SDN controller. An SDN controller performs a large number of tasks and eventually can have its performance affected by the number of statistics requests issued by the *Collect Streaming Data* component. Another approach to minimizing the adverse effects of a high rate of statistics requests in the SDN controller would be to collect the counters values for messages sent to the OF switches, bypassing the SDN controller. This approach could enable more readings request and consequently improve the quality of the statistics provided.

- The *Data Acquisition* component (Figure 5.1) logically belongs to the application layer of the SDN architecture (Figure 2.2). Most SDN controllers nowadays make it possible to add new features to the controller by attaching new modules to it. For instance, OpenDaylight allows the attachment of new modules using the OSGi standard. The *Data Acquisition* component can then be moved from the application layer to the control-plane layer to decrease the number of layers crossed to reach the counters of an OF switch.

- The topology used to validate the proposed monitoring method is limited in the number of hosts, switches, and ports; these numbers could be increased to verify the scalability of the Big Data tools used in the proposed implementation.

- Network topologies using segments can be used to expand the proposed method. Network segments will require at least one SDN controller to manage every segment, and the output results from every segment, which are generated by a local monitoring application, should be sent to a highly hierarchical monitoring application to summarize the traffic for all segments.

- The proposed monitoring method considers the collection of counters for flow tables and flows, but the implementation developed here collected only port traffic counters. A new implementation should be developed to collect flow tables and flows to analyze the impacts of these new values on the performance of the *Data Acquisition*, *Data Aggregation*, and *Data Persistence* components (Figure 5.1).

- The proposed MapReduce TM estimation method was designed to be generic from the network topology point of view. The topology used in the experiments provided some challenges for the method, such as more than one root path between the same OD host pair. The method overcame this problem by creating separate *keys* for each path. Despite the flexibility of the MapReduce paradigm, the proposed method should be applied to more network topologies to verify its capability to adapt to different network topologies and its performance in delivering the estimated TM.

- In this research, the network parameter monitored was the *throughput* because its value is easily obtained from the counter values. The proposed methods can also be applied to other network parameters, with the difference that some parameters such as *packet loss* and *network delay* are harder to obtain and could impact the performance of the *Data Acquisition*, *Data Aggregation*, and *Data Persistence* components (Figure 5.1).

- The current implementation of the Big Data traffic monitoring method provided new insights into network behaviour, such as the ratio of each port to its switch load and the throughput capacity used on each port, path, and switch. This contribution opens up the possibility of investigating whether, based on these insights or new ones, new network parameters can be suggested. New network parameters improve the quality of traffic monitoring because of the increment in the number of network performance indicators.

- The methods proposed in this research provided real-time statistics and traffic analysis based on batch processing. The Lambda Architecture design enables real-time values to be compared with batch values with the aim of verifying whether a real-time value lies in a healthy range compared with historical data or whether it is outside that range. Activities such as anomaly detection would benefit from this comparison.

Bibliography

[1] H. Farhady, H. Lee, and A. Nakao, “Software-Defined Networking: A survey,” *Computer Networks*, vol. 81, pp. 79–95, 2015.

[2] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, “A roadmap for traffic engineering in software defined networks,” *Computer Networks*, vol. 71, pp. 1–30, 2014.

[3] D. Kreutz, F. M. V. Ramos, P. E. Verísssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-Defined Networking : A Comprehensive Survey,” *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[4] I. Z. Bholebawa, R. K. Jha, and U. D. Dalal, “Performance Analysis of Proposed OpenFlow-Based Network Architecture Using Mininet,” *Wireless Personal Communications*, vol. 86, no. 2, pp. 943–958, 2015.

[5] ONF, “OpenFlow Switch Specification 1.4.0,” 2013.

[6] K. Grolinger, M. Hayes, W. A. Higashino, A. L’Heureux, D. S. Allison, and M. A. Capretz, “Challenges for MapReduce in Big Data,” in *2014 IEEE World Congress on Services*, pp. 182–189, IEEE, Jun. 2014.

[7] X. Shu, F. Liu, and D. (Daphne) Yao, “Rapid Screening of Big Data Against Inadvertent Leaks,” in *Big Data Concepts, Theories, and Applications* (S. Yu and S. Guo, eds.), pp. 193–235, Cham: Springer International Publishing, 2016.

[8] N. Marz and J. Warren, *Big Data - Principles and Best Practices of Scalable Real-Time Data Systems*. Manning Publications Co., 2015.

[9] T. L. Foundation, “Opendaylight project.” `https://www.opendaylight.org/`.

[10] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, “Software defined networking: State of the art and research challenges,” *Computer Networks*, vol. 72, pp. 74–98, 2014.

[11] H. Song, "Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), pp. 127–132, ACM, 2013.

[12] M. Smith, M. Dvorkin, Y. Laribi, V. Pander, P. Garg, and W. N., "Opflex control protocol," Apr 2014.

[13] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, 2014.

[14] M. Sune, V. Alvarez, T. Jungel, U. Toseef, and K. Pentikousis, "An OpenFlow Implementation for Network Processors," in *2014 Third European Workshop on Software Defined Networks*, no. Cmm, pp. 123–124, IEEE, Sep. 2014.

[15] O. N. Foundation, "Openflow switch specification 1.5.0," 2014.

[16] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "Research challenges for traffic engineering in software defined networks," *IEEE Network*, vol. 30, pp. 52–58, 2016.

[17] Z. Shu, J. Wan, J. Lin, S. Wang, D. Li, S. Rho, and C. Yang, "Traffic engineering in software-defined networking: Measurement and management," *IEEE Access*, vol. 4, pp. 3246–3256, 2016.

[18] P. Megyesi, A. Botta, G. Aceto, A. Pescapé, and S. Molnár, "Challenges and solution for measuring available bandwidth in software defined networks," *Computer Communications*, vol. 99, pp. 48–61, Feb. 2017.

[19] N. L. M. Van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow software-defined networks," *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*, 2014.

[20] D.-H. Luong, A. Outtagarts, and A. Hebbar, "Traffic Monitoring in Software Defined Networks Using Opendaylight Controller," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10026 LNCS, pp. 38–48, 2016.

[21] H. Tahaei, R. B. Salleh, M. F. Ab Razak, K. Ko, and N. B. Anuar, "Cost Effective Network Flow Measurement for Software Defined Networks: A Distributed Controller Scenario," *IEEE Access*, vol. 6, pp. 5182–5198, 2018.

[22] J. Suárez-Varela and P. Barlet-Ros, "Flow monitoring in Software-Defined Networks: Finding the accuracy/performance tradeoffs," *Computer Networks*, vol. 135, pp. 289–301, Apr. 2018.

[23] A. A. Safaei, "Real-time processing of streaming big data," *Real-Time Systems*, pp. 1–44, 2016.

[24] Q. Zhao, Z. Ge, J. Wang, and J. Xu, "Robust traffic matrix estimation with imperfect information," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, p. 133, Jun. 2006.

[25] K. Xie, C. Peng, X. Wang, G. Xie, and J. Wen, "Accurate recovery of internet traffic data under dynamic measurements," *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pp. 1–9, 2017.

[26] K. Papagiannaki, N. Taft, and A. Lakhina, "A distributed approach to measure IP traffic matrices," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement - IMC '04*, (New York, New York, USA), p. 161, ACM Press, 2004.

[27] J. Lin, "In Defense of MapReduce," *IEEE Internet Computing*, vol. 21, pp. 94–98, May 2017.

[28] R. Masoudi and A. Ghaffari, "Software defined networks: A survey," *Journal of Network and Computer Applications*, vol. 67, pp. 1–25, May 2016.

[29] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[30] O. N. Foundation. `https://www.opennetworking.org/sdn-definition/`.

[31] W. Stallings, "Software-Defined Networks and OpenFlow," *The Internet Protocol Journal*, vol. 16, pp. 2–14, 2013.

[32] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 16, pp. 1–20, 2014.

[33] A. Hakiri, A. Gokhale, P. Berthou, D. C. Schmidt, and T. Gayraud, "Software-defined networking: Challenges and research opportunities for future internet," *Computer Networks*, vol. 75, pp. 453–471, 2014.

[34] H. Yin, H. Xie, T. Tsou, D. Lopez, P. Aranda, and R. Sidi, "SDNi: A message exchange protocol for software defined networks (SDNS) across multiple domains," *Internet Engineering Task Force, Internet Draft*, 2012.

[35] Y. Guo, Z. Wang, X. Yin, X. Shi, and J. Wu, "Traffic engineering in SDN/OSPF hybrid network," *Proceedings - International Conference on Network Protocols, ICNP*, pp. 563–568, 2014.

[36] B. Niven-Jenkins, D. Brungard, M. Betts, N. Sprecher, and S. Ueno, "Requirements of an MPLS Transport Profile," *RFC 5654*, 2009.

[37] S. J. Samuel, K. Rvp, K. Sashidhar, and C. R. Bharathi, "a Survey on Big Data and Its Research Challenges," *ARPN Journal of Engineering and Applied Sciences*, vol. 10, no. 8, pp. 3343–3347, 2015.

[38] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, no. 2, pp. 137–144, 2015.

[39] D. Laney, "3D Data Management: Controlling Data Volume, Velocity, and Variety," *Application Delivery Strategies by META Group Inc.*, vol. 949, p. 4, 2001.

[40] A. D. Meshram, A. S. Kulkarni, and S. S. Hippargi, "Big Data Analytics using Real-Time Architecture," *International Journal of Latest Trendes in Engineering and Technology(IJLTET)*, vol. 6, no. 4, pp. 241–247, 2016.

[41] J. A. Stankovic, S. H. Son, and J. Hansson, "Misconceptions About Real-Time Databases," *Computer*, vol. 32, pp. 29–36, 1999.

[42] M. Bralow, *Real-Time Big Data Analytics: Emerging Architecture*. O'Reilly Media Inc., first ed., 2013.

[43] S. Shahrivari, "Beyond Batch Processing: Towards Real-Time and Streaming Big Data," *Computers*, vol. 3, no. 4, pp. 117–129, 2014.

[44] A. L'Heureux, K. Grolinger, H. F. Elyamany, and M. A. M. Capretz, "Machine Learning With Big Data: Challenges and Approaches," *IEEE Access*, vol. 5, pp. 7776–7797, 2017.

[45] E. B. Claise, "Cisco systems netflow services export version 9," RFC 3954, RFC Editor, 2014.

[46] "sflow." `http://sflow.org/about/index.php`.

[47] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, “FlowSense: Monitoring network utilization with zero measurement cost,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7799 LNCS, pp. 31–41, 2013.

[48] L. Jose, M. Yu, and J. Rexford, “Online measurement of large traffic aggregates on commodity switches,” in *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE’11, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2011.

[49] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, “Planck: Millisecond-scale Monitoring and Control for Commodity Networks,” in *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM ’14*, (New York, New York, USA), pp. 407–418, ACM Press, 2014.

[50] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, “PayLess: A low cost network monitoring framework for Software Defined Networks,” in *2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–9, May 2014.

[51] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, “OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN,” in *2014 IEEE 34th International Conference on Distributed Computing Systems*, pp. 228–237, IEEE, Jun. 2014.

[52] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with opensketch,” *... 10th USENIX Symposium on Networked Systems ...*, pp. 29–42, 2013.

[53] Y. Sinha, S. Vashishth, and K. Haribabu, “Real Time Monitoring of Packet Loss in Software Defined Networks,” in *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, vol. 218, pp. 154–164, 2018.

[54] J. Suarez-Varela and P. Barlet-Ros, “Towards a NetFlow Implementation for OpenFlow Software-Defined Networks,” in *2017 29th International Teletraffic Congress (ITC 29)*, vol. 1, pp. 187–195, IEEE, Sep. 2017.

[55] X. T. Phan and K. Fukuda, “SDN-Mon: Fine-Grained Traffic Monitoring Framework in Software-Defined Networks,” *Journal of Information Processing*, vol. 25, no. 0, pp. 182–190, 2017.

[56] J. Suárez-Varela and P. Barlet-Ros, “Reinventing NetFlow for OpenFlow Software-Defined Networks,” 2017.

[57] M. Hartung and M. Körner, “SOFTmon - Traffic Monitoring for SDN,” *Procedia Computer Science*, vol. 110, pp. 516–523, 2017.

[58] L. Hendriks, R. D. O. Schmidt, R. Sadre, J. A. Bezerra, and A. Pras, “Assessing the Quality of Flow Measurements from OpenFlow Devices,” *8th International Workshop on Traffic Monitoring and Analysis (TMA)*, 2016.

[59] C. Xing, K. Ding, C. Hu, and M. Chen, “Sample and Fetch-Based Large Flow Detection Mechanism in Software Defined Networks,” *IEEE Communications Letters*, vol. 20, pp. 1764–1767, Sep. 2016.

[60] Y. Afek, A. Bremler-Barr, S. Landau Feibish, and L. Schiff, “Detecting heavy flows in the SDN match and action model,” *Computer Networks*, vol. 136, pp. 1–12, May 2018.

[61] A. I. Frunza, C. I. Rincu, and A. Jitaru, “Remote Network Monitoring Using SDN Based Solutions,” in *2018 International Conference on Communications (COMM)*, pp. 301–304, IEEE, Jun. 2018.

[62] P. H. A. Rezende, P. R. S. L. Coelho, L. F. Faina, L. J. Camargos, and R. Pasquini, “Analysis of monitoring and multipath support on top of OpenFlow specification,” *International Journal of Network Management*, vol. 28, p. e2017, May 2018.

[63] T. Yingying Cheng and X. Jia, “Compressive Traffic Monitoring in Hybrid SDN,” *IEEE Journal on Selected Areas in Communications*, vol. 36, pp. 2731–2743, Dec. 2018.

[64] C. Lin, Y. Bi, H. Zhao, Z. Liu, S. Jia, and J. Zhu, “DTE-SDN: A Dynamic Traffic Engineering Engine for Delay-Sensitive Transfer,” *IEEE Internet of Things Journal*, vol. 5, pp. 5240–5253, Dec. 2018.

[65] S.-H. Shen, “An Efficient Network Monitor for SDN Networks,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 46, pp. 95–96, Jan. 2019.

[66] B. Wang and J. Su, “FlexMonitor: A Flexible Monitoring Framework in SDN,” *Symmetry*, vol. 10, p. 713, Dec. 2018.

[67] S. C. Madanapalli, M. Lyu, H. Kumar, H. H. Gharakheili, and V. Sivaraman, “Real-time detection, isolation and monitoring of elephant flows using commodity SDN system,” in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–5, IEEE, Apr. 2018.

[68] R. Cohen and E. Moroshko, “Sampling-on-Demand in SDN,” *IEEE/ACM Transactions on Networking*, vol. 26, pp. 2612–2622, Dec. 2018.

[69] L. Wang, Q. Li, Y. Jiang, X. Jia, and J. Wu, "Woodpecker: Detecting and mitigating link-flooding attacks via SDN," *Computer Networks*, vol. 147, pp. 1–13, Dec. 2018.

[70] P.-W. Tsai, N. Xia, C.-Y. Hsu, S.-W. Lee, C.-S. Yang, and T.-L. Liu, "Design and Implementation of an Adaptive Flow Measurement for SDN-based Cellular Core Networks," in *2018 15th International Symposium on Pervasive Systems, Algorithms and Networks (I-SPAN)*, pp. 179–184, IEEE, oct 2018.

[71] G. Tangari, D. Tuncer, M. Charalambides, Y. Qi, and G. Pavlou, "Self-Adaptive Decentralized Monitoring in Software-Defined Networks," *IEEE Transactions on Network and Service Management*, vol. 15, pp. 1277–1291, Dec. 2018.

[72] "Software-defined network monitoring." `https://www.ca.com/us/info/modern-network-monitoring/software-defined-networking-monitoring.html`.

[73] "Extrahop." `https://www.extrahop.com/solutions/sdn-monitoring/`.

[74] A. Soule, K. Salamatian, A. Nucci, and N. Taft, "Traffic matrix tracking using Kalman filters," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, pp. 24–31, 2005.

[75] K. Xie, L. Wang, X. Wang, G. Xie, J. Wen, and G. Zhang, "Accurate recovery of Internet traffic data: A tensor completion approach," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, Apr. 2016.

[76] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: Towards Programmable Network MEasurement," *IEEE/ACM Transactions on Networking*, vol. 19, pp. 115–128, Feb. 2011.

[77] Y. Guo, Z. Wang, X. Yin, X. Shi, and J. Wu, "Traffic engineering in hybrid SDN networks with multiple traffic matrices," *Computer Networks*, vol. 126, pp. 187–199, oct 2017.

[78] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic Matrix Estimator for OpenFlow Networks," in *Springer*, pp. 201–210, 2010.

[79] Y. Yu, C. Qian, and X. Li, "Distributed and collaborative traffic monitoring in software defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*, (New York, New York, USA), pp. 85–90, ACM Press, 2014.

[80] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[81] M. Polverini, A. Baiocchi, A. Cianfrani, A. Iacovazzi, and M. Listanti, "The Power of SDN to Improve the Estimation of the ISP Traffic Matrix Through the Flow Spread Concept," *IEEE Journal on Selected Areas in Communications*, vol. 34, pp. 1904–1913, Jun. 2016.

[82] M. Polverini, A. Iacovazzi, A. Cianfrani, A. Baiocchi, and M. Listanti, "Traffic matrix estimation enhanced by SDNs nodes in real network topology," in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, vol. 2015-Augus, pp. 300–305, IEEE, Apr. 2015.

[83] Y. Tian, W. Chen, and C.-T. Lea, "An SDN-Based Traffic Matrix Estimation Framework," *IEEE Transactions on Network and Service Management*, vol. 15, pp. 1435–1445, Dec. 2018.

[84] C. Liu, A. Malboubi, and C.-N. Chuah, "OpenMeasure: Adaptive flow measurement & inference with online learning in SDN," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, vol. 2016-Septe, pp. 47–52, IEEE, Apr. 2016.

[85] M. Malboubi, Y. Gong, W. Xiong, C.-N. Chuah, and P. Sharma, "Software Defined Network Inference with Passive&#x002F;Active Evolutionary-Optimal pRobing (SNIPER)," in *2015 24th International Conference on Computer Communication and Networks (ICCCN)*, vol. 2015-Octob, pp. 1–8, IEEE, Aug. 2015.

[86] M. Malboubi, S.-M. Peng, P. Sharma, and C.-N. Chuah, "A learning-based measurement framework for traffic matrix inference in software defined networks," *Computers & Electrical Engineering*, vol. 66, pp. 369–387, Feb. 2018.

[87] Y. Gong, X. Wang, M. Malboubi, S. Wang, S. Xu, and C.-N. Chuah, "Towards accurate online traffic matrix estimation in software-defined networks," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research - SOSR '15*, (New York, New York, USA), pp. 1–7, ACM Press, 2015.

[88] D. Li, C. Xing, N. Dai, F. Dai, and G. Zhang, "Estimating SDN traffic matrix based on online adaptive information gain maximization method," *Peer-to-Peer Networking and Applications*, pp. 1–16, Mar. 2018.

[89] D. Jiang, L. Huo, and Y. Li, "Fine-granularity inference and estimations to network traffic for SDN," *PLOS ONE*, vol. 13, p. e0194302, May 2018.

[90] W. Queiroz, M. A. Capretz, and M. Dantas, “An approach for SDN traffic monitoring based on big data techniques,” *Journal of Network and Computer Applications*, vol. 131, pp. 28–39, Apr. 2019.

[91] ONF, “OpenFlow Switch Specification 1.5.0,” 2014.

[92] “Graphstream a dynamic graph library.” `http://graphstream-project.org`.

[93] OpenStax, *Introductory Statistics*. https://openstax.org/, 2018.

[94] D. Forsyth, *Probability and Statistics for Computer Science*. Springer International Publishing, 2018.

Wander Jácome de Queiroz

EDUCATION

PhD Degree, Software Engineering
Western University, London, Canada

Master of Science, Computer Science
University of Brasilia, Brasilia, Brazil

HONORS & AWARDS

- Award for Outstanding Presentation in Graduate Symposium – Department of Electrical and Computer Engineering, Western University, 2018.
- PhD Scholarship – CNPq (National Council for Scientific and Technological Development) - Science without border program, Brazil, 2015-2019
- MSc Scholarship – CAPES (Coordination for the Improvement of Higher Education Personnel) Brazil, 1997-1999

PUBLICATIONS

Journal papers

- Wander Queiroz, Miriam A.M. Capretz, Mario Dantas. *"An approach for SDN traffic monitoring based on big data techniques"*, Journal of Network and Computer Applications, Volume 131, Pages 28-39, 2019.

RELATED WORK EXPERIENCE

Teaching Assistant **Aug/2015 – Apr/2018**
Department of Electrical and Computer Engineering, Western University, London, Canada

IT Assistant **Aug/2014 – Apr/2015**
Bank of Brazil, Brasilia, Brazil

Lecturer **Feb/2013 – Feb/2015**
UniCEUB, Brasilia, Brazil

Software Engineer **Feb/2009 – Aug/2014**
Social Development and Fight Against Hunger Ministry, Brasilia, Brazil