Window Based Monitoring: Packet Drop Detection in the Network Data Plane

Afonso de Paiva e Pona Corte Real Gonçalves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Fernando M. V. Ramos
Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. Fernando M. V. Ramos
Member of the Committee: Prof. Paulo da Fonseca Pinto

November 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my advisors Professor Luís Rodrigues and Professor Fernando Ramos for welcoming me as their student and for the valuable lessons learned during this year.

I want to thank my family, especially my parents, Filipa and Nuno Gonçalves, my grandmother Avó Pilar, and my godmother Zinha, for their support during this strenuous journey. Their encouragement and caring over all these years was a warm sunbeam that kept me pushing forward.

I also want to thank Margarida Chinopa for her continuous encouragement and for being so understanding and thoughtful during this year. To Hugo Sequeira, for the interesting and productive conversations we had and for inspiring me to be a better version of my self, thank you. To all my other friends, especially Marcelo Santos and Tiago Rijo, thank you very much for being always there during the good and bad moments of my life.

To each and every one of you – Thank you.

Abstract

Detecting network anomalies, such as packet loss, is becoming an increasingly important task to network operators, as applications are becoming more and more performance sensitive. Several solutions aim to detect these events as soon as they occur, as well as to disclose where they are taking place. Unfortunately, some of them incur in unacceptable overhead while others have to sacrifice coverage in order to cope with the increasing traffic intensity. Software Defined Networks and the Programmable Data Plane are relatively recent technologies that allow network operators to configure how switches process packets, which opens the door to efficient monitoring solutions. In this thesis, we develop WBMon, a passive solution that leverages the Data Plane programmability to perform an inter-switch coordination algorithm that detects packet drops in arbitrary paths at line speed. Additionally, we employ a Failure Inference Algorithm (NetBouncer [1]) to enable localizing the links responsible for packet drops. Our evaluation shows that WBMon is able to detect every packet drop in less than 2ms, which allows to detect short-lived failures.

Keywords

Drop Detection; SDN; Programmable Switches.

Resumo

A deteção de anomalias nas redes de computadores, nomeadamente a deteção de perdas de pacotes, tem vindo a tornar-se uma tarefa cada vez mais importante para os operadores de rede, à medida que os utilizadores exigem melhor desepenho por parte dos serviços da internet. Existem várias soulções que pretendem detetar estas anomalias assim que as mesmas aconteçam, bem como revelar onde são originadas. Infelizmente, algumas soluções implicam custos adicionais inaceitáveis, enquanto que outras sacrificam a qualidade da monitorização a fim de se conseguirem adaptar ao aumento da intensidade de tráfego. As Redes Definidas por Software e o Plano de Dados Programável são tecnologias relativamente recentes que permitem que os operadores de redes configurem a forma como os switches processam os pacotes, permitindo o desenvolvimento de soluções de monitorização mais eficientes. Nesta tese concebemos WBMon, uma solução passiva que tira proveito da programabilidade do Plano de Dados para realizar um algoritmo de coordenação de switches que deteta perdas de pacotes em caminhos arbitrários da rede. Para além disso, usamos um algoritmo de inferência de falhas (NetBouncer [1]) de modo a localizar as ligações responsáveis por essas mesmas perdas. A nossa avaliação mostra que WBMon consegue detetar todas as ocorrências desta anomalia em menos de 2 milissegundos, permitindo detetar falhas de curta duração.

Palavras Chave

Deteção de Perdas de Pacotes; Redes Definidas por Software; Switches Programáveis.

Contents

1 Introduction **1**

1.1 Contributions . . . 3

1.2 Results . . . 3

1.3 Structure of the Document . . . 4

2 Background **5**

2.1 Network Monitoring . . . 7

2.2 Software Defined Networks . . . 7

2.2.1 SDN . . . 8

2.2.2 Programmable Data Plane . . . 9

2.3 The architecture of a Programmable Data Plane . . . 9

2.4 Packet Reordering . . . 11

3 Related Work **13**

3.1 Active vs Passive Monitoring . . . 15

3.2 Placement . . . 16

3.2.1 Host-Based . . . 16

3.2.2 Switch Assisted . . . 16

3.2.3 In-Switch . . . 17

3.3 Techniques . . . 17

3.3.1 Probing . . . 18

3.3.2 Mirroring . . . 18

3.3.3 Sampling . . . 18

3.3.4 Filtering . . . 19

3.3.5 Compressing . . . 19

3.3.6 Sketching . . . 19

3.3.7 Coding . . . 20

3.3.8 Selecting . . . 20

3.4 State of the Art . . . 20

3.4.1 PingMesh . . . 21
3.4.2 NetBouncer . . . 22
3.4.3 Planck . . . 23
3.4.4 Everflow . . . 23
3.4.5 OpenSketch . . . 24
3.4.6 UnivMon . . . 25
3.4.7 FlowRadar . . . 26
3.4.8 NetSight . . . 27
3.4.9 NetSeer . . . 28
3.5 Discussion . . . 29

4 Window Based Monitoring **31**
4.1 System Architecture Overview . . . 33
4.2 Underlying Model . . . 35
4.3 Drop Detection Algorithm . . . 36
4.3.1 Handling Edge Cases . . . 39
4.3.2 Exchanging Sequence Numbers . . . 40
4.4 Failure Inference Algorithm . . . 41

5 Evaluation **43**
5.1 Drop Detection Algorithm . . . 45
5.1.1 Reordering Tolerance . . . 45
5.1.2 Drop Burst Tolerance . . . 48
5.1.3 Performance with realistic traces . . . 51
5.1.4 Convergence Time . . . 53
5.2 M-Switch Placement . . . 55
5.2.1 Simulation Setup . . . 55
5.2.2 ISP Topology . . . 56
5.2.3 Data Center Topology . . . 59

6 Conclusion **63**
6.1 Conclusions . . . 65
6.2 Future Work . . . 65

Bibliography **67**

A WBMon Data Plane Source Code **75**

1 Introduction

Contents

1.1 Contributions 3
1.2 Results 3
1.3 Structure of the Document 4

The task of monitoring the operation of computer networks is a key component to ensure the performance of current distributed systems. Network monitoring allows to gather information that can be used for planning the network evolution, to verify that the network operation complies with target service level agreements, and to detect anomalies, such as faults and intrusions.

In our work we are interested in the use of network monitoring for anomaly detection and, in particular, to detect links that experience excessive packet loss rates. We survey the main techniques that can be used to detect network anomalies, giving emphasis to techniques that leverage the availability of programmable switches to increase the accuracy and efficiency of this task.

From previous work we have identified two main strategies to detect faulty links. The first strategy involves the active exchange of probe traffic among different "observation-points" placed in strategic locations in the network; the data collected by these observation points can be correlated to give hints on the location of eventual faulty links and/or switches. The second strategy uses programmable network switches to detect faulty links in a passive manner, without the need to inject probe traffic; unfortunately, it only works for links that connect directly two programmable switches.

Based on these observations, we propose a new strategy that combines and extends the two techniques above. First, we aim at using programmable switches to detect packet loss in the path connecting these switches, even if the path includes multiple links and non-programmable switches, such that a few programmable switches can be used as *passive* observation points. Then, we plan to correlate the information collected by these switches to *narrow* the set of potential faulty physical links. We believe that the proposed approach has the advantage of avoiding probe traffic, of providing a faster detection of anomalous behaviour, and of being easier to deploy in already functioning networks.

1.1 Contributions

This thesis analyses and implements techniques to detect and locate packet drops in computer networks. The main contribution of this thesis is the following:

- We propose an architecture, named WBMon, that allows to detect packet drops that occur between pairs of programmable switches. It uses the collected data to provide an indication of the subset of links and switches that can be the root cause for the observed anomaly.

1.2 Results

This thesis produced the following results:

- An implementation of the Drop Detection Algorithm for the Intel Barefoot Tofino switch, using the P4 programming language;

- An experimental evaluation of the Drop Detection Algorithm implementation, regarding its accuracy, limitations, and performance under a realistic scenario;
- A theoretical evaluation that discusses the cost and limitations of deploying the proposed solution in different network topologies.

1.3 Structure of the Document

The rest of the report is organized as follows: In Chapter 2 we present background related with our work; Chapter 3 presents and analyses the state-of-the-art solutions in this topic; Chapter 4 describes the developed solution; Chapter 5 presents how we evaluated our solution; and Chapter 6 concludes the report and discusses future work.

2
Background

Contents

2.1 Network Monitoring 7

2.2 Software Defined Networks 7

2.3 The architecture of a Programmable Data Plane 9

2.4 Packet Reordering 11

This chapter introduces the fundamental concepts to this work. Section 2.1 defines and motivates network monitoring; Section 2.2 introduces Software Defined Networks (SDN) and the Programmable Data Plane, and indicates the main differences between legacy and SDN networks; Section 2.3 describes the architecture of the programmable switches used in this work. Finally, Section 2.4 formalizes the reordering concepts that were used during this work.

2.1 Network Monitoring

Our society is highly dependent of networked computer systems. Online banking, online commerce, e-mail, messaging, social networking, virtual meetings, media streaming and on-line gaming are just a few examples of the myriad of daily activities that are dependent on the correct operation of computer networks.

Network monitoring is the task of continuously extracting information regarding the operation of a computer network, in order to better understand how it is being used and to detect potential anomalies, faults, attacks, or other impairments to its correct operation. Network monitoring is required to accomplish many high-level tasks such as: capturing usage patterns and changes in those patterns, understanding how the load is distributed in the network, understanding which flows consume more resources, detecting faulty components, verifying if the routing of packets complies with established routing policies, detect intrusions and/or denial of service attacks, among many others.

The need for network monitoring, as part of the broader task of network management, has been recognized from the inception of the Internet, and early protocols, such as SNMP [2], already provided support for this task. However, the scale of computer networks, as well as the amount of traffic they support, has grown immensely. For instance, networks with thousands of switches and hundreds of thousands of links, transporting Pbps of traffic, are common today [1, 3, 4]. The scale of current networks, combined with the heterogeneity of equipment and protocols that can be used, make the task of performing network monitoring extremely challenging. Fortunately, some technological advances in the networking architecture and hardware, including Software Defined Networks [5] and programmable switches [6–8], can now be leveraged to make network monitoring more accurate and efficient.

2.2 Software Defined Networks

Network routers and switches can perform multiple tasks. One of the main tasks is packet forwarding, which consists in receiving a packet from an ingress link and forwarding it to the next hop towards the destination via an egress link. To perform this task, the switch needs to maintain a *forwarding table*, that specifies which egress link should be selected when forwarding a packet. The other main task, according to the original Internet design, is to execute the logic required to populate the forwarding table, typically a distributed routing protocol, such as RIP [9], OSPF [10], or BGP [11]. The former task is designed to execute in the *Data Plane*, and the latter in the *Control Plane*.

There are several advantages of running the Control Plane in every router, being one of the most important the autonomy and decentralization it provides: routers that can execute the Control Plane can coordinate with each other to populate their forwarding table without being dependent of other additional components. However, this choice also comes with some disadvantages. First, distributed routing protocols are notoriously complex and difficult to debug. Second, routing equipment was typically provided by vendors with proprietary implementation of a fixed set of routing protocols that could not be easily adapted or expanded.

2.2.1 SDN

Software Defined Networking is an architectural model that decouples the Data Plane from the Control Plane. In this model, switches only implement packet forwarding and export an interface that allows an external component to populate the forwarding table. The Control Plane is executed in this logically centralized entity, the *controller*, that decides how to populate the forwarding table of every switch.

Having a single point of control has some issues. For instance, it can become a bottleneck and, without appropriate fault-tolerant measures, it can be a single point of failure. This challenge is handled today with production-level distributed controllers [12]. However, it also brings several relevant advantages. Namely, it makes the control logic simpler to program and easier to verify, and facilitates network configuration. Moreover, it allows to have a global view of the network, and consequently to compute optimized solutions for the entire network, which was not possible to do with a distributed Control Plane [5].

OpenFlow [13] is a standard that specifies the interface between the controller and the switches and that allows the controller to remotely update the forwarding tables. According to this standard, each switch maintains a Flow Table that keeps a list of Match-Action rules, each consisting of a matching and an action part. The matching rule corresponds to a set of conditions that must be met to activate the action part to that packet. It is possible to match packets based on many header fields, from different OSI layers, namely the TCP/UDP source and destination IPs and ports, ARP and ICMP parameters, the switch ingress port where that packet came from, and so on. The action, in turn, defines what should be done to the matched packet. Typical actions are dropping a packet or forwarding it to one or multiple egress ports, but can also include changing or pushing header fields. For each packet, the switch finds the first matching rule and applies the corresponding action, or the default action if no rule matched the packet.

2.2.2 Programmable Data Plane

Processing packets according to OpenFlow rules requires switches to be able to extract the information required to match those rules. For this reason, before performing the Match-Action phase, switches execute a Parsing stage to extract that information.

Traditionally, both the parsing and match-action stages were static and ingrained in the switch, which limited the number of different headers it could recognize and parse, and the type of matches and actions it could support. These *fixed-function switches* [14] impaired the development of SDN in two ways [5]: First, the heterogeneity between switches from different manufacturers forced the Control Plane to be aware of the Data Plane implementation, which hindered the Control and Data Plane disaggregation. Second, the fixed-function logic prevented switches from processing packets according to new or custom protocols, and the only way to introduce new packet processing was to design a new switch chip.

The emergence of Reconfigurable Match-Action Tables (RMT) [6] allowed switches to parse arbitrary headers and define match-action rules programmatically. This further led to the development of architectures and languages that were able to leverage this capability. The P4 programming language [7] can now be used to specify exactly how to parse packet headers and the Match-Action rules to be applied in the forwarding pipeline. Moreover, the P4 language creates an abstraction that completely separates the Control Plane from the Data Plane, as it can be compiled into numerous targets, such as ASIC switches, Field-Programmable Gate Arrays (FPGA), etc. [7]. The language includes the P4 Runtime API [15], a gRPC-based mechanism that allows a remote controller to update the tables of any P4-programmable target.

Switch programmability allowed to redesign multiple network solutions, as it granted network insights that proved to be extremely useful in tasks such as network debugging and monitoring.

2.3 The architecture of a Programmable Data Plane

The performance requirements demanded in current networks force programmable switch vendors to design devices with high throughput. Consequently, state of the art switch designs end up having a rigid architecture with constraints that are not commonly seen in other processing units. This section presents a brief discussion of the Intel Tofino switch architecture, the reference for a modern Programmable Data Plane, and how it affects the implementation of network functions and monitoring solutions.

Figure 2.1 (taken from [16]) presents a diagram depicting the switch processing pipeline. The packet processing starts at the Ingress Parser. In this section, the switch extracts the packet headers into a structure called the Packet Header Vector (PHV), which accompanies the respective packet during its processing. The PHV may also include additional fields that allow to transfer information across the processing pipeline, so called metadata. After the Ingress Parser, the PHV enters the Ingress Control. In

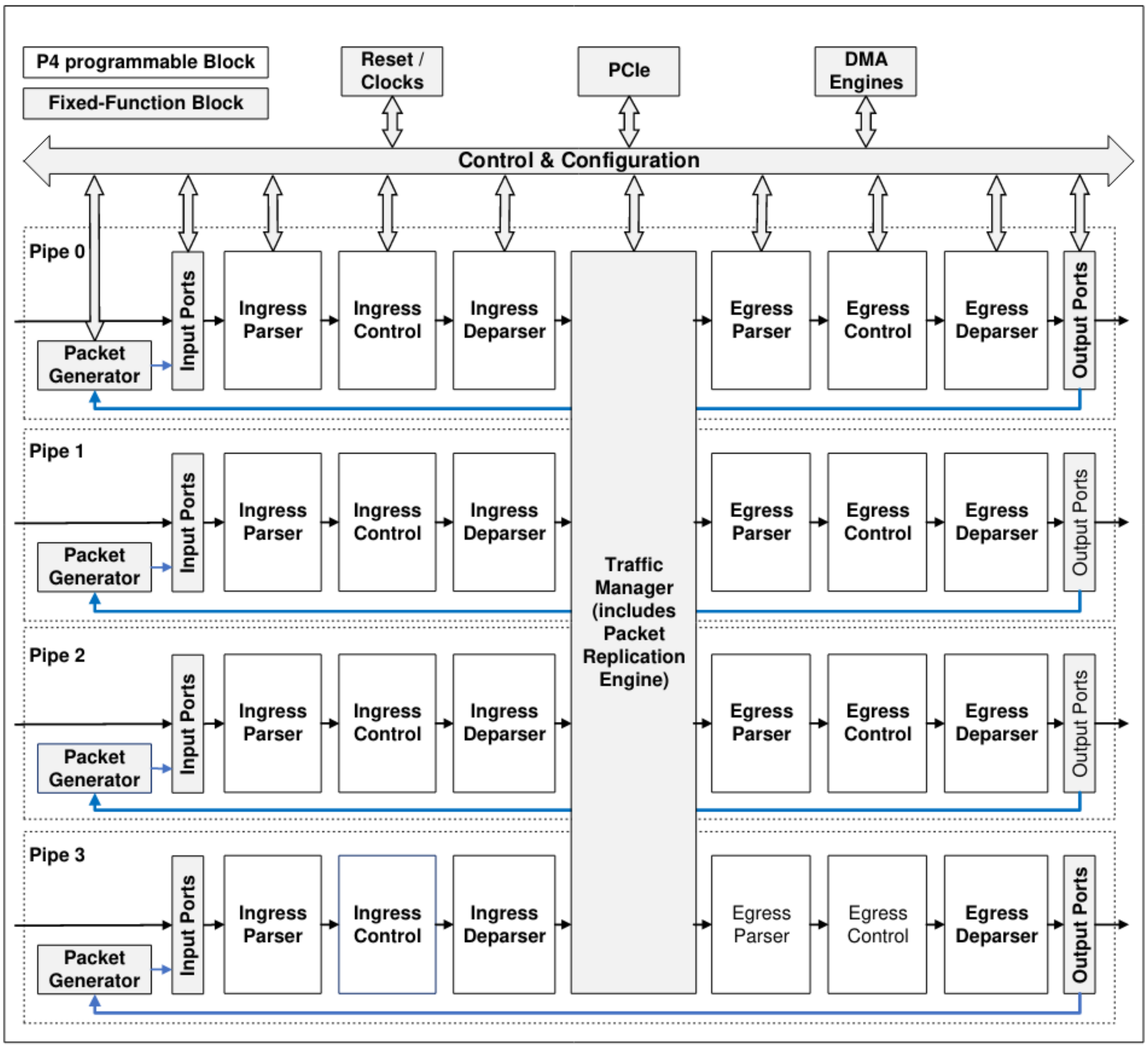


Figure 2.1: Intel Tofino switch architecture [16]

this section, the switch matches the PHV contents to a Match Table and executes an action (e.g., drop, forward, or clone), or other additional computations. Then, the packet goes to the Ingress Deparser, where its headers are reassembled from the PHV, and then it is sent to the Traffic Manager. The latter is responsible for enqueuing the received packets and executing the forwarding instructions given by the Ingress. Finally, the packet is sent to the Egress, where its final processing takes place. The Egress has a similar structure to the Ingress.

The Ingress and Egress Control blocks are split into different stages, organized into a pipeline. Each stage has the resources required to apply a single Match-Action Table and execute the respective action. Nonetheless, the operations allowed in each action are limited: For instance, each action can perform a maximum of two comparisons and access, at most, 12 bits of the PHV.

The data in the PHV is local to each packet and expires after it finishes its processing, making the PHV unsuitable to store data that needs to persist across packets, such as byte counters or state buffers. Fortunately, the Control block stages are equipped with hardware components, such as Registers, that allow to solve this problem. However, each stage can only have a single Register array and each can only be accessed in the stage it belongs to. One can, however, use the PHV to carry Register information to subsequent stages, or to carry the value to be written to an upcoming Register. In sum, a Register value may only be used after the stage that Register resides in, and the value to write on a Register must be computed until the packet reaches that Register's stage, which poses serious challenges when programming the Data Plane.

One example of that limitation is when trying to swap the values of two registers A and B: Each register must reside in its own stage. Since stages are executed sequentially, one of them is necessarily executed first. Let's suppose that the first stage to be executed contains Register A and the other contains Register B. It is trivial to write the value of A in B: One would have to read Register A, store its value in the metadata and then, when the processing reaches the stage of Register B, it would write the value stored in the PHV into Register B. However, when it comes to write the value of B on A, the task is not that easy since it requires writing a value that can only be read after the last stage where it can be written. Tofino switches support a mechanism to reprocess packets called *resubmit*. Packets can only be resubmitted in the Ingress and, when they do, they reenter the Ingress processing pipeline. This mechanism allows, for instance, to write the value of B in A. Unfortunately, resubmissions affect throughput, as resubmitted packets compete with other arriving packets for packet processing.

Another task that is affected by this limitation is updating the value of one Register based on its value and some other metadata. Since the value can only be read and written in the stage the Register resides in, all the computation should be performed during the limited time (a few ns) the packet stays in that stage. The limited computational resources available in the Tofino switch thus severely reduce the variety of operations of this nature that can be performed.

2.4 Packet Reordering

Whenever two machines are communicating over a network, it is not guaranteed that the packets arrive in the same order as they were sent. In this section, we formalize the reordering concepts used along this document.

Let's start by assuming that the sending machine numbers packets in the order they are sent, and that the receiver records the highest number it has received. For simplicity, we call *sequence number*, or *sn* to the number representing the packet ordering, and *max_sn* to the highest *sn* stored by the receiver. Note that these sequence numbers are different from the ones used in TCP. While TCP sequence numbers refer to the *byte* ordering, the sequence numbers used in this work are relative to the *packet* ordering, regardless of the size of each packet. For instance, after sending a packet with sn $= x$, we consider that the next packet will always have sn $= x + 1$.

In this work we define reordered packets in accordance to [17]. To detect reordered packets, the receiver can compare the sequence number of the incoming packet with the value of *max_sn*. We say a packet is reordered if, at the time of its arrival, its sequence number is smaller than *max_sn* [17]. In other words, reordered packets are packets that arrive after any of its successors. For this reason, we call reordered packets *late* packets. For instance, if the downstream receives the packet sequence $\{1, 2, 4, 5, 3\}$, we only consider packet 3 to be reordered. In the context of the present work, it is also important to consider packets that arrive sooner than expected. For simplicity, we will refer to those packets as *early*, or *premature* packets. We say a packet is *early* if, at the time of its arrival, its sequence number is greater than *max_sn* $+ 1$. For instance, in the example given before, packet 4 is premature.

We can also calculate the *displacement* of any packet at the time of its arrival, by computing the difference between the expected and received sequence numbers. More precisely, *displacement* $=$ $($*max_sn* $+ 1) -$ *received_sn*. With this definition, late packets will have a positive displacement, and premature packets will have a negative displacement. For instance, in the above packet sequence, packets 1, 2 and 5 will have a displacement equal to zero, packet 4 will have a displacement of $(2+1) - 4 = -1$, and packet 3 will have a displacement equal to $(5 + 1) - 3 = 3$.

Summary

In this chapter we introduced fundamental background that will be necessary in future sections. Namely, we motivated the network monitoring task and provided essential insights on SDN and on the Programmable Data Plane. We also defined the Reordering concepts and terms that will be used in the rest of the document.

3
Related Work

Contents

3.1 Active vs Passive Monitoring 15
3.2 Placement 16
3.3 Techniques 17
3.4 State of the Art 20
3.5 Discussion 29

In this chapter we make an overview of the network monitoring techniques and systems that are most relevant to our work. We start by characterising the different approaches to network monitoring, then we enumerate the most common techniques that are used in the implementation of monitoring systems, and finally we describe, with some detail, a number of monitoring systems that can be used to detect network anomalies.

3.1 Active vs Passive Monitoring

Monitoring strategies can be classified as *active* or *passive*. Both approaches have advantages and disadvantages.

Active monitoring relies on exchanging packets whose sole purpose is to perform monitoring tasks. These extra packets are called *probing* or *monitoring traffic*, to distinguish it from the *application traffic* that exists in the network. Active monitoring sends probing messages to the network and extracts information from the behaviour of these probes. For example, an active approach can send ping messages to a given node to measure the round-trip time (RTT) of that path.

Passive monitoring, on the other hand, avoids sending additional traffic on the network, and extracts the desired information from the data collected during the forwarding of application traffic. A passive approach can calculate network latency by observing the time each application packet spends at each switch, or detect faulty links by observing how many packets are lost in each link.

One advantage of active monitoring is that it makes monitoring more independent from application traffic. For instance, it allows to measure the latency of a link when no other traffic occurs. Also, active monitoring allows to artificially create and test scenarios, such as specific sequence of packets, that can occur only sporadically but that need to be addressed [4]. However, active monitoring has also some important disadvantages:

First, one can claim that active monitoring is unsound by design, as it only detects anomalies that directly affect monitoring traffic (e.g. does not detect black-hole drops that affect application traffic), allowing false negatives to take place [18]. In the limit, a faulty network may be reported as healthy if its anomalies only affect application traffic, rendering this solution ineffective. Moreover, active monitoring cannot detect transient anomalies that take place between probing epochs. Passive approaches, on the other hand, do not face this problem, since they directly monitor the application traffic.

Second, active monitoring can be less efficient, as it requires additional traffic to be generated and, often, a substantial part of this traffic does not contribute to detect any anomaly [3]. From this perspective, active monitoring can be more effective as a complementary diagnosing tool, after an anomaly is detected by some other mechanism [4].

Finally, the latency of anomaly detection is a function of the probing frequency, and anomalies that are recurrent but of short duration may pass unnoticed, as they are unlikely to occur when probing takes effect. This can be mitigated by increasing the probing frequency, but this may generate more monitoring traffic in the network, which may cause more congestion and further degrade its performance [19]. With passive monitoring, an anomaly that affects the application traffic can potentially be detected faster.

In our work, we will give preference to passive techniques, to avoid the costs of exchanging probing traffic.

3.2 Placement

One can classify different solutions based on where the information is collected and processed, each location having its advantages and limitations. Our survey allowed the categorization of placement into three types, *Host-Based*, *Switch Assisted* and *In-Switch*, that will be discussed below.

3.2.1 Host-Based

We denote host-based monitoring as an approach that runs on end-hosts without any support from other network components. Host-based solutions can be passive, if they use solely the traffic that is being generated by the application, or active, if they resort to sending probing messages, such as ping messages or dedicated data packets. These solutions are very general and consequently easier to deploy in already functioning networks, as they do not require any network modification. However, these approaches are unable to get network insights that are crucial to detect and locate certain anomalies, such as packet traces or the traffic intensity distribution.

3.2.2 Switch Assisted

We say a solution is Switch Assisted when it employs switches to collect the network statistics used to monitor the network, but requires an external component (or set of components) to process them.

Although it was already possible to perform Switch Assisted monitoring with tools like SNMP [2], the emergence of SDN made it more powerful. It allowed to install forwarding rules that would give more network insights not possible to attain before. For example, it enabled the collection of packet traces which would reveal the last switch that processed a certain packet or disclose the presence of routing loops [18, 20]; to assemble network statistics that would unveil devices overloaded with traffic [21]; or even to collect inter-packet time statistics to identify malicious activity in the network [22]. These insights not only allow for higher coverage, as there is more information available to the monitoring task, but some may also help locating the causes of network anomalies.

Despite these advantages, this approach still poses a challenge that must be addressed: As only a small portion of traffic suffers network anomalies [4], most of the monitoring traffic generated by these solutions will not be useful. This poses a serious efficiency problem given that the monitoring traffic may congest the network, cause more anomalies, and/or require significant computational resources to process it [4]. Although there are several techniques that aim to alleviate this limitation (Section 3.3), it is not possible to completely mitigate it because it lies on the design of Switch Assisted solutions: As the anomaly detection is done exclusively in the Control Plane, switches are unable to select only the necessary information and consequently will always produce unnecessary monitoring traffic.

3.2.3 In-Switch

We denote in-switch monitoring as an approach that performs the anomaly detection inside the switch. Note that although these solutions may store the collected information in external devices for further analysis, they do not depend on any external device to monitor the network.

The advent of the PISA architecture [7] and the emergence of programmable switches enabled this approach, that has several advantages. First, it avoids the efficiency limitation discussed in the previous approach: as programmable switches allow to migrate the entire anomaly detection logic to the Data Plane [3, 23–25], the monitoring traffic can consist exclusively of anomaly related information. Second, In-Switch solutions are, in general, more scalable since the detection logic is distributed across the network.

Nonetheless, this approach is not exempt from limitations: First, to maintain high throughput levels, modern programmable switches have limited computational and memory resources available. For instance, it is not possible to execute multiplications or cycles in current programmable switches. This limitation constrains the logic that can be executed in the switches. Moreover, fine-grained monitoring becomes extremely challenging to implement entirely inside the switch, as it requires large amounts of memory to store all the required counters. Second, resource limitations prevent switches from storing large amounts of anomaly information and external components must be used to perform this task. These components may become a bottleneck as the number of detected anomalies increases, and the traffic generated to send this information may be unacceptably large in some situations. Hence there is a continuous effort to reduce the size of the required monitoring traffic.

3.3 Techniques

In this section, we enumerate some of the main techniques used to monitor the network and discuss the strengths and limitations of each. These are *Probing*, *Mirroring*, *Sampling*, *Filtering*, *Compressing*, *Sketching*, *Coding* and *Selecting*.

3.3.1 Probing

This approach injects monitor traffic in the network to infer its state based on what happens to that traffic. For example, a solution may detect packet drops, forwarding loops or black-holes if the injected traffic does not reach its destination. This approach has the benefit of allowing to test specific conditions that may not occur often in the network. However, as the traffic generated in this approach is artificial, it may not reflect the behaviour of the network with real application traffic.

3.3.2 Mirroring

This technique consists of making switches copy certain packets and sending those copies to an observation point in the network. These copies are often processed before being sent, to include only the necessary information, such as the switch ID or *in/egress* ports [26]. It allows end-hosts to collect information that may be crucial to detect anomalies, but incurs in the risk of generating too much traffic that may disrupt network performance.

3.3.3 Sampling

Sampling occurs when a solution only considers a random subset of the application traffic, and is often used to reduce the processing and monitoring traffic overhead. We have identified two major approaches to sampling:

In the first approach, packets are fully randomly sampled. This can either be done by picking every *ith* packet, or by picking every packet until it reaches the sampling capacity [26]. As the order in which packets pass through switches is unpredictable, this approach successfully covers a wide variety of flows and packets. Nevertheless, this method cannot ensure that different devices will sample the same packets, which may be required for some tasks [27].

In the second approach, different devices may sample the same random subset of packets, typically by relying on hash-functions to select which packets to take [4]. This technique can be used to sample random packets, by hashing the packet identifier, or to sample all the packets that belong to the same flow, by hashing the flow identifier, for example. As long as every device uses the same hash function, it is certain that if a packet is sampled in one device, then it will be sampled in every device that processes it, which may be useful in some scenarios.

Note that the assumption that the collected sample is a good representation of the network traffic may not hold in every situation. For example, the presence of heavy hitters in the network may bias the measured statistic. Moreover, packets that are not sampled may contain crucial information to some monitoring tasks, such as identifying flow size distribution or detecting black holes. For this reason, sampling may not be appropriate to some applications [18, 28, 29].

3.3.4 Filtering

Another way to reduce the number of processed packets is by filtering only the packets that satisfy certain rules, specified by the network operators. This technique differs from sampling in the way that the former targets specific traffic, while the latter targets a random subset of it. This approach may lead to more accurate monitoring results as it grants a finer control on the monitored traffic. For example, it allows to collect only the packets that are originated from or targeted to a certain set of IPs, or packets that follow a specific protocol, such as TCP or ARP [4]. Nevertheless, contrarily to sampling, this approach cannot estimate the amount of traffic that will be monitored, as it depends on the traffic that is circulating in the network [18]. For instance, it is possible that either every single packet or no packet at all matches an established filter.

3.3.5 Compressing

Unlike the previous techniques, compression aims to curtail the monitoring traffic without discarding any information, by reducing the size that information takes. There are several ways of using compression. For instance, some approaches compute the *diffs* of consecutive packets (diff encoding) [20] while others employ off the shelf compression algorithms, such as LZMA, gzip or rar [30]. This technique allows to collect more network information and to consequently achieve more accurate results at the cost of consuming more computational resources.

3.3.6 Sketching

Sketches are space-efficient probabilistic data structures used to compute accurate network statistic estimates with low memory requirements and provable resource-accuracy tradeoffs [21, 31–33]. The computation required to operate these data structures is simple enough to be computed inside the switch. Indeed, several solutions today use this approach to fulfil numerous tasks, including frequency estimation [34], heavy hitter detection [23], distinct flow counting, [35], change detection [36], entropy estimation [37], and attack detection [22].

The use of sketches has two main shortcomings: First, sketches demand higher computing resources, which limits the amount of sketches that can be calculated in each switch. Second, these structures typically store "heavy" traffic, often losing the "mice" flows. The coarse-grained statistics thus obtained may lose information crucial to specific fine-grained monitoring tasks, such as anomaly location or per-flow monitoring [3, 18].

There is an active effort to overcome these limitations. To deal with the limited number of sketches that can be computed in each switch, some solutions allow to dynamically change and configure the sketches computed at each switch [21]. Others employ *universal streaming* primitives, from which it is possible to calculate several metrics [32, 38, 39]. It is also possible to calculate fine-grained sketches by filtering packets into different sketches. However, this solution incurs in a tradeoff between granularity and memory cost.

3.3.7 Coding

This technique works by *encoding* the information to be transmitted into a different representation, with the goal of either reducing the size of the monitoring traffic or, on the other hand, improving transmission robustness by adding coding redundancy. The coded representation can then be *decoded* to retrieve the original information. Contrary to techniques based on sketches, coding is based on deterministic algorithms. For instance, FlowRadar [18] (further analysed in Section 3.4) is a monitoring solution that employs this technique by encoding (with a bit-wise XOR) multiple packet counters in colliding table entries. These entries are then decoded to retrieve the original packet counters. A solution that uses sketches would either need to store a counter for each flow, which would use too much memory, or to employ stochastic data structures, such as Count Min sketches [40] or Bloom sketches [41], to hold that information. This benefit comes at the cost of demanding additional computational power to perform the encoding and decoding operations. These operations are hard to fit into devices with limited resources, such as switches, although recent work gives hope that the challenge is not insurmountable [42].

3.3.8 Selecting

This technique is able to reduce the monitoring traffic by discarding unnecessary information. It consists of picking only the information that represents monitoring targets [3]. For example, NetSeer [3] is a solution that employs this technique to detect packet anomalies, such as drops or high latency. Instead of storing packet counters or measuring the time every packet takes at each switch, it only reports (or *selects*) the dropped packets or the ones that experience latency higher than a threshold.

3.4 State of the Art

In this section we analyze monitoring systems that illustrate different techniques that can be used to detect packet loss. The goal of this analysis is twofold. On the one hand, to understand potential limitations of existing solutions. On the other, to get a more in-depth and practical view of techniques that may help us in achieving our goals.

3.4.1 PingMesh

PingMesh [43] is a Host-Based Active solution that uses Probing to measure the latency between hosts in a geo-distributed data-center network. It makes end-hosts *ping* other nodes to collect statistics and has three main components: the Controller, the Agent and the Data Storage and Analysis (DSA).

The Controller generates a *pinglist* file for each Agent. These files contain the set of peers each Agent will ping, as well as additional parameters to configure the number and size of each probe. It aims to find a balance between network coverage and the amount of traffic the *pinglists* will generate: On the one hand, the set of pinglists must cover a wide range of paths in the network to present accurate results. On the other hand, having too many pings may cause an unacceptable traffic overhead that may damage network performance. The best compromise was found to be the following: By leveraging the Clos topology [44] of the target network, the authors were able to cluster different hosts according to the Pod they belong to. Every host would ping every other host belonging to the same Pod. To test inter-Pod connectivity, the Controller would make every host of each Pod probe a single host of every other Pod. Finally, to test inter-data-center connectivity, each data center would select some of its hosts and each of them would ping a single host of every other data-center. This scheme allows to test virtually every connection in the entire network while minimizing the number of redundant probes.

Each server in the data-center has a PingMesh Agent instance running in it and will periodically retrieve the most recent pinglist file from the controller and ping the other Agents listed in that file. These probes use TCP/HTTP traffic to be as similar to application traffic as possible. After collecting the probing results, each Agent calculates the desired performance metrics (latency and packet drop rate) and then uploads them to the DSA for storage and further analysis.

The DSA is able to detect packet drops and black-holes from the latency data. As the TCP timeout value is known for the target data-center and is significantly higher than the average RTT, it is possible to infer the number of retransmissions done by TCP, and consequently the percentage of dropped packets, from the latency values. This information can further be used to deduce the presence of packet black-holes: if several servers connected to the same ToR experience higher packet drops rates than usual, then it is possible that that ToR switch is causing black-hole packet drops. The same reasoning can be done for higher levels in the network topology. If several ToR switches experience higher packet drops, maybe the drops are caused by the Leaf or Spine layer.

A crucial feature of PingMesh is that it allows to monitor the connectivity and latency between virtually every host pair while generating relatively few probing messages. Moreover, as it is Host-Based, it can be deployed without requiring any modification to the targeted network.

Despite being able to identify connectivity problems, this solution cannot locate the devices that may be hindering that connectivity because it only has data collected in the edge of the network. Additionally, by following an Active approach, this solution generates unacceptable amounts of monitoring traffic for modern data-center networks (at least 4×10^6 probes per epoch [43]).

3.4.2 NetBouncer

NetBouncer [1] is another Host-Based, Active monitoring solution that uses Probing to locate the links and switches that cause packet drops in the network. It consists of three main components: the Hosts, the Controller and the Processor.

Hosts probe the network by sending IP-in-IP [45] "*bouncing packets*" to specific switches that lie in it. Each host creates an IP packet addressed to the intended switch and inserts another IP packet addressed to itself in the payload of the first packet. When that packet reaches the intended switch, it unwraps the inner IP packet and sends it back to the original host. In this way, hosts are able to obtain a count of the number of both sent and received packets for each switch, statistics that are then sent to the Processor. This behaviour allows hosts to act independently, as eventual failures will not affect the measurements of other servers, further improving the accuracy of this solution.

The Controller is responsible for generating a probing plan that specifies which switches each Host should probe and to keep them updated with this plan. The probing plan should be *link identifiable*, meaning that it should generate enough data to determine the status of every link. The authors prove that in a layered network where every switch is traversed by, at least, one path that does not drop any packet, a probing plan where every host probes all the paths to top-layer switches is link identifiable. As the Controller knows the network topology at any instant, it is trivial to generate a link identifiable probing plan.

The Processor is assigned to infer the faulty devices based on the data collected from the Hosts. Faulty switches are identified as the ones that have no healthy path traversing it (a healthy path is one that did not drop any packet during a certain epoch).

To create a link failure location mechanism, the authors modeled the network as a graph and assigned a drop probability to every link. These probabilities are assumed independent and, for this reason, the drop probability of a path can be defined as the product of the probabilities of its links. This result can be used to create an equation system that correlates the measured packet drops in each probed path with the drop probabilities of each link. As Host measurements may contain noise, the authors converted this equation system into an optimization problem and added a regularization term to approximate the measured probabilities from 0 or 1. Armed with this mechanism, the Processor is able to estimate the drop rate of each individual link in the network and to identify the faulty ones as those that have a probability higher than a certain threshold.

The main feature of this solution is its ability to locate faulty links and switches inside the network based exclusively on the drop rates measured by the Hosts. Moreover, despite being a Host-Based solution, NetBouncer is able to probe arbitrary paths in the network, starting at any end-host.

Nevertheless, as this solution only monitors probing traffic, the results of its measurements may fail to identify links or devices that drop specific application packets. In addition, the additional probing traffic may congest the network and induce drops in healthy regions of the network.

3.4.3 Planck

In contrast to the previous solutions, Planck [26] is a Passive Switch Assisted monitoring system that employs Mirroring and Sampling to calculate the real-time throughput and congestion in every link in the network. To achieve this goal, it makes every switch mirror every packet and send them to a *Collector*, which computes the intended metrics from the gathered data. These results are then stored for future application queries.

The sampling occurs naturally, as switches oversubscribe the mirroring port. When the mirrored traffic intensity exceeds the port capacity (note that usually there is a single port to mirror the traffic of every other port), excess packets start accumulating in the switch queue and are dropped once that queue is full. One can say that this mechanism allows Planck to dynamically sample traffic, according to its intensity.

This unpredictable sampling rate creates a new challenge when computing the throughput of each flow, as it is not possible to use the packet sizes to calculate the number of sent bytes anymore. Instead, Planck uses header fields (e.g. *SYN* value for *TCP*) of two different packets from the same flow to infer that value. The throughput of each flow is then calculated by dividing the number of bytes by the elapsed time between the reception of those packets. The Controller is able to calculate the throughput of each link by summing the throughput of each flow that is sent to each link. This latter information can easily be obtained from the network topology and routing tables of each switch.

Planck grants a higher mirroring rate than other solutions [27] since it mirrors packets directly in the Data Plane. Doing it using the switch CPU significantly reduces the mirroring throughput [26]. However, oversubscribing mirroring ports will fill congestion buffers with mirrored traffic, which may increase the number of application packet drops and may reduce the accuracy of the calculated throughput at the Collector, as the time delta between packets may be altered.

3.4.4 Everflow

Everflow [4] is another Passive, Switch Assisted monitoring solution that employs Mirroring, Sampling, Filtering and Probing to detect network anomalies, such as routing loops and packet drops. This solution uses network switches to collect packet traces from the entire network by mirroring certain packets to external Analysers. After receiving a complete packet trace, Analysers process it to detect anomalies that may have occurred and store the results in a common storage device. The network Controller can then query that storage to retrieve the anomaly information and further answer application queries.

To reduce the number of mirrored packets and to assure that traced packets are traced at every switch, packet sampling is based on the hash value of their *packet identifier*. Moreover, this solution determines the Analyser mirrored packets are sent to according to the hash value of their *flow identifier*, to arrange the traces of the same flow into the same Analyser. Additionally, it also mirrors packets that contain a certain debug bit in the header set to 1, as it allows to force certain packets to be traced. These mirroring rules alone, however, may disregard smaller flows, as the smaller number of packets makes them less likely to be sampled. For this reason, this solution also mirrors every packet that establishes or terminates a TCP connection (Filtering).

After receiving an entire trace, Analysers can process the buffered information to detect and locate network anomalies. As a practical example, if a switch appears more than once in a packet trace, then that packet suffered a routing loop. Additionally, packet drops are detected when the last switch in a packet trace does not correspond to the expected last switch for that flow, which is given by the network topology and routing policies. Note that this technique may generate False Positives if the network drops the mirrored packet sent by the final switch.

Finally, Everflow is able to actively inject guided probes in the network to further investigate certain anomalies. For instance, this mechanism allows to determine if a detected packet drop is an intermittent or persistent fault. To this end, Everflow crafts special packets that will follow a certain path in the network. That path is established by using IP-in-IP [45] and the debug bit in the packet header is set to 1 to assure it will be traced at every switch. Despite this Active characteristic, we still consider this solution Passive since the guided probes are used solely to diagnose already detected anomalies.

There are two key ideas we can take from this solution: First, by employing active probes in a Passive approach, it is possible to test arbitrary scenarios that could be impossible to have in an entirely passive solution, while avoiding the unbearable traffic overhead typical of Active solutions. Second, the ability to collect complete packet traces allows to detect packet drops or routing loops, which could not be detected otherwise. Nonetheless, the techniques used to reduce the monitoring traffic overhead end up disregarding application traffic that is still susceptible to suffer network anomalies. This detail seriously tarnishes the coverage of this solution [3]. From this solution, we can also observe that when collecting packet traces, it is crucial to find a compromise between the monitoring coverage and the consequent traffic overhead.

3.4.5 OpenSketch

OpenSketch [21] is a Passive, Switch Assisted solution that uses Sketching, Sampling and Filtering to perform fine-grained analysis with lower traffic overhead. To this end, switches compute sketches that represent the targeted network statistics and regularly send the collected data to the Controller, which is responsible for analysing it to detect network anomalies.

Switches are equipped with generic and efficient sketches and let the Controller determine the ones to be computed. This design allows to dynamically change the metrics that are being collected at switches, and to implement new analysis algorithms on the Controller, without requiring to reprogram the Data Plane. Furthermore, the Controller is able to automatically configure the precision of each sketch based on network operators' needs and on the available resources at the switch. This characteristic grants a great measurement flexibility that is not present in other solutions. Nonetheless, the switch scarce computational resources limit the number of metrics that can be computed simultaneously [32].

OpenSketch performs a finer-grained analysis by accounting packets in different sketches based on user defined filters. These filters become more expressive with the usage of hashing. For example, filtering packets that match a certain Bloom Filter or randomly sampling packets based on their hash prefix become possible with the usage of hash functions. More practically, it allows to separately count the number of packets destined to a specific set of IPs, enabling the detection of DDoS attacks. Unfortunately, this fine-granularity is constrained by the available memory on each switch, thus tasks such as tracking per-flow counters are unfeasible in this approach.

3.4.6 UnivMon

Univmon [32] is a Passive, Switch Assisted solution that employs Sketching to monitor the network.

Similarly to OpenSketch [21], the Control Plane regularly sends a *manifest* to every switch, stating the sketches it will compute. Nonetheless, this solution differs from the latter in two main ways: To begin with, it employs universal streaming algorithms [38, 39] to compute more metrics with fewer sketches. Furthermore, the Controller runs an optimization algorithm to assign sketches to each switch in a way that reduces the required computation at each switch. The authors noticed that if every switch computed the same set of sketches, that redundant computation would hinder the solution performance, thus, the employed optimization algorithm assigns sketches to a subset of switches that cover every flow. Ideally, that subset would be the smallest possible, however doing so would assign every sketch to the same subset of switches, which would waste the computing power of every other switch. For this reason, the algorithm also pretends to evenly distribute sketches among switches. This way, UnivMon successfully creates a "one big switch" abstraction [46], i.e. it is able to monitor the network with the same detail as if it were a single switch.

Since the metrics computed by UnivMon only consider the top-k flows, it effectively reduces the communication overhead by identifying those flows in the Data Plane and sending the respective counters to the Controller. Notwithstanding, UnivMon still lacks in the variety of metrics it can compute [47] and its accuracy is below desirable [48].

3.4.7 FlowRadar

FlowRadar [18] is a Passive, Switch-Assisted solution that uses Coding to track, for each flow, the number of packets that were processed by each switch in the network. It then uses that information to locate packet drops and to identify routing loops and black-holes in the network.

FlowRadar keeps a table to store per-flow counters and uses hashing to directly access the table entries. However, as opposed to other solutions, it encodes colliding flows into the same entries. To do so, it keeps a Bloom Filter to track the flows that were already registered and a table that stores the flow counters. Each table entry contains three fields: *FlowXOR*, *FlowCount* and *PacketCount*, containing, respectively, a cumulative XOR of flow identifiers, the number of flows that were mapped to that entry and the total number of packets that were accounted for every flow. For each incoming packet, FlowRadar calculates *l* different hashes that will index *l* different rows where that packet will be accounted and increments the *PacketCount* field of all those entries. If the Bloom Filter indicates this packet belongs to a new flow, FlowRadar registers it in the Bloom Filter and then, for each of the *l* rows, it proceeds to XOR the flow identifier in the *FlowXOR* field and increments the *FlowCount* entry by one.

Each switch periodically sends this table to a remote Controller, which uses its increased computational power to decode this information. To do so, it first identifies the entries that store a single flow, by checking the *FlowCount* field of each row. The values present in the *FlowXOR* and *PacketCount* fields of those entries correspond to the identifier (*flow_id*) and packet count (*count*) of the respective flow, hence these are called *pure entries*. For each pure entry, the Controller determines the other rows that flow was encoded into by calculating the same *l* hash values that were computed in the switch, and proceeds to remove the information related to that flow from the other entries. To do so, it i) XORs the *flow_id* into the *FlowXOR* field, ii) subtracts *count* from the *PacketCount* field and iii) decrements the *FlowCount* by one. This process may generate new pure entries, which will allow to decode more flows, thus it is repeated until there are no pure entries left.

When decoding, it is possible to exhaust the pure entries in the table, while still having entries to decode, which makes further decoding impossible. In this situation, FlowRadar leverages the information received from other switches to decode more flows: For every neighbouring switch pair $switch_i$ and $switch_j$, FlowRadar finds the flows that were decoded by the first but not by the latter and, from those, selects the flows that were registered in $switch_j$'s Bloom Filter. For those flows, FlowRadar uses the hashes employed in $switch_j$ to get the rows where those flows were stored and proceeds to remove the respective information from those entries. If this process generates new pure entries, FlowRadar can further decode more flows. Due to possible packet loss between neighbouring switches, packet count decoding must be performed by solving a linear equation system, created from the *PacketCount* values of each table and the combination of the entries where each flow was mapped to. Nonetheless, this mechanism cannot guarantee that every flow will be successfully decoded, which may hinder the monitoring task.

Although this solution detects packet drops in the network, FlowRadar can only locate faulty links if the monitoring switches are connected by a physical link. This constraint does not allow to deploy this solution in networks with a mix of programmable and non-programmable switches. Additionally, packet duplication may conceal packet drops, as switches have no mechanism to prevent counting the same packet twice.

3.4.8 NetSight

NetSight [20] is a Passive, Switch Assisted monitoring solution that tracks every step a packet takes inside the network and uses Compression to minimize the generated traffic. This information is then used by numerous applications to perform a plethora of monitoring tasks, including locating packet drops, and identifying routing loops and black holes. This solution assumes that every switch in the network can be remotely configured using OpenFlow [13] and is connected to a NetSight Server, that will be collecting its reports. There may be more than one NetSight Server in the network.

Whenever a switch processes a packet, it creates a *postcard* of that packet and sends it to the NetSight Server it is assigned to, for future analysis. A postcard contains the packet headers, the identifier of the switch that created the postcard, and the port that packet was forwarded to. NetSight aims to aggregate the postcards of the same packet into the same *packet history*, however these postcards may be scattered across the network, as the switches the packet passed through may be assigned to different NetSight Servers. For this reason, NetSight Servers regularly reshuffle the postcards they have received, using the hash of the flow identifier to determine the Server each postcard will be sent to. This mechanism successfully aggregates postcards of the same flow into the same Server while equally distributing the flows across every Server.

NetSight effectively reduces traffic overhead and memory costs by compressing postcards and histories, before shuffling and storing, respectively. It leverages the similarity of consecutive packets and employs delta encoding to reduce their size. Finally, it uses a standard compression algorithm to further minimize its size. Although it is possible to use Filtering or Sampling to reduce the traffic overhead, those techniques would inhibit a full traffic coverage and consequently hinder the monitoring capability.

The main feature of this solution is its ability to monitor every packet circulating in the network, which grants a high coverage to this solution. Nevertheless, the bandwidth and computing power required to shuffle and compress postcards of every packet become intolerable in networks with high traffic intensity [4]. Additionally, to fully detect anomalies, this solution requires to generate a postcard at every switch, thus it could not be deployed incrementally in an already functioning network.

3.4.9 NetSeer

NetSeer [3] is a Passive, In-Switch monitoring solution that detects and locates network anomalies, such as packet drops, congestion, path change and routing loops. It leverages the Data Plane programmability to effectively Select the packets that experience the targeted anomalies and sends that information to an external storage for future queries. Each reported anomaly contains information about the affected flow, as this information helps reducing the anomaly detection and recovery time.

Packet drops can occur either inside the switch (*intra-switch* packet drops), for example, due to invalid header formats or congestion, or in the link that connects two switches (*inter-switch* packet drops). This solution effectively detects intra-switch packet drops by following the packet processing pipeline and creating an event reporting packet drops whenever one is detected. For instance, NetSeer generates a packet drop event whenever a packet is discarded due to full queues.

NetSeer runs a switch coordination algorithm to detect inter-switch packet drops. Every switch maintains a sequential number for each of its outgoing ports, representing the sequence of packets that were sent to each, and attaches it to every packet that is sent to the respective port. They also record the highest sequence number received from each incoming connection, and updates it as it receives new packets. NetSeer assumes switches are connected by physical links, and for this reason, packets will arrive in a FIFO order. Therefore, if the downstream switch receives a packet with a sequence number new_seq such that $new\_seq - old\_seq > 1$, being *old_seq* the previously stored sequence number for that ingress port, then it is sure that $new\_seq - old\_seq - 1$ packets were dropped.

To keep the flow-event mapping, the upstream switch keeps a buffer where it stores the flow associated with each sequence number, for each egress port. When the downstream switch detects a packet drop, it informs the upstream which sequence numbers were missing and the latter then uses the buffer to identify the flows that suffered those drops.

NetSeer employs two techniques to further reduce the traffic overhead it generates. First, it aggregates events affecting the same flow into the same *flow event*. Each flow event stores the affected flow identifier, the number of affected packets and other event-related information, such as queuing latency for congestion events, or drop cause for packet drop events. The flow-event is reported whenever the number of affected packet exceeds certain thresholds. Second, this solution uses packet recirculation in the switch to aggregate multiple flow events into the same message before sending it to the external storage. As flow events are smaller than the minimum ethernet frame size, this technique promotes an efficient bandwidth usage.

One of the main features of this solution is its ability to detect inter-switch packet drops inside the network. Nonetheless, the switch coordination algorithm assumes that the monitoring switches are connected by a FIFO link – i.e., they are *directly* connected – which does not hold in every situation. For instance, if the switches are interconnected by another network, this assumption does not hold. Even

System	Activity	Location	Prb	Mir	Spl	Flt	Cpr	Skt	Cdg	Sel
PingMesh	Active	Host-Based	✓							
NetBouncer	Active	Host-Based	✓							
NetSight	Passive	Switch-Assisted		✓			✓			
Planck	Passive	Switch-Assisted		✓	✓					
Everflow	Passive	Switch-Assisted	✓	✓	✓	✓				
OpenSketch	Passive	Switch-Assisted			✓	✓	✓	✓		
Univmon	Passive	Switch-Assisted						✓		
FlowRadar	Passive	Switch-Assisted							✓	
NetSeer	Passive	In-Switch					✓			✓
WBMon	Passive	In-Switch								✓

Table 3.1: State of the art comparison. Prb. stands for Probing, Mir. for Mirroring, Spl. for Sampling, Flt. for Filtering, Cpr. for Compression, Skt. for Sketching, Cdg. for Coding, and Sel. for Selecting

in the case this solution is run in a single-domain network, in the common situation where there is a mix of programmable and non-programmable switches, packet reordering may occur, and the NetSeer solution will not be effective. This makes it difficult to gradually deploy this solution in already functioning networks, and is a strong motivation for our work.

3.5 Discussion

Table 3.1 presents a summary and comparison of the previously surveyed systems. The fact that a vast majority is Switch-Assisted reflects the challenge that is implementing the anomaly detection logic in the limited switch resources, and the advantage that comes from having finer network insights.

In this work, we aim to quickly and accurately perform packet drop detection. As there are already solutions that perform this task, we can learn from them and use that knowledge in the design of our solution. And we can also learn from their limitations.

NetBouncer [1] is a relatively recent solution that allows to infer the location of packet drops by correlating the drop rates measured in different network paths. Although this solution effectively locates faulty links with few observation points, it is limited in two ways. First, being an Active solution, NetBouncer may incur in undesirable overhead and miss application traffic anomalies. Second, the coverage accuracy of this solution may be limited because the observation points are restricted to the edge of the network. We hypothesise that having observation points in the core of the network may grant better results.

While NetSight [20], Planck [26] and Everflow [4] are able to detect and locate packet drops, these solutions incur in unacceptable traffic overhead [3], due to the employed mirroring technique. Moreover, the Sampling and Filtering techniques used to reduce this overhead end up damaging the accuracy and coverage of these solutions. Plank, on the one hand, neglects traffic during the occurrence of traffic intensity spikes; Everflow, on the other hand, neglects traffic that is filtered out by its rules.

FlowRadar [18] is also able to passively locate packet drops with low traffic overhead, however its encoding mechanism may lead to data loss that can hinder this task.

Finally, NetSeer [3] is able to passively locate packet drops entirely in the Data Plane, with low traffic overhead and high coverage and accuracy. Nonetheless, this solution is based on the assumption that the entire network is composed of programmable switches, directly connected, and all running NetSeer. These assumptions do not generalise to the most common cases, namely in already functioning networks.

Our goal is to develop a Passive, In-Switch monitoring solution that relaxes the assumptions required by NetSeer. We consider the network is monitored by a set of programmable switches, connected by other non-programmable switches or even an external network, that may reorder, duplicate, and drop packets. To fulfill our objective, we propose an inter-switch drop detection algorithm similar to NetSeer that effectively detects packet drops in the network, while tolerating packet reordering. In addition, we adapt the Failure Inference algorithm used in NetBouncer to help pinpoint faulty links.

Summary

In this chapter we started by characterizing general approaches to network monitoring and specified the most common techniques used when implementing monitoring solutions, and discussed the advantages and disadvantages of each. We have also analysed how these techniques were implemented in several monitoring solutions that were relevant to our work. For each of those solutions, we highlighted their operation and limitations. Finally, we explained how WBMon fits among the described solutions. The next chapter details the design and implementation of WBMon.

4

Window Based Monitoring

Contents

4.1 System Architecture Overview 33
4.2 Underlying Model 35
4.3 Drop Detection Algorithm 36
4.4 Failure Inference Algorithm 41

This chapter introduces Window Based Monitoring (WBMon), a system that detects packet drops in the network and identifies the lossy links that caused them. It leverages the availability of programmable switches (we call them *m-switches*) to insert observation points inside the network, but improves over state-of-the-art solutions (namely, NetSeer and NetBouncer) in several aspects:

First, NetSeer requires every monitored link to be physically connected to two m-switches, which greatly increases the cost of deploying that solution in legacy networks. To do so, operators must either select a small subset of links to be monitored, which hinders coverage, or upgrade a larger amount of switches, which comes with a higher cost. Instead, WBMon is able to detect packet drops in arbitrary long sequences of links that are delimited by m-switches. This way, we can cover a wider range of links, while keeping the number of monitoring switches at a minimum. Additionally, it allows to gradually deploy and increase the solution coverage and accuracy by upgrading new switches over time.

Second, by using passive monitoring, our solution is able to directly monitor application traffic. This not only reduces the signaling overhead required by active monitoring solutions, but also allows to detect network failures that only affect application traffic, such as routing blackholes or ACL misconfigurations. Additionally, passive monitoring allows to collect data at higher rates with no additional cost, which allows to detect transient faults (Section 5.1.4).

Finally, by monitoring packet drops across well defined paths in the network, WBMon is able to identify the set of links that *may* have caused each drop. We incorporated NetBouncer's Failure Inference Algorithm [1] to correlate the packet drops detected along multiple paths and *identify* the faulty links in the network. However, NetBouncer is restricted to monitor the network from its end-hosts, which limits the variety of paths that can be monitored. By performing in-switch monitoring, WBMon can monitor a wider variety of paths and to collect more and better data, which improves the failure inference accuracy.

Section 4.1 presents a bird's-eye view of WBMon's workflow; Section 4.2 formalizes the model assumed in the design of WBMon, and provides important definitions that will be used throughout the document; Section 4.3 describes the algorithm used to detect packet drops between observation points; finally, Section 4.4 details how we adapted NetBouncer's Failure Inference Algorithm to work in the current model.

4.1 System Architecture Overview

This section presents the architecture designed for Window Based Monitoring, and details the role of each component and how they interact with each other. A diagram of the architecture of WBMon can be found in Figure 4.1.

Before deploying WBMon, network operators should determine the optimal location for the m-switches, given the target network topology. This placement should consider the operators' constraints: For ex-

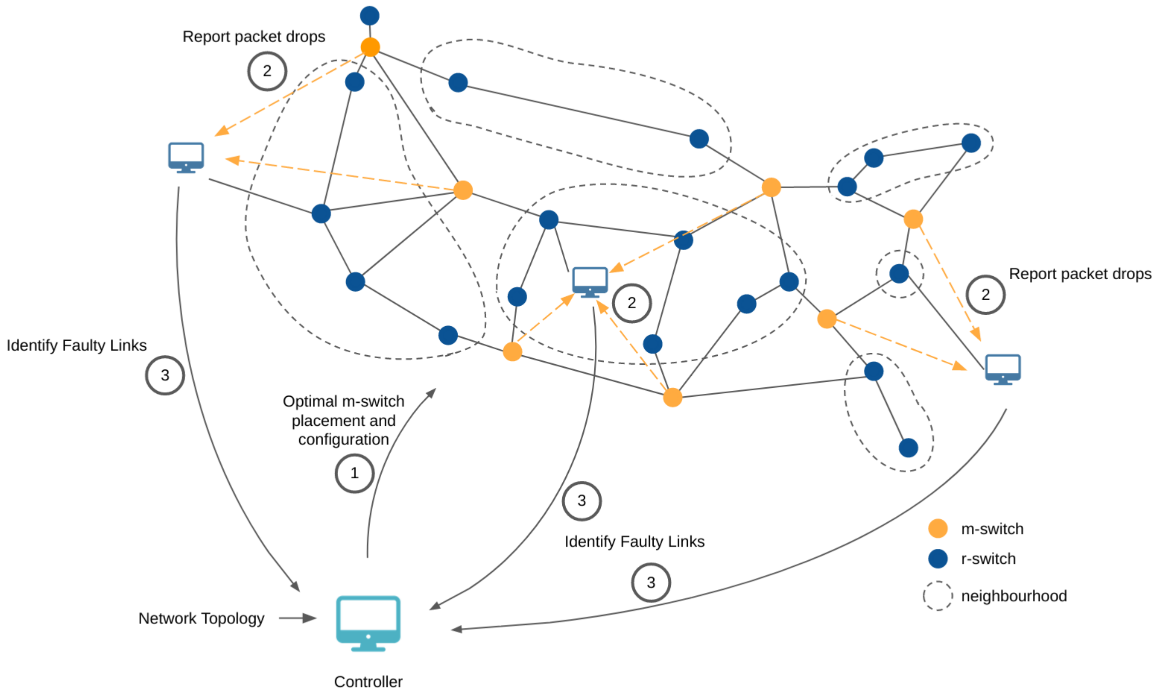


Figure 4.1: WBMon workflow overview

ample, the first WBMon deployment may have budget limitations that cap the number of m-switches that can be installed; In another example, when adding more observation points to an already functioning WBMon instance, it may not be possible to change the location of the already deployed m-switches.

The m-switch placement must be such that the network links are *identifiable* [1], *i.e.* the data collected in the monitored paths should be enough to identify the lossy links in the network. For self-containment we introduce the link identifiability problem defined in [1]:

"Given a network graph G and all its possible paths U, how to construct a set $A \subset U$, so that the set of equations $\{y_j = \prod_{link_i \in path_j} xi | path_j \in A\}$ has a unique solution for all x_is."

Finding the best location for m-switches is an example of an NP-Hard problem, called the facility location problem [49]. Panopticon [50] provides an efficient algorithm to determine the legacy switches that should be upgraded first, according to the operators needs. For this reason, solving this problem is out of the scope of the current work. Nonetheless, Section 5.2 discusses the implementability of WBMon in different network topologies.

After having the m-switches deployed in the network (the yellow circles in Figure 4.1), the Controller configures them with the forwarding rules required to monitor the network. This configuration can lead the m-switches to use different routes for different flows, in order to increase the number of monitored paths. Then, as the application traffic circulates through the network, each pair of m-switches executes the coordination algorithm described in Section 4.3 to count the number of packet drops that took place in each of the paths connecting them. The m-switches regularly send the number of detected packet drops and the number of processed packets to the network *Analysers*, which in turn run the Failure Inference Algorithm to locate the lossy links in the network. Although the default behavior is waiting for the information to arrive from the m-switches, Analysers can also query them if necessary.

When defining the m-switch placement, the Controller can also split the network into multiple *neighborhoods* (Section 4.2), depicted by the dashed regions in Figure 4.1. This technique reduces the task of locating the faulty links in the entire network to locate the faulty links in each individual neighborhood. Additionally, by assigning each Analyser to subsets of neighborhoods, we allow WBMon to scale with the network size.

4.2 Underlying Model

WBMon considers a network consisting of a mix of programmable and non-programmable switches connected by bidirectional links, where only the programmable switches have monitoring capabilities. This network is modeled as an undirected graph $\mathcal{G} = (\mathcal{S}, \mathcal{L})$, where $\mathcal{S}$ denotes the switches, and $\mathcal{L}$ the set of bidirectional links. Additionally, $\mathcal{S}$ is partitioned into two subsets $\mathcal{M}$ and $\mathcal{R}$, where the former contains all the programmable switches (*monitoring switches*, or *m-switches*), and the latter the non-programmable switches (*regular switches*, or *r-switches*). Each link $l_i \in \mathcal{L}$ has a certain probability of successfully transmitting a packet, denoted as x_i. We assume that the success probabilities of different links are independent [1, 51–53]. We also consider that both the topology and the forwarding rules at each switch are known, and do not change over time. This allows us to define a *virtual connection*, or *vconn*, as a sequence of links that connects two m-switches, without containing any loop and without crossing any other m-switch. We assume that a vconn may drop, reorder and duplicate packets, and denote the set of all vconns in $\mathcal{G}$ as $\mathcal{V}$. Note that, as opposed to links, vconns are unidirectional. If two links participate in the same set of virtual connections, we say those links are *indistinguishable* since it is impossible to distinguish them using only the data collected by the m-switches [53]. We denote the set of links that cannot be distinguished from l as $\mathcal{I}(l)$. This concept will be relevant in Sections 4.4 and 5.2.

We also define a *neighborhood* as the set of r-switches and links that form a connected component on the network obtained after removing from $\mathcal{G}$ the m-switches and the links that directly connect two m-switches [50]. This notion is useful since it allows to analyze different neighborhoods independently, which simplifies the problem we are trying to solve, and allows our solution to scale to larger networks.

Figure 4.2 presents a network composed of 4 m-switches and 4 r-switches, organized into two neighborhoods: $\{2\}$ and $\{4, 6, 7\}$. We can see that there are three vconns departing from 5: $(5 \to 6 \to 4 \to 3)$, $(5 \to 6 \to 4 \to 7 \to 8)$, and $(5 \to 6 \to 7 \to 8)$. Note that there exists no vconn connecting m-switches

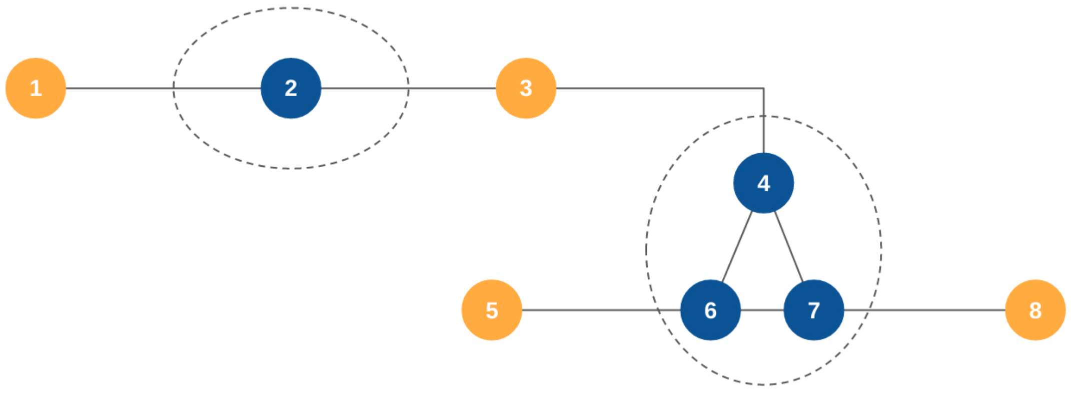


Figure 4.2: Simple network with two neighbourhoods.

5 and 1 since every path from 5 to 1 traverses one m-switch. We can also see that the links connecting switches 1 to 2, and 2 to 3 are indistinguishable, since both belong to the same vconn. If switch 3 detects packet drops in the traffic coming from 1, it is impossible to know if they occurred in switch 2 or in any of the links it is connected to.

4.3 Drop Detection Algorithm

In this section we proceed to describe the inter-switch coordination algorithm that executes in the network switch Data Plane. The source code for this algorithm is available in Appendix A.

To better understand the drop detection algorithm, let's assume we have an infinite buffer split into different slots with *W* bits each. Let's also consider that the arriving packets carry a sequence number (*sn*), corresponding to the order in which they were sent to the respective vconn. During the rest of the document, we will refer to the slot in position *i* as s_i, and to packet with sequence number *x* as p_x. We call *current slot* to the slot the arriving packet belongs to, and *newest slot* to the slot that received the packet with the highest sequence number.

The buffer is initially filled with zeros and whenever a packet arrives, the m-switch writes a 1 in the position corresponding to the packet's sequence number. That position consists of a pair $(slot\_idx, offset)$, where the first field determines which slot will register that packet, and the second determines the bit that will be set to 1 in that slot. These values are calculated as follows:

$$slot\_idx = \left\lfloor \frac{sn}{W} \right\rfloor ; offset = sn \bmod W$$

Since packets may be reordered, reporting a packet as dropped immediately after receiving one of its successors could be premature, since that packet may just be "late". To avoid premature reports, we define a tolerance window that waits for late packets to arrive before reporting them as lost. This window comprises the *Tolerance Window size* (*tws*) slots that record the packets with the highest sequence numbers. More precisely, the tolerance window ranges from $s_{\textit{newest\_slot}-\textit{tws}+1}$ to $s_{\textit{newest\_slot}}$, inclusive. We call "tolerance slots" to the slots belonging to the tolerance window.

As new packets arrive, the newest slot will eventually be updated and consequently the tolerance window will slide over the buffer. When this happens, some of the tolerance slots exit the tolerance window range, and the m-switch counts the zeros of those slots to determine the number of dropped packets. We thus assume we have waited enough for those packets, that they were probably not just reordered, but were in fact dropped. Note that despite using a tolerance window, this mechanism will still produce false positives if p_i arrives after $p_{i+W\times \textit{tws}}$. This stresses the importance of having an adequate *tws*: On the one hand, a small tolerance window is more likely to produce false positives. However, if it is too large, it will take too long to detect packet drops. Note that the ideal *tws* depends on the traffic patterns of the network that is being monitored and should be adjusted for each scenario.

In our implementation (in a programmable switch, the Intel Tofino [16]), each slot is a Register. To make efficient use of the finite resources available in the hardware, we reuse the same Register for multiple slots. We assume that the m-switch contains *N* available Registers. Registers were reused in a round robin basis, i.e., slots $\{0, N, 2N, ...\}$ are assigned to Register 0, slots $\{1, N+1, 2N+1, ...\}$ are assigned to Register 1, and so on. We say that slots stored in the same register are *cohabitants*, and that a slot is *active* if it contains information about which packets have already arrived, and *inactive* otherwise.

To assure correctness, each Register must be cleaned before being reused by another slot. Otherwise, the remaining bits set to one could conceal some packet drops. For this reason, we introduced a cleaning window that is responsible to *clean* the *dirty* slots that exit the tolerance window range as it advances. We call the slot that is being cleaned the "*slot to clean*". The cleaning window comprises at most *Cleaning Window size* (*cws*) slots and is, by definition, mutually exclusive with the tolerance window. To maximize reordering tolerance, we set the window sizes such that $\textit{cws} + \textit{tws} = N$. Note that if there are no dirty slots, the cleaning window has a size of zero. The current implementation is able to clean an entire slot for each processed packet. Hence, we conclude that the cleaning window advances *W* times faster than the tolerance window.

Figure 4.3 illustrates the process of advancing the tolerance window and cleaning the dirty slots during the arrival of four packets, in an m-switch with $N = 4$ Registers. For simplicity, we assume each Register contains $W = 10$ bits. In this diagram, slots are represented by rectangles and are arranged such that cohabitant slots are displayed in the same column. The gray circle indicates the position where each packet will be registered.

Initially, we assume the buffer is in its normal state, having no dirty slots and receiving a packet (p_{29})

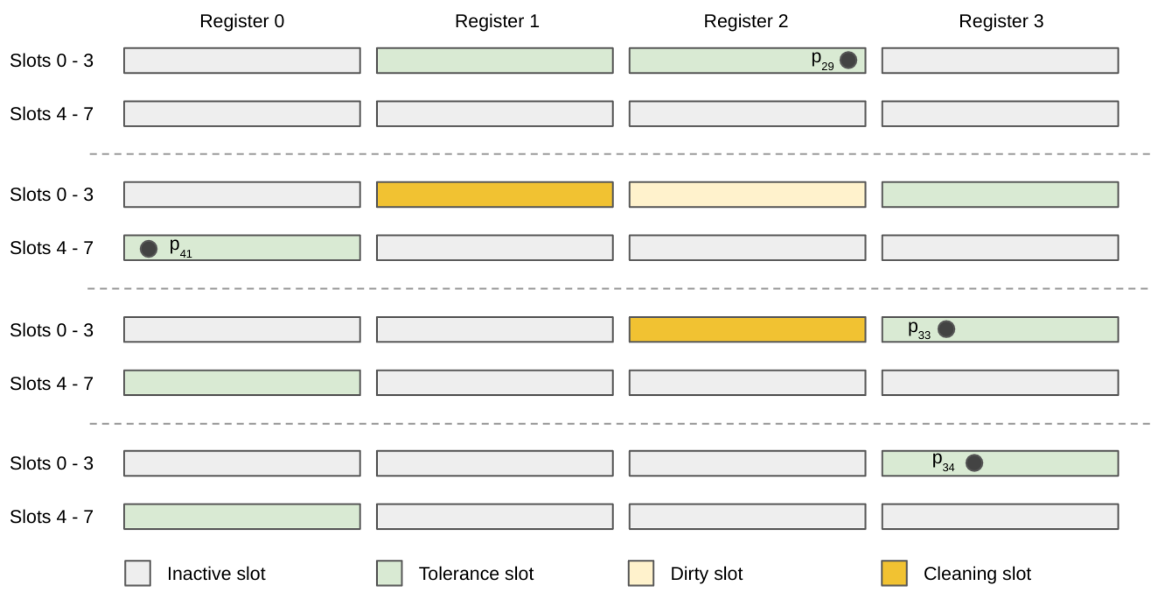


Figure 4.3: Tolerance Window update

belonging to the tolerance window. In this example, the tolerance window consists of s_1 and s_2. Let's suppose that the second packet (p_{41}) is an early packet, and arrives before its predecessors. Since that packet belongs to s_4 and the previously stored *newest slot* was s_2, its arrival will cause the tolerance window to advance by $4 - 2 = 2$ slots. Consequently, s_1 and s_2 become dirty slots and must be cleaned, so that s_5 and s_6 are ready to receive new packets (note that s_1 and s_5, and s_2 and s_6 are cohabitants). During the arrival of p_{41}, we can immediately clean s_1 and detect which of its packets were lost by counting the remaining zeros in that slot. Similarly, we can do the same to s_2 when the third packet (p_{33}) arrives. Finally, when the fourth (p_{34}) packet arrives, there are no more dirty slots and the buffer returns to its normal operation.

The algorithm described in this section is trivially extended to a network with multiple vconns. To do so, each m-switch must keep, for each incoming vconn, *N* Registers to store the buffer, and two Registers to store the *newest slot* and the *slot to clean*. For each outgoing vconn, each m-switch must only keep a single register, that tracks the next sequence number that is going to be sent to the respective vconn. In this work we assume that the Controller regularly resets the buffers and sequence numbers of m-switches such that sequence numbers do not grow boundlessly.

The resources available in the adopted switch model allowed us to use $N = 4$ Registers with $W = 32$ bits each, for 256 vconns. We believe that, as these switches evolve, there will be more Registers available and that, in time, these buffers may be larger and comprise more slots.

4.3.1 Handling Edge Cases

From Figure 4.3, one may notice that the arrival of packets with certain sequence numbers may lead our solution to produce incorrect results. For instance, if a packet arrives after its slot is cleaned, we know that packet was falsely considered dropped. We can define a safety slot interval that guarantees the solution correctness if every packet falls in it. This interval can be described with the following equations:

$$\textit{current slot} > \textit{slot to clean} \tag{4.1}$$

$$\textit{current slot} < \textit{slot to clean} + N \tag{4.2}$$

Where *current slot* denotes the slot the arriving packet belongs to, and *slot to clean* denotes the slot that will be cleaned during the processing of that packet. If there are no dirty slots, the *slot to clean* corresponds to the first tolerance window slot.

Despite having the above integrity conditions, they may not hold in a real world application. Therefore, it is important to determine how to behave if these conditions are not met. Ideally, m-switches would be able to detect the violation of these conditions, and emit a message reporting the incident. However, the hardware limitations of the Tofino switches inhibited this behavior. In this discussion we aim to find the responses that minimize the impact of those errors.

When a late packet p_l arrives such that it violates condition 4.1, it is likely that p_l was previously considered dropped. One could argue that decrementing the drop count, or registering such packet anyways could compensate the previously produced false positive. However, it is possible that p_l is a late duplicate packet. If so, registering such packet could produce more false negatives, which would consequently degrade the accuracy of our solution. Instead, we opt to ignore that packet and accept the possibility of having a false positive.

When an early packet p_e arrives such that it violates condition 4.2, it is impossible to know if its predecessors were reordered or dropped until they arrive. This scenario requires two decisions: whether to update the newest slot, and whether to register the arrival of p_e.

Regarding the first decision, if the newest slot is not updated, the presence of a drop burst with more than $N \times W$ packets could completely stop our algorithm from working: In that case, none of the packets that arrived after the drop burst would be able to update the newest slot, since their slot would violate condition 4.2. Consequently, the tolerance window would not move and no drops would be detected. On the other hand, if p_e's predecessors were reordered instead of dropped, updating the newest slot would generate false positives, since some of those packets would violate condition 4.1 at the time of their arrival. We argue that having a finite number of false positives is better than having the algorithm completely stopped, thus we conclude that the newest slot must be always updated.

Regarding the second decision, if we do not register p_e, it will inevitably be considered dropped, generating a false positive. However, if the packet is registered in the buffer, even if in a wrong slot, it is possible that p_e's offset coincides with the offset of a lost packet. If this happens, there would be no false negative nor false positive generated. Hence, we opt to register early packets, since it provides a non-zero chance of having a correct result, despite of the fault.

Finally, it is important to decide what to do if the current slot coincides with the cleaning slot. In this situation, the same Register should be cleaned to detect packet drops, but should also register the arriving packet. As seen in Section 2.3, it is only possible to perform a single action on a Register for each processed packet, forcing us to decide which one to take.

If we record the packet instead of cleaning the slot, the respective Register will remain dirty when the next cohabitant slot activates. This behavior may conceal packet drops and produce up to *W* false negatives. On the other hand, if we clean that slot instead of registering the packet, the solution will produce, at most, a false positive. Thus, the current implementation prioritizes cleaning the slot over registering it.

4.3.2 Exchanging Sequence Numbers

An important decision to make when implementing this algorithm is how to send the sequence numbers and vconn identifiers between m-switches. This information needs to be transparent to hosts and r-switches, i.e. we want these devices to be unmodified for our solution to work. In this section we provide a brief discussion on the possible locations where packets can carry this information, and analyse the advantages and disadvantages of each option.

A first option that may come to mind is leveraging unused fields in packet headers, such as VLAN fields or IP Options, to carry the desired information. This approach has the advantage of causing little traffic overhead. However, it only allows to monitor the traffic that uses such protocols, and additionally inhibits existing traffic from using those fields. This option is viable in controlled environments, such as data center networks [3, 4].

Another alternative is inserting this information as a thin layer between Layers 3 and 4. This may be a more suitable option in networks with a wider variety of traffic, as it does not interfere with any protocol in Layers 2 and 3. One must be aware that this option may hinder load balancing, or interfere with other security/monitoring mechanisms that require Layer 4 data [54, 55]. It is also necessary to guarantee that this thin layer is removed in the egress vconn to guarantee it is not sent to end hosts.

We conclude that this decision depends on the target network characteristics and should be made on a case-by-case basis. The current implementation used the IP Options field to carry this information.

4.4 Failure Inference Algorithm

The technique we proposed in the previous sections enable monitoring the number of packet drops in multiple vconns. We now propose to correlate that data to locate the links that are responsible for those drops. As discussed in Section 3.4.2, NetBouncer [1] proposes an optimization algorithm to locate the faulty links in a network. For self-containment, the next paragraphs include a brief description of that algorithm we will adapt to our work. For a more in-depth analysis, the reader can consult [1].

By assuming that the packet loss events are independent in different links, one can calculate the success probability $\hat{y}_i$ of *path*$_i$ as the product of the success probabilities of the links composing it. More precisely:

$$\hat{y}_i = \prod_{j:l_j \in \mathit{path}_i} x_j$$

One can combine this notion with the measured success probabilities of each vconn (y_i), and formulate an optimization problem that finds the values of x_i that minimize the error between y_i and $\hat{y}_i$:

$$\begin{aligned} \text{minimize} \quad & E = \sum_{j:\mathit{vconn}_j \in \mathcal{V}} (y_j - \prod_{i:l_i \in \mathit{vconn}_j} x_i)^2 \\ \text{subject to} \quad & 0 \le x_i \le 1, \forall i \end{aligned} \tag{4.3}$$

After finding the optimal values of x_i, NetBouncer identifies the faulty links as the ones that have a success rate lower than a user defined threshold.

Note that NetBouncer calculates y_i as the ratio of bouncing packets that returned to the sending host. Instead, our solution calculates those values using the drop and packet counters from the m-switches. This approach allows to perform passive monitoring, which comes with several advantages, discussed in Section 3.1.

The existence of indistinguishable links may lead the failure location mechanism to detect a wrong set of faulty links. This happens since those links belong to the same set of vconns, therefore, permuting the x_i of those links will not change the value of E. In fact, as the algorithm that solves the above optimization problem is initialized with random values, it is possible that it swaps the values assigned to indistinguishable links in different executions. For this reason, our failure inference algorithm yields the calculated success probabilities of each link along with the sets of indistinguishable links. With this information, network operators should investigate both the links with values of x_i below the stipulated threshold, as well as the links that cannot be distinguished from those.

Summary

This chapter describes the design and implementation details of WBMon. By performing a reorder tolerant inter-switch coordination algorithm, our solution is able to detect packet drops between pairs of switches entirely in the Data Plane. By correlating the information collected from different switches, our solution is able to identify the set of faulty links.

5 Evaluation

Contents

5.1 Drop Detection Algorithm 45
5.2 M-Switch Placement 55

This chapter details the experiments performed to evaluate our solution. Section 5.1 focuses on the Drop Detection algorithm. Its goal is to measure the detection accuracy for different drop and reordering configurations, and to discuss the limitations of the developed algorithm. Section 5.2 entails a theoretical analysis that aims to understand the impact of the chosen m-switch arrangement on WBMon's performance, and study the suitability of implementing our solution in two common real world topologies.

5.1 Drop Detection Algorithm

In this section we detail the experiments conducted to measure the accuracy of our solution. We start by performing controlled benchmarks to understand how WBMon tolerates packet loss and reordering (Sections 5.1.1, 5.1.2), and then test it against realistic packet traces (Section 5.1.3) to study its behavior under a real-world scenario. Finally, we estimate the time required for the Drop Detection Algorithm to correctly identify the packet drop rate of each monitored vconn.

These experiments were executed using a single *m-switch* and a single Controller, to reduce possible errors originated outside of our algorithm. Both the Controller and the *m-switch* were executed on a Virtual Machine running Ubuntu 20.04.4 LTS, with 4GB RAM. The *m-switch* was emulated using the Intel Tofino SDE 9.7.0. The implementation was configured with tolerance window of size $\textit{tws} = 2$ and uses $N = 4$ registers of $W = 32$ bits each.

In each experiment, we used a traffic generator to produce packet sequences that are be sent to the emulated switch. These sequences are configured to contain arbitrary packet drops and reordering, and packets were guaranteed to arrive the switch in the same order they were generated. After executing each experiment we query the number of drops detected by the switch (*detected_drops*). By comparing it with the number of actual drops (*n_drops*), we can calculate the number of False Positives (*#FP*), and False Negatives (*#FN*). *#FP* corresponds to the number of reported drops that do not correspond to effective packet drops, and is obtained as $\max(0, \textit{detected\_drops} - \textit{n\_drops})$; The *#FN* denotes the number of packet drops that were not detected by the switch, and is calculated as $\max(0, \textit{n\_drops} - \textit{detected\_drops})$. Note that, by definition, it is impossible to simultaneously have False positives and False Negatives.

5.1.1 Reordering Tolerance

In this experiment the goal is to understand the extent to which our solution tolerates packet reordering. For that purpose, we generated multiple traces, each with a single reordered packet, and varied the reordering displacement from -128 to 128. These traces did not have duplicates nor dropped packets. To isolate the effect of packet reordering, every trace consisted of 266 packets, starting with the same sequence number, and the reordered packet was the same in every trace (p_{165}). The switch was reset to the same initial state for each trace. Since the traces did not include packet drops, the experiment does not produce any False Negative. Hence we will only analyse the number of False Positives.

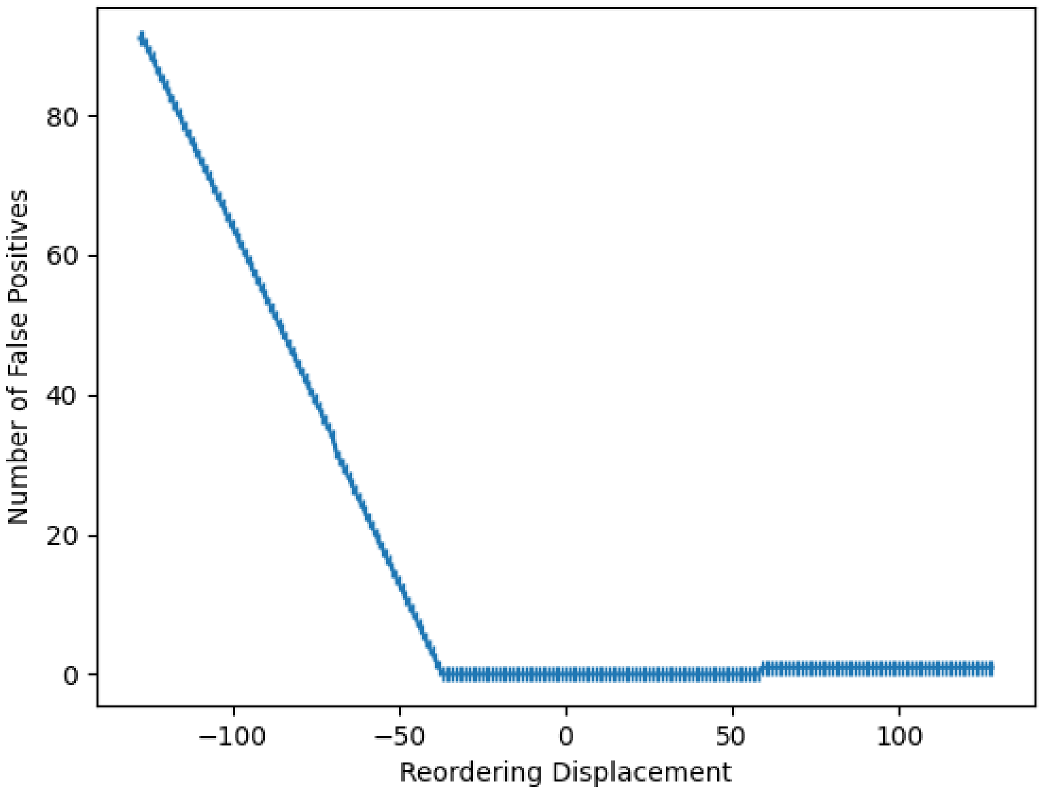


Figure 5.1: Number of False Positives (*#FP*) for different Reordering Displacement values

We expect errors to arise from the reordered packets that violate the safety conditions 4.1 and 4.2, as discussed in Section 4.3.1. Figure 5.1 presents the results obtained from the execution of this experiment. We can see that our solution produced no False Positives for traces with a reordering displacement in range $[-37, 59]$; a single False Positive for traces with a reordering displacement greater than 58; and a linearly decreasing number of False Positives for traces with reordering displacement lower than -37.

The packets that were reordered with a displacement in the range $[-37, 59]$ simultaneously satisfied safety conditions 4.1 and 4.2. Hence, they were correctly registered in the respective slot and did not produce any error. The packets with reordering displacement greater than 59, in turn, reach the switch after their respective slot is cleaned. Hence, those packets are considered dropped before they arrive, and the solution produces a single False Positive.

When the reordered packets have a displacement lower than -37, they arrive too early and lead our solution to prematurely advance the tolerance window. When this happens, the switch will start cleaning the slots of some packets that did not arrive yet and will inevitably consider those packets as dropped. Note that the earlier a packet arrives (the smaller the reordering displacement), the fewer of its predecessors are registered in the buffer, and consequently the more of them will be falsely considered dropped.

We can generalize the tolerated reordering displacement with the following reasoning: A late packet belonging to slot$_k$ will always generate a single False Positive if it arrives after the first packet belonging to slot$_{k+\textit{tws}}$. Hence, packet with *sn* $= i$ will produce a False Positive if it arrives after a packet with *sn* $= i + (W - \textit{offset}) + W(\textit{tws} - 1)$. In the previous expression, $W - \textit{offset}$ corresponds to the distance between i and the first position of slot$_{k+1}$, with *offset* $= i \bmod W$; and $W(\textit{tws} - 1)$, in turn, corresponds to the distance between the first position of slot$_{k+1}$ and the first position of slot$_{k+tws}$. Hence, by the displacement definition given in Section 2.4, our solution tolerates packet reordering with displacement higher than $(i + (W - \textit{offset}) + W(\textit{tws} - 1)) + 1 - i = W \times \textit{tws} - \textit{offset} + 1$. We can confirm this result with the values obtained in the present experiment: $32 \times 2 - 5 + 1 = 60$. In fact, if p_{165} was reordered with a displacement of 60, then the last packet received before the reordered packet will have *sn* $= 60 + 165 - 1 = 224$, which belongs to the first position of slot $5 + 2 = 7$.

We can take an analogous reasoning to determine the solution tolerance to early packets: An early packet belonging to slot$_k$ will cause a premature tolerance window update if it arrives before the last packet of slot$_{k-\textit{tws}}$. If that happens, the arrival of such packet will cause the tolerance window to advance to slot$_k$, and consequently slot$_{k-\textit{tws}}$ will be cleaned before all of its packets arrive, producing False Positives. In other words, a packet with *sn* $= i$ will cause a premature tolerance window update if it arrives before a packet with *sn* $= i - (\textit{offset} + 1) - W(\textit{tws} - 1)$. Here, $(\textit{offset} + 1)$ corresponds to the distance to the last offset of slot$_{k-1}$, and $W(\textit{tws} - 1)$ corresponds to the distance between the last position of slot$_{k-1}$ to the last position of slot$_{k-\textit{tws}}$. Using the displacement definition given in Section 2.4, we can say that our solution tolerates packet reordering with displacement greater or equal than $(i - (\textit{offset} + 1) - W(\textit{tws} - 1) + 1) - i = -W(\textit{tws} - 1) - \textit{offset}$. This means that, if the reordered packet contained *offset* $= 5$, the current solution will tolerate packet reordering with displacement greater or equal than $-32 \times (2 - 1) - 5 = -37$. In fact, if $\textit{max\_sn} + 1 - 165 = -37$, then the last packet received before the reordered packet had *sn* $= 127$, which belongs to offset $127 \bmod 32 = 31$ of slot $\lfloor \frac{127}{32} \rfloor = 3$. Since the last packet of slot 3 was successfully registered before the reception of the premature packet p_{165}, then the solution produces no False Positives.

From this result, we can define the number of False Positives generated as a function of the reordered packet offset and displacement:

$$\textit{\#FP}(\textit{disp}, \textit{offset}) = \begin{cases} -W(\textit{tws} - 1) - \textit{offset} - \textit{disp} & , \textit{disp} < -W(\textit{tws} - 1) - \textit{offset} \\ 1 & , \textit{disp} > W \cdot \textit{tws} - \textit{offset} \\ 0 & , \textit{else} \end{cases} \tag{5.1}$$

By assuming that the reordered packet can belong to any offset with equal probability, i.e. $\mathit{offset} \sim \mathit{unif}\{0, W\}$, we can express the expected number of False Positives as a function of the reordering displacement only [56]:

$$\mathit{EFP}(\mathit{disp}) = \frac{1}{W} \sum_{\mathit{offset}=0}^{W} \#\mathit{FP}(\mathit{disp}, \mathit{offset}) \tag{5.2}$$

We then used the data collected in [57] to estimate the number of False Positives generated for different reordering ratios. In that work, the authors presented a histogram containing the number of packets reordered with different displacement values. The displacement metric used in that work is the same that is used in the current one. We used the trace $F_{600}(\text{UDP}, \text{DC} \rightarrow \text{LA}, 1500)$, which is the one that contains the highest amount of packet reordering. That trace consists of more than 6.1 million UDP packets, in which 0.38% of them were reordered. We measured the *EFP* of the respective displacement value for every packet and computed the average number of False Positives emitted per reordered packet (*FPpR*). With this value, we can estimate the False Positive Ratio as follows:

$$\mathit{FPR} = \frac{\#\mathit{FP}}{\#\mathit{packets}} = \frac{\mathit{FPpR} \times \#\mathit{reordered}}{\#\mathit{packets}} = \mathit{FPpR} \times \mathit{reordering\ ratio}$$

Figure 5.2 illustrates the expected False Positive Ratio for different Reordering Ratio values. We estimate that when the reordering ratio is 1.65% (the maximum registered in [57]), our solution will only consider 0.12% of those packets as dropped. This means that the measured success rates of each vconn will have an error smaller than 0.2% (for reordering ratios measured in [57]), which is tolerated by the Failure Inference Algorithm [1].

5.1.2 Drop Burst Tolerance

This experiment's goal is to evaluate how our solution tolerates drop bursts. To this purpose, we generated several traces, each dropping a burst of successive packets, of size varying from 0 to $3NW - 1 = 383$. To isolate the effect of the drop burst, these traces did not contain duplicates nor reordered packets. Additionally, to avoid external noise, every trace started with the same sequence number (32) and the drop bursts also started in the same packet (p_{112}). The switch was reset to the same initial state for each trace.

Figure 5.3 depicts the variation of False Positives and False Negatives for different burst sizes. We can see that the solution produced no False Negatives in the entire experiment, meaning that every drop was correctly identified. Additionally, one can note that the number of False Positives gradually increases with the burst size. More specifically, the solution produces 0, 1, 2, and 3 False Positives for drop bursts of size in range $[0, 80[$, $[80, 208[$, $[208, 336[$, and $[336, 383]$, respectively. In these intervals there are also certain drop burst sizes that make WBMon produce an additional False Positive. In the

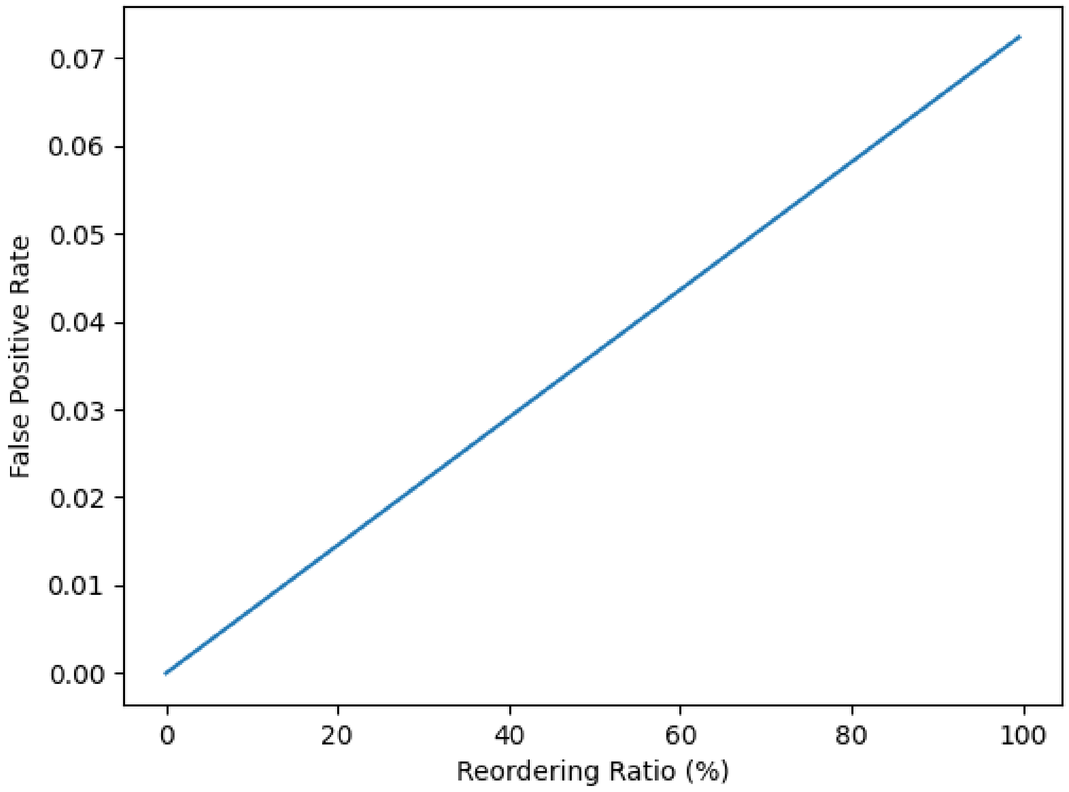


Figure 5.2: Estimated False Positive Ratio for different Reordering Ratio values

following paragraph we explain the cause of the slow increase on the number of False Positives, and the subsequent paragraphs detail the origin of the False Positive spikes.

When the first packet after the drop burst arrives, it will cause the tolerance window to slide over the buffer, which will generate new dirty slots. When the number of dirty slots is greater than N, some Registers will be cleaned more than once. It is inevitable that, during this process, the cleaning slot cohabits with the slot of the incoming packet, as there is no packet reordering. As discussed in Section 4.3.1, when this happens we opt to clean the slot, instead of registering the incoming packet, which produces a False Positive. We can consider the trace that dropped 80 packets to better understand this dynamic: The last packet received before the drop burst had a sequence number equal to 111 and belonged to slot $\lfloor \frac{111}{32} \rfloor = 3$. Since this was the most recent packet received, the newest_slot will be set to 3, and the tolerance window will comprise slots 2 and 3. The first packet arriving after the drop burst will have a sequence number equal to $112 + 80 = 192$ and will belong to offset $192 \bmod 32 = 0$ of slot $\lfloor \frac{192}{32} \rfloor = 6$. The arrival of such packet will update the tolerance window to comprise slots 6 and 5, making slots 2 and 3 dirty. This will make the switch clean slot 2 while registering the incoming packet in slot 6. Since

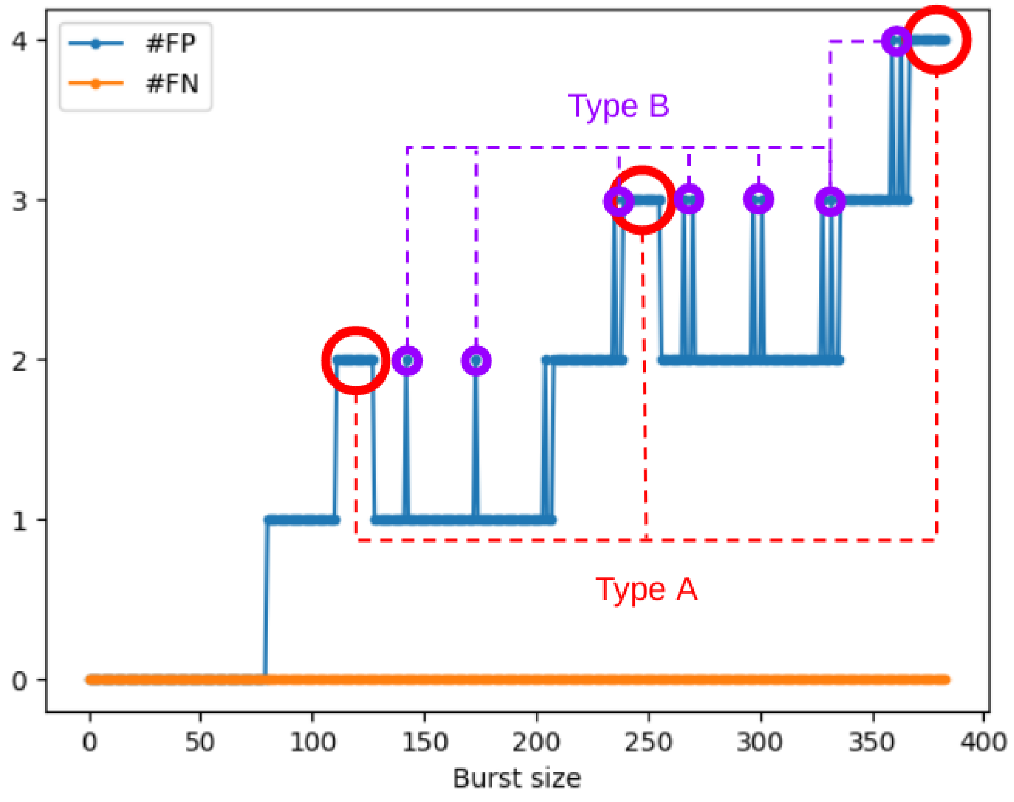


Figure 5.3: Number of False Positives and False Negatives generated for different drop burst sizes

slots 2 and 6 are cohabitants, it was only possible to perform one of those actions. We opted to clean the respective Register instead of registering the arrival of p_{192} (Section 4.3.1), which generated a False Positive. This phenomenon keeps occurring as the burst size increases. However, when it is greater or equal than $80 + NW = 208$, there will be two packets whose slot will cohabit the cleaning slot, which will generate two False Positives. Note that the value 80 corresponds to the distance between the position of the last packet received before the drop burst (p) and the first position of slot $i +$ *tws* $+ 1$, being i the slot of packet p. It can be expressed as $(W -$ *offset*$) + W \times$ *tws*, where *offset* corresponds to the offset of p. We can generalize this result and conclude that when the drop burst size is greater than $(W -$ *offset*$) + W \times ($*tws* $+ kN)$, WBMon will produce at least $k + 1$ False Positives.

There are two circumstances that may generate the additional False Positive that can be seen in Figure 5.3. We classify those as either Type A or Type B, depending on their origin. The False Positives of Type A are generated when the drop burst has a size such that the first packet arriving after the burst is registered in a slot whose bit of the corresponding offset is already set to one. As discussed in Section 4.3, this generates one False Positive.

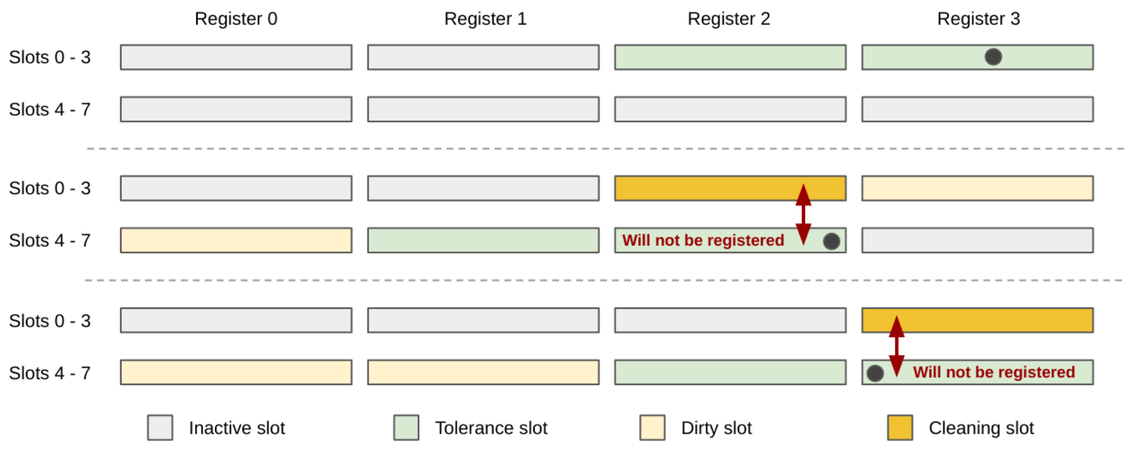


Figure 5.4: Example illustrating the cause of Figure 5.3 Type B spikes

The False Positives of Type B, in turn, arise from drop bursts with size such that one of the packets that cannot be registered has *offset* $W - 1$. Figure 5.4 illustrates the origin of such False Positives. In this scenario, there are multiple consecutive packets that are dropped after the arrival of the first packet depicted in the image. The next packet to arrive belongs to slot 6 and will induce the advancement of the tolerance window and mark slots 2, 3 and 4 as dirty. Since this packet belongs to a slot that cohabits the cleaning slot, it will not be registered, as discussed previously. However, as this packet belongs to the last position of its slot, its successor will belong to the first position of the subsequent slot. Hence, when the third packet arrives, its corresponding slot will once again cohabit the cleaning slot and an additional False Positive will be generated.

The absence of False Negatives in this experiment allows us to conclude that WBMon will be able to detect every packet drop in a drop burst, despite of its size. This happens since we allow the same Register to be cleaned multiple times when the Tolerance Window advances by more than kN slots. Note that our solution may generate False Positives in the presence of drop bursts larger than $(W - (W-1)) + W \times (\textit{tws} + 0 \times N) = W \times tws + 1$ In the used implementation that represents less than $\frac{2}{65} = 3\%$ of the dropped packets, in the worst case.

5.1.3 Performance with realistic traces

We now try to understand how our solution performs in realistic scenarios, namely during the normal network operation, in the presence of a faulty switch or link in the vconn, and during a Denial of Service Attack. To this end, we generated multiple traces, each consisting of 500 000 packets, and configured each trace drop and reorder probabilities according to the scenario we wanted to simulate. The reordered packet displacement followed a normal distribution $\mathcal{N}(\mu = 0, \sigma^2 = 0.75^2)$ [57], and we considered 1% of duplicate packets in every trace. The reorder and drop ratios for each scenario are summarized in Table 5.1. Under normal operation, the literature shows that networks lose 1% of packets [3, 25, 58] and reorder 1.65% of them [57]. To simulate the presence of a faulty link in the monitored vconn, we kept the default value for the reordering ratio and increased the drop rate to 90% and 95% [53, 59–61].

We assumed that in the presence of a faulty switch in the vconn, the amount of packet reordering would increase significantly, and that there would be more dropped packets than under a normal circumstance. Therefore, we used 70% and 75% for the reordering probability and 5% and 15% for the drop ratios.

Finally, we considered that under a DoS attack, both the reordering and drop ratios would increase significantly. Hence, we used 60% and 70% for the reordering rates and 80% and 90% for the drop rates. We are aware that these arbitrary values may not coincide with real values, nonetheless they still allow us to evaluate WBMon's performance under several circumstances.

In each experiment, we calculated the False Positive Rate (FPR) by dividing the number of False Positives by the number of packets processed by the switch (#*packets*), and the Detected Drop Rate by dividing the number of detected packet drops by the total of packets (dropped and not dropped). More precisely:

$$\text{FPR} = \frac{\#\text{FP}}{\#\text{packets}}; \text{Detected Drop Rate} = \frac{\#\text{drops}}{\#\text{drops} + \#\text{packets}}$$

Scenario	Configuration		Number of Drops	Detected Drops	#FN	#FP	FPR (‰)	Detected Drop Rate
	Reordering Rate	Drop Rate						
Normal Operation	1.65%	1%	5007.3	5007.3	0.0	0	0.00	1.00%
Failed Link	1.65%	90%	449988.8	450013.6	0.0	24.8	0.05	90.00%
	1.65%	95%	475012.7	476224.1	0.0	1211.4	2.42	95.24%
Faulty Switch	70%	5%	25077.3	25077.3	0.0	0.0	0.00	5.02%
	75%	15%	75089.4	75089.4	0.0	0.0	0.00	15.02%
DoS Attack	70%	80%	400047.4	400049.8	0.0	2.4	0.00	80.01%
	60%	90%	450035.8	450127.8	0.0	92.0	0.18	90.03%
	70%	90%	449728.3	449821.4	0.0	93.1	0.19	89.96%

Table 5.1: Detected Drop Rate under different realistic scenarios

Table 5.1 presents the number of False Negatives and False Positives generated in each experiment. The measurement values shown in each column correspond to the average of 10 executions. The absence of False Negatives in these results indicates that our solution is able to effectively detect every packet drop that takes place in the monitored vconn. As a matter of fact, every execution in every scenario produced exactly zero False Negatives.

We can observe that WBMon produces no False Positives in scenarios with drop rates lower or equal to 15%, despite of the amount of reordered packets. However, when the drop rate is 90%, and the reordered packet ratio is 1.65%, our solution produces an FPR of about 0.05‰. If the number of reordered packets increases to 60% and 70%, while keeping the amount of reordered packets, the value of the FPR increases to 0.18‰ and 0.19‰, respectively. This result suggests that under higher levels of packet loss, our system is more sensitive to the amount of packet reordering.

One can also note that when the monitored vconn drops 95% of its traffic, there is a great increase in the number of False Positives generated by our solution. We suspect this comes from the occurrence of many drop bursts big enough to induce these errors. Nonetheless, the amount of False Positives is relatively small when compared to the number of packets processed by the switch. These results indicate that for this experiment, WBMon has an FPR of 2.42‰.

These results indicate that our solution is able to correctly detect the drop rate of the monitored virtual connections. In most scenarios, the difference between the real and estimated drop rates was lower than 0.05%. The scenario that simulated a faulty link with a drop rate of 95% was the one in which the estimated drop rate varied the most. In this scenario, the expected drop ratio was 95.24%. The small values for the False Positive Rate obtained in this experiment help us understand why our solution was able to correctly infer the drop rates of the monitored virtual connections.

5.1.4 Convergence Time

Finally, we analyse the time required to collect enough data for the Failure Inference Algorithm to converge. Unfortunately, there was no reference number of samples required to correctly approximate link drop rates. Hence, we decided to obtain it experimentally as follows: For each drop rate y, we simulated the emission of several packets, and generated a random number that would determine whether each packet was dropped. Then we could trivially determine the estimated drop rate $\hat{y}$ as the number of dropped packets divided by the total number of packets generated. We estimate that k corresponds to the number of packets that need to be sent such that $|\hat{y} - y| < \epsilon$. For each drop rate, we repeated the experiment 100000 times to calculate the average number of required packets. With this value, we can trivially calculate the detection time by dividing the number of packets by the vconn throughput. We assumed that the average packet size is 1KB [3, 18].

Figure 5.5 shows the time that the Drop Detection Algorithm takes to correctly approximate the vconn drop ratio with an error lower than $\epsilon = 0.05\%$, for different loss rates (x axis) and throughput values (y axis). We estimated these times for both ISP and data center networks. According to the Trace Statistics for CAIDA Passive OC48 and OC192 Traces [62], ISP networks carry between 100Mbps and 5Gbps traffic, and lose from 0% to 3% of packets. However, some of the traces report up to 20% of lost packets, so we will include these values in the detection times as well. For data center networks, the literature indicates that traffic intensity varies between 10Gbps and 50Gbps [63, 64], while losing from 0.1% to 1% of its traffic [3, 4].

In Figure 5.5, the estimated times for ISP and data center networks are highlighted by the blue

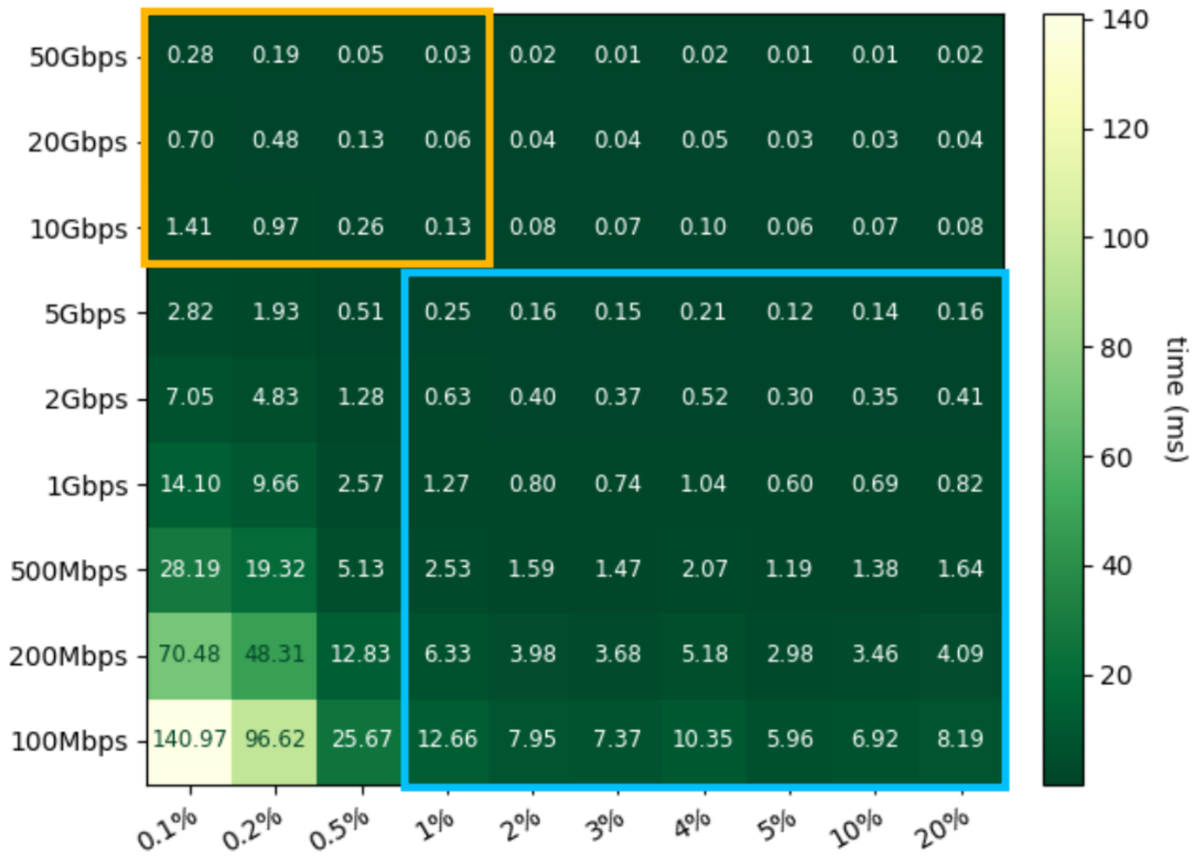


Figure 5.5: Time (ms) required to approximate each vconn drop rate for different vconn loss rates (x axis) throughput values (y axis), with $\epsilon < 0.05\%$

and orange rectangles, respectively. The results show that WBMon takes less than 13ms to detect the vconn loss ratio for ISP networks. Moreover, if the traffic throughput in ISP networks is higher than 1Gbps, it takes less than 2ms to do so. In data center networks, in turn, the detection time for WBMon is lower than 2ms. This means our solution is able to detect transient failures of at least 2ms, which is comparable to the 10ms observation slots of FlowRadar [18], and 5 orders of magnitude faster than active approaches [1, 43].

One can note that the convergence times highly depend on the network throughput. In fact, WBMon will not be able to detect vconn loss rates if there is no traffic circulating in them, which is a serious shortcoming of passive approaches. We argue that paths that carry low traffic volumes are not as critical to monitor as the ones carrying higher volumes, and thus it is not problematic taking more time to detect the drop rate of those paths.

5.2 M-Switch Placement

In this thesis, we propose a solution that allows to monitor transitional networks [50] and focuses on the possibility to incrementally add new m-switches to improve the monitoring coverage. When deploying WBMon to a legacy network, it is important to determine the number of m-switches required, as well as their optimal disposition. This is an example of the NP-Hard facility location problem [49, 65], and the solution depends on the network that will be monitored. A naive approach to solve this problem would be to compute every m-switch placement in the target network and choose the one that would minimize the number of required m-switches, while assuring link identifiability [1] in the set of monitored links. Although this method may work in smaller networks, the time it takes to find the optimal m-switch arrangement grows exponentially with the network size, making it impractical to apply to bigger networks. There are solutions in the literature that are able to tradeoff execution time and solution optimality [65]. These solutions may be the only viable option to find the best m-switch arrangement in arbitrary large networks.

In the following sections we study different m-switch arrangements on two real-world topologies, and evaluate their impact on WBMon's performance, more specifically, on the amount of faulty links successfully and falsely identified. We start by describing the followed procedure in Section 5.2.1 and then discuss the results obtained in Sections 5.2.2 and 5.2.3.

5.2.1 Simulation Setup

For each analysed topology, we select several m-switch arrangements and proceed as follows: We start by selecting a random subset of links $\mathcal{F} \subseteq \mathcal{L}$ to be faulty, and assign a success probability x_i to each, according to the loss model used in [53, 59–61]: Lossy links successfully transmit packets with a probability between 0% and 95%, and non-lossy links successfully transmit packets with a probability between 99.8% and 100%. In different executions, we varied the number of faulty links in the network from 1 to 3. We then generated the drop reports that Analysers would receive from the m-switches under the given configuration. These reports were generated by simulating the journey of 100 000 packets through each vconn, and by counting the number of packets that would be dropped. For every link l_i each packet passed through, we generated a random number k_i and, if $k_i < x_i$, we accounted that packet as dropped. This technique allowed us to generate drop reports that already include noise, as they marginally differ from the theoretical loss rate of the respective vconn. Additionally, these values are similar to the ones the failure inferrer would get in a real scenario [53]. Finally we fed the drop reports of each vconn to the Failure Inference algorithm to obtain the set of "blamed" [1] links $\mathcal{B}$. In this experiment we blamed all the links with the estimated success probability $\hat{x}_i$ lower than 99.8%, as well as all the other links that could not be distinguished from those. More formally, we have that:

$$\mathcal{B}^* = \{l_i : \hat{x}_i < 99.8\%\}, \mathcal{B} = \bigcup_{l_i \in \mathcal{B}^*} \mathcal{I}(l_i)$$

Where $\mathcal{I}(l_i)$ corresponds to the set of links that are indistinguishable from l_i (Section 4.2).

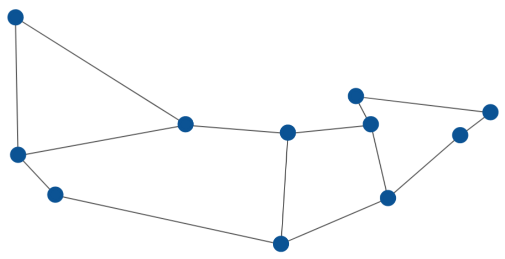
By comparing the faulty and blamed links, we were able to calculate the number of True Positives (*#TP*) produced by our solution. This value corresponds to the number of faulty links that were successfully detected, and is given by $\#(\mathcal{F} \cap \mathcal{B})$. With this value, we were able to compute the Recall and the Precision of the failure inference algorithm. The Recall corresponds to the percentage of faulty links that were successfully detected, and is obtained as $\#TP/\#\mathcal{F}$. The Precision, in turn, corresponds to the percentage of blamed links that are faulty, and is given by $\#TP/\#\mathcal{B}$. To better illustrate these metrics, let's suppose that in a network with 100 faulty links, our solution correctly identified 80 of them, but classified 10 non-lossy links as faulty (accused a total of $80 + 10 = 90$ links). Then, the recall is given by $80/100$, and the precision is $80/90$.

5.2.2 ISP Topology

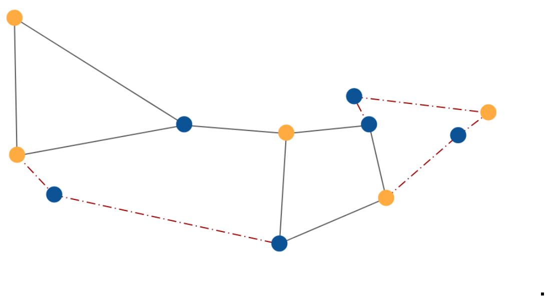
This experiment was conducted on Abilene, a real-world ISP topology extracted from the Internet Topology Zoo [66]. We started by generating several random m-switch placements, while varying the percentage of monitoring switches in the network between 40% and 80%. We generated 100 random m-switch placements for each m-switch ratio, and for each placement and faulty link ratio, we selected 100 subsets of faulty links. This allowed us to study how the amount of m-switches affects WBMon's ability to locate faulty links. Then, we repeated the same experiment with three hand-made m-switch arrangements that aimed to reduce the number of indistinguishable links, while using the minimum amount of m-switches. Figure 5.6(a) depicts the Abilene Topology, and Figures 5.6(b) to 5.6(d) illustrate the hand-made arrangements used in this experiment. r-switches are depicted by the blue circles, and m-switches by the yellow circles.

In Placement 1 (Figure 5.6(b)), we aimed to keep the m-switch ratio lower than 50%, while minimizing the number of indistinguishable links. This resulted in a configuration with 5 m-switches and an m-switch ratio of 45%, containing three pairs of indistinguishable links, represented by the red dashed lines. Placement 2 (Figure 5.6(c)) was obtained by adding the minimum number of m-switches to Placement 1 such that the network did not contain any indistinguishable link. The m-switch ratio of this placement is 64%. Placement 3 (Figure 5.6(d)), in turn, derived from Placement 1, by upgrading the r-switches that were connected to indistinguishable links. The m-switch ratio for this placement is 73%.

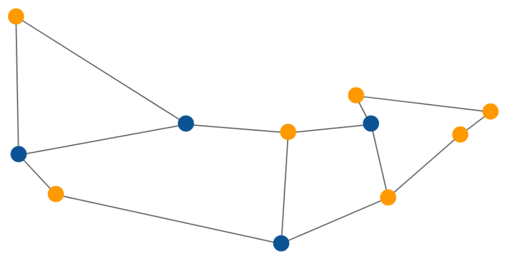
The results obtained in this experiment are summarized in Figure 5.7. The graphics are organized as a grid, where the first row contains the Recall scores for the different configurations, and the second row contains the Precision scores. The graphics on the left are relative to the random m-switch



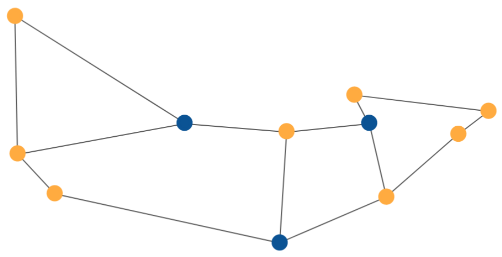
(a) Abilene topology [66]



(b) Placement 1



(c) Placement 2



(d) Placement 3

Figure 5.6: Hand-made m-switch placements for the Abilene topology

arrangements, and the graphics on the right display the results of Placement 1 (P1), Placement 2 (P2), and Placement 3 (P3). For simplicity, we will refer to the random m-switch arrangement with X% of m-switches as $R(X)$.

When we analyse the Recall scores of the random m-switch placements, we can see that our solution is able to detect 70% of the faulty links (Recall), while monitoring the network from only 40% of its switches; and 80% of the faulty links while monitoring the network from 60% of its switches. Additionally, when there is a single faulty link in the network, WBMon was able to detect it in every iteration, while using only 60% of the monitoring switches. The Recall scores consistently increase with the m-switch ratio, indicating that adding more m-switches to the network increases the number of faulty links that can be successfully detected.

One can also note that, for the same number of m-switches in the network, the solution Recall decreases with the increasing number of faulty links. This happens since the Failure Inference Algorithm tends to blame a small set of links [1]. Additionally, when it has few observation points, it is more likely to blame the wrong ones.

Despite having relatively high Recall scores, the random m-switch placements for this topology lack in

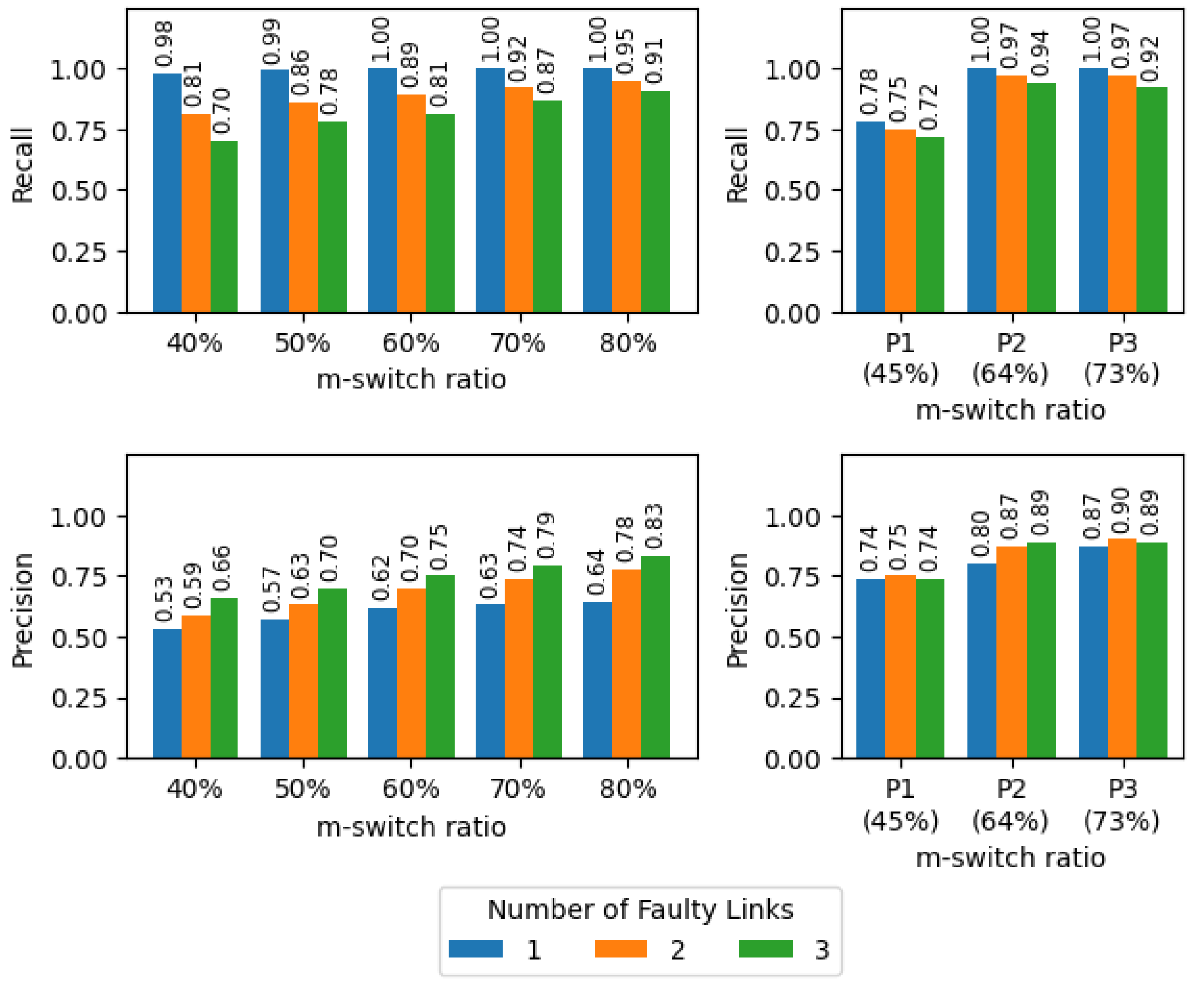


Figure 5.7: Recall and Precision of the Failure Inference Algorithm for the Abilene topology [66]. Random m-switch placements on the left, Hand-made m-switch placements on the right.

Precision, scoring values ranging from 53% to 83%. This indicates that WBMon often blames the wrong links in this topology. We hypothesize that this result comes from the existence of indistinguishable link sets (ILS) originated by the random m-switch placements. If a faulty link cannot be distinguished from other links, but is correctly identified as faulty, all its indistinguishable links will be blamed as well, which tarnishes the solution Precision. Note that if we did not blame the entire ILS, we would risk damaging the solution Recall.

We can observe a significant increase in the Recall and Precision of WBMon when using the hand-made m-switch arrangements. Namely, P2 and P3 were able to correctly identify more faulty links (higher Recall) and blamed fewer healthy links (higher Precision) than *R(80)*, while using only 64% and 73% of the switches to monitor the network. Additionally, P1 obtained higher Precision scores than R(70) in the experiments with less than three faulty links, and higher Precision scores than R(50) in the experiments with 3 faulty links. These results stress the impact of having a good m-switch placement. In fact, having optimal m-switch arrangements reduces the amount of m-switches required to monitor the network with the same Precision and Recall.

5.2.3 Data Center Topology

Clos topologies were widely used in former data center networks [1,4,43], and their well defined structure makes them a good candidate to study different m-switch placements. In this experiment, we study a 4-ary Fat Tree topology, which is a special instance of a Clos network. The switches of Fat Tree topologies are divided into the Edge, Aggregation and Core layers: The hosts connect to the Edge layer, the Edge is connected to the Aggregation layer, and the latter is connected to the Core layer (Figure 5.8). Since hosts are exclusively connected to the Edge layer, the traffic also follows a predictable pattern. For this reason, we tested this topology using two hand-made m-switch arrangements.

The first m-switch placement considered was obtained by upgrading only the switches at the Edge layer (Figure 5.9(a)), which resulted in a configuration with an m-switch ratio of 40% and no indistinguishable links. This placement corresponds to the minimal set of monitoring switches that allow to monitor the entire traffic circulating in the network. The second m-switch arrangement, in turn, consisted of updating the switches at the Edge and Core layers. This resulted in an arrangement with 60% of monitoring switches and no indistinguishable links. Note that the paths measured with the last configuration are the same as the ones monitored in NetBouncer's link identifiable probing plan [1].

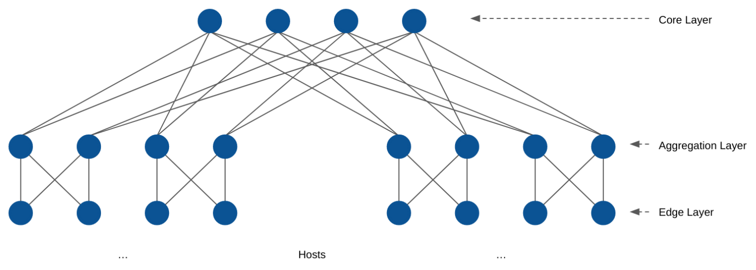
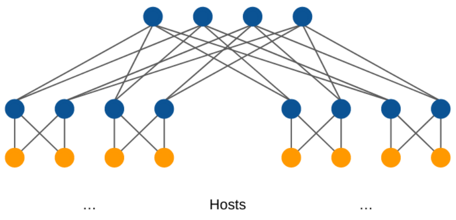


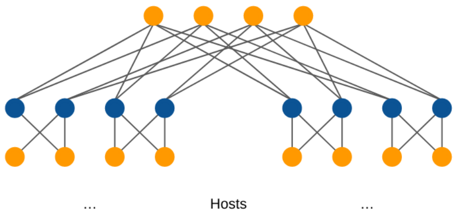
Figure 5.8: Fat Tree topology

The results shown in Figure 5.10 reveal that the Edge placement is able to successfully identify 99% of the faulty links in the network, while using only 40% of monitoring switches. However, it produces too many False Positives, which makes the Precision of that arrangement have scores of at most 51%. This suggests that the Edge arrangement is not link identifiable, and that the number of monitoring points in this configuration was not enough to correctly monitor the network.

By adding the Core switches to the monitoring set, we were able to drastically improve the solution's Precision. In fact, the second m-switch configuration was able to correctly identify 100% of the faulty links



(a) Edge placement



(b) Edge and Core placement

Figure 5.9: Different m-switch placements for a 4-ary Fat Tree Topology

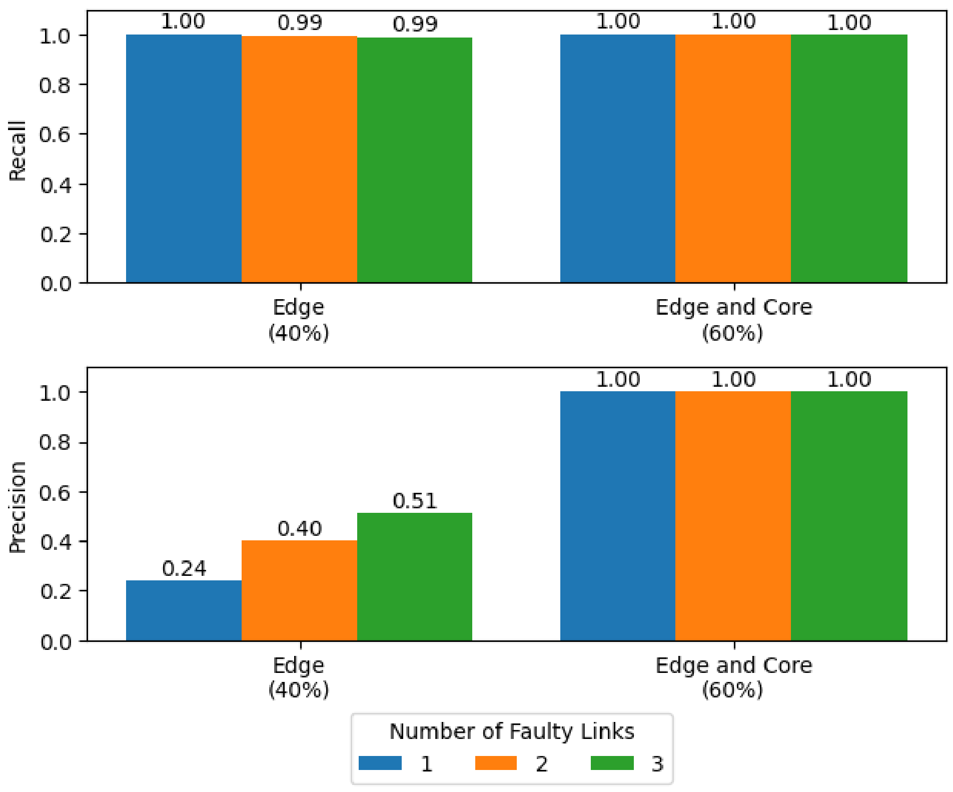


Figure 5.10: Comparing the Recall and Precision of the Edge and Edge+Core m-switch placements in the Fat Tree topology.

and did not blame any non-lossy link, while using only 60% of monitoring switches. We can understand this result as this arrangement is link identifiable [1].

This experiment demonstrates that, for certain network topologies, it is possible to reduce the number of monitoring switches without damaging the monitoring quality.

Summary

This chapter presented the evaluation of WBMon. We first evaluated the Drop Detection Algorithm limitations, by generating micro-benchmarks that allowed us to conclude to what extent it tolerates packet drop and reordering. We then tested it against realistic traces, to foresee its performance under a real world scenario. We also estimated the time required for the Drop Detection Algorithm to correctly approximate the measured paths success rates. Finally, we analysed multiple m-switch placements on different real world topologies.

6

Conclusion

Contents

6.1 Conclusions 65
6.2 Future Work 65

6.1 Conclusions

Detecting packet loss and locating its origin is a crucial task to manage and enhance the performance of computer networks. In fact, multiple solutions aim to solve this problem with countless techniques. Some of them provide fine grained metrics that allow to locate faulty devices with high precision, at the cost of requiring multiple observation points. This requirement increases the solution deployment cost, which may prevent operators from adopting it. Others aim to reduce this cost by computing coarse grained statistics that may not be enough to precisely locate the faulty devices.

In this thesis, we developed and evaluated WBMon, a passive monitoring solution that sits in the middle ground between existing solutions. On the one hand, we leverage the Data Plane programmability to perform an inter-switch coordination algorithm that allows to detect the exact number of dropped packets in individual network paths. On the other hand, by employing a correlation algorithm, our solution requires fewer observation points, which makes it cheaper to deploy in already functioning networks. Our evaluation demonstrates that the developed solution is able to correctly identify the majority of faulty links while requiring about half of the monitoring points. Additionally, WBMon's design allows to gradually increase the number of observation points to further improve the solution coverage and accuracy.

6.2 Future Work

As this work considers networks with many nodes and SDN switches, it was not possible to test our solution in a real world scenario due to lack of available resources. For future work, we intend to deploy WBMon in a real network and measure the associated traffic overhead. We also plan to implement the Controller algorithm that is responsible to find the optimal placement for the monitoring points and to configure the m-switches according to it. Finally, the current solution is based on the assumption that the paths taken by each virtual connection are known and do not change over time. We are aware that this premise may not hold in every scenario, thus we aim to extend the Failure Inference algorithm to correctly identify the faulty links without this assumption. We plan to achieve this by updating the latent factor model such that the success probability of each virtual connection corresponds to the weighted average of the success probabilities of all its paths, based on the probability with which each path is used.

Bibliography

[1] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, “Netbouncer: Active device and link failure localization in data center networks,” in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’19. Boston, Massachusetts, USA: USENIX Association, 2019, p. 599–613.

[2] D. Harrington, B. Wijnen, and R. Presuhn, “An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks,” RFC 3411, dec 2002. [Online]. Available: https://rfc-editor.org/rfc/rfc3411.txt

[3] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu, “Flow event telemetry on programmable data plane,” in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’20. Virtual Event, USA: Association for Computing Machinery, 2020, p. 76–89. [Online]. Available: https://doi.org/10.1145/3387514.3406214

[4] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, “Packet-level telemetry in large datacenter networks,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15. London, United Kingdom: Association for Computing Machinery, 2015, p. 479–491. [Online]. Available: https://doi.org/10.1145/2785956.2787483

[5] L. Peterson, C. Cascone, B. O’Connor, T. Vachuska, and B. Davie, *Software-Defined Networks: A Systems Approach*. Systems Approach, LLC, 2021.

[6] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn,” in *Proceedings of the 2013 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’13. Hong Kong, China: Association for Computing Machinery, 2013, p. 99–110. [Online]. Available: https://doi.org/10.1145/2486001.2486011

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, p. 87–95, jul 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[8] Intel, "Intelligent fabric processors," https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html, 2019.

[9] C. Hedrick *et al.*, "Routing information protocol," RFC 1058, Rutgers University, Tech. Rep., 1988.

[10] J. T. Moy, *OSPF: anatomy of an Internet routing protocol.* Addison-Wesley Professional, 1998.

[11] Y. Rekhter, T. Li, S. Hares *et al.*, "A border gateway protocol 4 (bgp-4)," 1994.

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, p. 351–364.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, p. 69–74, mar 2008. [Online]. Available: https://doi.org/10.1145/1355734.1355746

[14] I. Butun, Y. K. Tuncel, and K. Oztoprak, "Application layer packet processing using pisa switches," *Sensors*, vol. 21, p. 8010, 2021.

[15] T. P. A. W. Group, "P4runtime specification," https://opennetworking.org/wp-content/uploads/2020/10/P4Runtime-Specification-120.html, 2020.

[16] barefootnetworks, "P416 intel® tofino™ native architecture – public version," https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf, 2021.

[17] A. Morton, G. Ramachandran, S. Shalunov, L. Ciavattone, and J. Perser, "Packet Reordering Metrics," RFC 4737, Nov. 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4737

[18] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Santa Clara, California, USA: USENIX Association, 2016, p. 311–324.

[19] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *Computer*, vol. 40, pp. 103–105, 2007.

[20] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, “I know what your packet did last hop: Using packet histories to troubleshoot networks,” in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’14. Seattle, Washington, USA: USENIX Association, 2014, p. 71–85.

[21] M. Yu, L. Jose, and R. Miao, “Software defined traffic measurement with opensketch,” in *Proceedings of the 10th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI’13. Lombard, Illinois, USA: USENIX Association, 2013, pp. 29–42.

[22] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. Ramos, and A. Madeira, “Flowlens: Enabling efficient flow classification for ml-based network security applications,” in *Proceedings of the 28th Network and Distributed System Security Symposium*, ser. NDSS’21, San Diego, California, USA, 2021.

[23] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, “Heavy-hitter detection entirely in the data plane,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’17. Santa Clara, California, USA: Association for Computing Machinery, 2017, p. 164–176. [Online]. Available: https://doi.org/10.1145/3050220.3063772

[24] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, “Qpipe: Quantiles sketch fully in the data plane,” in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT ’19. Orlando, Florida: Association for Computing Machinery, 2019, p. 285–291. [Online]. Available: https://doi.org/10.1145/3359989.3365433

[25] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, “Blink: Fast connectivity recovery entirely in the data plane,” in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’19. Boston, Massachusetts, USA: USENIX Association, 2019, p. 161–176.

[26] 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 Special Interest Group on Data Communication*, ser. SIGCOMM ’14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, p. 407–418. [Online]. Available: https://doi.org/10.1145/2619239.2626310

[27] M. Wang, B. Li, and Z. Li, “sflow: towards resource-efficient and agile service federation in service overlay networks,” in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, ser. ICDCS 2004. Hachioji, Tokyo, Japan: IEEE, 2004, pp. 628–635.

[28] N. Duffield, C. Lund, and M. Thorup, “Estimating flow distributions from sampled flow statistics,” in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures,*

and Protocols for Computer Communications, ser. SIGCOMM ’03. Karlsruhe, Germany: Association for Computing Machinery, 2003, p. 325–336. [Online]. Available: https://doi.org/10.1145/863955.863992

[29] C. Estan, K. Keys, D. Moore, and G. Varghese, “Building a better netflow,” in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM ’04. Portland, Oregon, USA: Association for Computing Machinery, 2004, p. 245–256. [Online]. Available: https://doi.org/10.1145/1015467.1015495

[30] P. I. Politopoulos, E. P. Markatos, and S. Ioannidis, “Evaluation of compression of remote network monitoring data streams,” in *NOMS Workshops 2008-IEEE Network Operations and Management Symposium Workshops*, ser. NOMS 08. Salvador da Bahia, Brazil: IEEE, 2008.

[31] N. Alon, Y. Matias, and M. Szegedy, “The space complexity of approximating the frequency moments,” *Journal of Computer and system sciences*, vol. 58, no. 1, pp. 137–147, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000097915452

[32] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, “One sketch to rule them all: Rethinking network flow monitoring with univmon,” in *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’16. Florianopolis, Brazil: Association for Computing Machinery, 2016, p. 101–114. [Online]. Available: https://doi.org/10.1145/2934872.2934906

[33] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, “Nitrosketch: Robust and general sketch-based monitoring in software switches,” in *Proceedings of the 2019 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’19. Beijing, China: Association for Computing Machinery, 2019, p. 334–350. [Online]. Available: https://doi.org/10.1145/3341302.3342076

[34] G. Cormode and M. Hadjieleftheriou, “Finding frequent items in data streams,” in *Proceedings of the VLDB Endowment*, vol. 1. VLDB Endowment, 2008, p. 1530–1541. [Online]. Available: https://doi.org/10.14778/1454159.1454225

[35] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, “Counting distinct elements in a data stream,” in *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer, 2002, pp. 1–10.

[36] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, “Sketch-based change detection: Methods, evaluation, and applications,” in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’03. Miami Beach, Florida, USA: Association for Computing Machinery, 2003, p. 234–247. [Online]. Available: https://doi.org/10.1145/948205.948236

[37] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, “Data streaming algorithms for estimating entropy of network traffic,” in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’06/Performance ’06. Saint Malo, France: Association for Computing Machinery, 2006, p. 145–156. [Online]. Available: https://doi.org/10.1145/1140277.1140295

[38] V. Braverman, R. Ostrovsky, and A. Roytman, “Zero-one laws for sliding windows and universal sketches,” in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 573–590. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/5324

[39] V. Braverman and R. Ostrovsky, “Zero-one frequency laws,” in *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, ser. STOC ’10. Cambridge, Massachusetts, USA: Association for Computing Machinery, 2010, p. 281–290. [Online]. Available: https://doi.org/10.1145/1806689.1806729

[40] G. Cormode and M. Muthukrishnan, “Count-min sketch.” 2009.

[41] Y. Zhou, H. Jin, P. Liu, H. Zhang, T. Yang, and X. Li, “Accurate per-flow measurement with bloom sketch,” in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops*, ser. INFOCOM WKSHPS. Honolulu, HI, USA: IEEE, 2018, pp. 1–2.

[42] D. Goncalves, S. Signorello, F. M. Ramos, and M. Médard, “Random linear network coding on programmable switches,” in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[43] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, “Pingmesh: A large-scale system for data center network latency measurement and analysis,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15. London, United Kingdom: Association for Computing Machinery, 2015, p. 139–152. [Online]. Available: https://doi.org/10.1145/2785956.2787496

[44] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004, ch. 6.3.

[45] W. Simpson *et al.*, “Ip in ip tunneling,” RFC 1853, October, Tech. Rep., 1995.

[46] N. Kang, Z. Liu, J. Rexford, and D. Walker, “Optimizing the ”one big switch” abstraction in software-defined networks,” in *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT ’13. Santa Barbara, California,

USA: Association for Computing Machinery, 2013, p. 13–24. [Online]. Available: https://doi.org/10.1145/2535372.2535373

[47] M. Yu, "Network telemetry: Towards a top-down approach," *SIGCOMM Comput. Commun. Rev.*, vol. 49, p. 11–17, feb 2019. [Online]. Available: https://doi.org/10.1145/3314212.3314215

[48] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, p. 561–575. [Online]. Available: https://doi.org/10.1145/3230543.3230544

[49] L.-Y. Wu, X.-S. Zhang, and J.-L. Zhang, "Capacitated facility location problem with general setup cost," *Computers & Operations Research*, vol. 33, no. 5, pp. 1226–1241, 2006.

[50] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann, "Panopticon: Reaping the Benefits of incremental SDN deployment in enterprise networks," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 333–345. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/levin

[51] N. Duffield, "Network tomography of binary network performance characteristics," *IEEE Transactions on Information Theory*, vol. 52, no. 12, pp. 5373–5388, 2006.

[52] V. N. Padmanabhan, L. Qiu, and H. J. Wang, "Server-based inference of internet link lossiness," in *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, ser. INFOCOM '03, vol. 1. IEEE, 2003, pp. 145–155.

[53] D. Ghita, H. Nguyen, M. Kurant, K. Argyraki, and P. Thiran, "Netscope: Practical network loss tomography," in *2010 Proceedings IEEE INFOCOM*. IEEE, 2010, pp. 1–9.

[54] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *19th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'22. Renton, WA: USENIX Association, Apr. 2022, pp. 1345–1358. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/zeng

[55] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. Los

Angeles, CA, USA: Association for Computing Machinery, 2017, p. 15–28. [Online]. Available: https://doi.org/10.1145/3098822.3098824

[56] S. M. Ross, *Introduction to probability and statistics for engineers and scientists*. Academic press, 2020.

[57] P. Rodrigues Torres-Jr and E. Parente Ribeiro, “Packet reordering metrics to enable performance comparison in ip-networks,” *Journal of Computer Networks and Communications*, 2020.

[58] E. C. Molero, S. Vissicchio, and L. Vanbever, “Fast in-network gray failure detection for isps,” in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM ’22. Amsterdam, Netherlands: Association for Computing Machinery, 2022, p. 677–692. [Online]. Available: https://doi.org/10.1145/3544216.3544242

[59] H. X. Nguyen and P. Thiran, “Network loss inference with second order statistics of end-to-end flows,” in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’07. San Diego, California, USA: Association for Computing Machinery, 2007, p. 227–240. [Online]. Available: https://doi.org/10.1145/1298306.1298339

[60] V. N. Padmanabhan, L. Qiu, and H. J. Wang, “Server-based inference of internet performance,” in *IEEE INFOCOM*, vol. 3, 2003, pp. 1–15.

[61] H. H. Song, L. Qiu, and Y. Zhang, “Netquest: A flexible framework for large-scale network measurement,” in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’06/Performance ’06. Saint Malo, France: Association for Computing Machinery, 2006, p. 121–132. [Online]. Available: https://doi.org/10.1145/1140277.1140293

[62] C. for Applied Internet Data Analysis. The caida ucsd statistical information for the caida anonymized internet traces. [Online]. Available: https://www.caida.org/data/passive/passive_trace_statistics

[63] A. Kumar, M. Sung, J. J. Xu, and J. Wang, “Data streaming algorithms for efficient and accurate estimation of flow size distribution,” in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’04/Performance ’04. New York, NY, USA: Association for Computing Machinery, 2004, p. 177–188. [Online]. Available: https://doi.org/10.1145/1005686.1005709

[64] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo, “Burstradar: Practical real-time microburst monitoring for datacenter networks,” in *Proceedings of the 9th Asia-Pacific Workshop on Systems*,

ser. APSys ’18. Jeju Island, Republic of Korea: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3265723.3265731

[65] B. Heller, R. Sherwood, and N. McKeown, “The controller placement problem,” *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, p. 473–478, sep 2012. [Online]. Available: https://doi.org/10.1145/2377677.2377767

[66] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, “The internet topology zoo,” *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.

A

WBMon Data Plane Source Code

In this section, we present the source code developed to perform the Drop Detection Algorithm in the Data Plane. For readability purposes, we will split the source code in multiple listings.

Listing A.1 contains the header declarations for the Tofino switch. These headers declare the packet structures and are used by the switch Parsers and Deparsers to extract and reassemble the packet data. This listing contains the definition of the standard ethernet and IPV4 headers, as well as the definition of the vconn data structure. We also define the structure that will carry the Drop Reports to the solution Analysers, and an internal header that is used to transmit information from the switch Ingress to the Egress.

Listing A.1: headers.p4

```
const bit<16> ETHERTYPE_IPV4 = 0x800;

typedef bit<48> macAddr_t;
header ethernet_h {
    macAddr_t   dst_addr;
    macAddr_t   src_addr;
    bit<16>     ether_type;
}

typedef bit<32> ip4Addr_t;
header ipv4_h {
    bit<4>      version;
    bit<4>      ihl;
    bit<8>      diffserv;
    bit<16>     total_len;
    bit<16>     identification;
    bit<3>      flags;
    bit<13>     frag_offset;
    bit<8>      ttl;
    bit<8>      protocol;
    bit<16>     hdr_checksum;
    ip4Addr_t   src_addr;
    ip4Addr_t   dst_addr;
}

header ipv4_option_h {
    bit<1> copy;
    bit<2> optClass;
    bit<5> option;
    bit<8> optLen;
}

typedef bit<N_VCONN_ID_BITS> vconn_id_t;
typedef bit<32> vconn_sn_t;
typedef bit<32> vconn_slot_t;
typedef bit<8> vconn_offset_t;
header vconn_h {
    vconn_id_t id;
    vconn_sn_t sn;
}

typedef bit<32> buf_slot_t;

typedef bit<32> drop_t;
typedef bit<32> pkt_cnt_t;
struct drop_report_t {
    vconn_id_t vconn;
    drop_t drop_count;
    pkt_cnt_t packet_count;
}

header internal_h {
    buf_slot_t slot_to_clean_val;
    vconn_id_t vconn_id;
}
```

Listing A.2 contains the code executed by the switch in the Ingress. We start by declaring the packet header structure, using the definitions given in Listing A.1, and the metadata data structure. Afterwards, we define the *IngressParser*, which is where the incoming packet will be parsed, and its information extracted. The *IngressParser* starts by extracting the ethernet and IPV4 data. If the incoming packet carries the vconn information in the IP Options field, we extract that data accordingly.

We then define the *Ingress* Control, which is where the packet processing behavior is coded. It starts by declaring the Registers that will be used, and then starts defining the actions that may be executing during the packet processing.

The *set_or_mask* action is responsible for storing into metadata the value that will be used to update the buffer when registering the packet arrival. Since updating the Register value must be a quick operation, we opted to use a bit mask to reflect the change we want to apply. Instead of computing the bit mask for every processed packet, which would incur in unacceptable overhead, we pre-computed all the possible mask values at compile time and stored them in the *get_or_mask* table. The code that generates the entries for this table is shown in Listing A.6.

Then we have the *read_newest_slot* and *check_cleaned_slots* RegisterActions. These are responsible for loading into metadata the value of the newest slot, and the number of the last cleaned slot. If the incoming packet belongs to a new slot, the value of the newest slot must be atomically updated. Note that during the processing of each packet, we can only access each Register once. If there are no slots to be cleaned, the *check_cleaned_slots* RegisterAction will return an invalid slot identifier. Otherwise, it will return the identifier that will be cleaned during the processing of the current packet.

Afterwards, we import the buffer update operations. Since these methods were verbose, we kept them in a separate file, to improve code readability. This code is presented in Listing A.5.

Then, we have the RegisterAction that gets the sequence number that will be attached to the packet in the next vconn. That value is immediately incremented, to be ready to be used by the next packet.

Subsequently, we define the actions that will determine the packet's next destination. The forwarding rules present in the *ipv4_forward* table will be dynamically set by the Controller, during the deployment of WBMon. These rules state that if the packet is being sent to another vconn, then we must update the vconn identifier as well as the packet sequence number. The current implementation carries this information in the IP Options field. If the packet is being sent directly to a host, then this information must be removed from the packet, to assure that WBMon is transparent to the end-hosts.

Finally, the *apply* block contains the logic that will be followed during the packet processing. We start by reading the newest slot and getting the *or_mask*. Afterwards, we obtain the next slot that will be cleaned, and determine the Registers that hold the slots that will be written to. Since we were not allowed to use the modulo operator, we had to do it using bit slicing. This forces N to be a power of two. Then, we have the code that will clean and/or update the buffer slots. This code must be structured in this way so that the compiler is able to assign these instructions to the stage the respective Register belongs to. Since we can execute a single RegisterAction per stage, the buffer update instruction must be inside the else guard. Remember we prioritize cleaning the buffer slots, in Section 4.3. If during this process we end up cleaning a slot, we want to keep its value in order to count the number of remaining zeros and consequently detect the number of packet drops. In the target Tofino architecture, we had not enough stages to do these operations in the Ingress, hence we were forced to compute it in the Egress. However, to detect packet drops in the Egress, we must pass the value that was stored in the slot before being cleaned. The only way to carry data from the Ingress to the Egress is via internal headers. Finally, after all this processing, the Ingress Control finishes by determining where the packet is being forwarded.

Listing A.2: ingress.p4

```
struct my_ingress_headers_t {
    internal_h      internal;
    ethernet_h      ethernet;
    ipv4_h          ipv4;
    ipv4_option_h   ipv4_option;
    vconn_h         vconn;
}

struct my_ingress_metadata_t {
    vconn_slot_t current_slot;
    vconn_offset_t current_offset;

    vconn_slot_t newest_slot;

    buf_slot_t or_mask;
    vconn_slot_t last_cleanable_slot;
    bit<2>  current_slot_idx;
    vconn_slot_t slot_to_clean;
    buf_slot_t slot_to_clean_val;
    bit<2>  slot_to_clean_idx;
}

parser IngressParser(
        packet_in                      pkt,
    out my_ingress_headers_t           hdr,
    out my_ingress_metadata_t          meta,
    out ingress_intrinsic_metadata_t    intr_meta
) {
    state start {
        meta = {0, 0, 0, 0, 0, 0, 0, 0, 0};

        pkt.extract(intr_meta);
        pkt.advance(PORT_METADATA_SIZE);
        transition parse_ethernet;
    }

    state parse_ethernet {
        pkt.extract(hdr.ethernet);
        transition select(hdr.ethernet.ether_type) {
            ETHERTYPE_IPV4:  parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition select(hdr.ipv4.ihl) {
            5: accept;
            default: parse_ipv4_option;
        }
    }

    state parse_ipv4_option {
        pkt.extract(hdr.ipv4_option);
        transition select(hdr.ipv4_option.option) {
            IPV4_OPT_VCONN: parse_vconn;
            default: accept;
        }
    }

    state parse_vconn {
        pkt.extract(hdr.vconn);
        // extract slot and offset from packet sn
        meta.current_slot = (vconn_slot_t)hdr.vconn.sn[31:5];
        meta.current_offset = (vconn_offset_t)hdr.vconn.sn[4:0];
        transition accept;
    }
}


// The compiler does not accept ~0. We have to use 0
const vconn_slot_t INVALID_SLOT = 0;

const vconn_slot_t TWS = 2;
const vconn_slot_t N_SLOTS = 4;


control Ingress(
    inout my_ingress_headers_t                          hdr,
    inout my_ingress_metadata_t                         meta,
    in    ingress_intrinsic_metadata_t                  intr_meta,
    in    ingress_intrinsic_metadata_from_parser_t      prsr_md,
    inout ingress_intrinsic_metadata_for_deparser_t     dprsr_md,
    inout ingress_intrinsic_metadata_for_tm_t           tm_md)
{

    // Records the newest slot used, for each vconn
    Register<vconn_slot_t, vconn_id_t>(MAX_VCONNS, 1) newest_slot_reg;

    // Records the last cleaned slot, for each vconn
```

```
Register<vconn_slot_t, vconn_id_t>(MAX_VCONNS, 1) cleaned_slots_reg;

// Buffer is split across different stages
Register<buf_slot_t, vconn_id_t>(MAX_VCONNS, 0) slot0_reg;
Register<buf_slot_t, vconn_id_t>(MAX_VCONNS, 0) slot1_reg;
Register<buf_slot_t, vconn_id_t>(MAX_VCONNS, 0) slot2_reg;
Register<buf_slot_t, vconn_id_t>(MAX_VCONNS, 0) slot3_reg;

// Records the next sequence number to be sent to each vconn
Register<vconn_sn_t, vconn_id_t>(MAX_VCONNS, 32) next_sn_reg;

action set_ormask(buf_slot_t or_mask){
    meta.or_mask = or_mask;
}
table get_or_mask {
    key = { meta.current_offset: ternary; }
    actions = { set_ormask; }
    size = 32;
    const entries = {
        #include "gen/output/or_entries.p4"
    }
}

/*
 * Gets the newest used slot
 * (updates newest slot if current > newest)
 */
RegisterAction<vconn_slot_t, vconn_id_t, vconn_slot_t> (newest_slot_reg)
    read_newest_slot = {
        void apply(inout vconn_slot_t newest_slot, out vconn_slot_t out_value){
            if ( meta.current_slot > newest_slot ) {
                newest_slot = meta.current_slot;
            }
            out_value = newest_slot;
        }
    };
action do_read_newest_slot() {
    meta.newest_slot = read_newest_slot.execute(hdr.vconn.id);
}

RegisterAction<vconn_slot_t, vconn_id_t, vconn_slot_t>(cleaned_slots_reg)
    check_cleaned_slots = {
        void apply(inout vconn_slot_t slot_to_clean, out vconn_slot_t out_value){
            if (slot_to_clean <= meta.last_cleanable_slot) {
                out_value = slot_to_clean;
                slot_to_clean = slot_to_clean + 1;
            } else {
                out_value = INVALID_SLOT;
            }
        }
    };
action do_check_cleaned_slots() {
    meta.slot_to_clean = check_cleaned_slots.execute(hdr.vconn.id);
}

#include "buffer_ops.p4"


RegisterAction<vconn_sn_t, vconn_id_t, vconn_sn_t>(next_sn_reg)
    get_next_sn = {
        void apply(inout vconn_sn_t next_sn, out vconn_sn_t out_value) {
            out_value = next_sn;
            next_sn = next_sn + 1;
        }
    };


action do_ipv4_forward(macAddr_t dst_addr, PortId_t port) {
    tm_md.ucast_egress_port = port;
    hdr.ethernet.src_addr = hdr.ethernet.dst_addr;
    hdr.ethernet.dst_addr = dst_addr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
action do_send_to_host(macAddr_t dst_addr, PortId_t port) {
    hdr.ipv4.ihl = 5;
    hdr.ipv4_option.setInvalid();
    hdr.vconn.setInvalid();
    do_ipv4_forward(dst_addr, port);
}
action do_send_to_vconn(macAddr_t dst_addr, PortId_t port, vconn_id_t next_vconn_id) {
    hdr.ipv4.ihl = IHL;

    hdr.ipv4_option.setValid();
    hdr.ipv4_option.copy = 0;
    hdr.ipv4_option.optClass = 0;
    hdr.ipv4_option.option = IPV4_OPT_VCONN;
    hdr.ipv4_option.optLen = IPV4_OPT_LEN;

    hdr.vconn.setValid();
    hdr.vconn.id = next_vconn_id;
    hdr.vconn.sn = get_next_sn.execute(hdr.vconn.id);

    do_ipv4_forward(dst_addr, port);
}
table ipv4_forward {
    key = { hdr.ipv4.dst_addr: lpm; }
    actions = {
        do_ipv4_forward;
        do_send_to_host;
        do_send_to_vconn;
        NoAction;
    }
    size = 1024;
    default_action = NoAction;
}

apply {
    if (hdr.vconn.isValid()) {
        do_read_newest_slot();
        meta.slot_to_clean_val = 32w0;
        get_or_mask.apply();

        // this comparison avoids negative last_cleanable_slot
        // must be cast to bit<8> because of following error:
```

```
        // error: condition too complex, limit of \
        // 4 bytes + 12  bits of PHV input exceeded
        if((bit<8>)meta.newest_slot >= (bit<8>)TWS) {
            meta.last_cleanable_slot = meta.newest_slot - TWS;
        }

        do_check_cleaned_slots();

        meta.current_slot_idx  = meta.current_slot[1:0];
        meta.slot_to_clean_idx = meta.slot_to_clean[1:0];

        if ( meta.slot_to_clean != INVALID_SLOT && meta.slot_to_clean_idx == 0 ) {
            hdr.internal.setValid();
            do_clean_slot0();
        } else {
            if ( meta.current_slot_idx == 0 ) {
                do_update_slot0();
            }
        }

        if ( meta.slot_to_clean != INVALID_SLOT && meta.slot_to_clean_idx == 1 ) {
            hdr.internal.setValid();
            do_clean_slot1();
        } else {
            if ( meta.current_slot_idx == 1 ) {
                do_update_slot1();
            }
        }

        if ( meta.slot_to_clean != INVALID_SLOT && meta.slot_to_clean_idx == 2 ) {
            hdr.internal.setValid();
            do_clean_slot2();
        } else {
            if ( meta.current_slot_idx == 2 ) {
                do_update_slot2();
            }
        }

```

```
            if ( meta.slot_to_clean != INVALID_SLOT && meta.slot_to_clean_idx == 3 ) {
                hdr.internal.setValid();
                do_clean_slot3();
            } else {
                if ( meta.current_slot_idx == 3 ) {
                    do_update_slot3();
                }
            }
        }

        if(hdr.internal.isValid()) {
            hdr.internal.slot_to_clean_val = meta.slot_to_clean_val;
            hdr.internal.vconn_id = hdr.vconn.id;
        }

        ipv4_forward.apply();
    }
}


control IngressDeparser(
            packet_out                                  pkt,
    inout   my_ingress_headers_t                        hdr,
    in      my_ingress_metadata_t                       meta,
    in      ingress_intrinsic_metadata_for_deparser_t   dprsr_md
) {
    apply {
        pkt.emit(hdr);
    }
}
```

Listing A.3 presents the code executed by the switch during the Egress processing. Just like the Ingress, the Egress starts by declaring the headers that will be parsed, and the metadata that will accompany the packet during its egress processing. Note that the Egress only defines the headers it expects to receive from the Ingress. This processing phase is responsible for keeping track of the number of processed packets and the number of detected drops. Thus, it contains two Registers *dropped_packets_reg* and *total_packets_reg*, one to hold each information.

The Egress starts by counting the number of drops according to the value that was recorded in the cleaned slot. Just like when calculating the *or_mask*, we pre-computed this function and stored the output values in a table. However, each slot comprises 32 bits, which creates 2^32 possible values to store in the table, which is far beyond the switch capabilities. To overcome this issue, since we only wanted to count the number of ones in those 32 bits, we split that computation in four steps, each handling a different segment of the 32 bit string. Between each table execution, the number of detected drops was accumulated in the metadata. After performing this step, the Egress simply adds the number of detected drops to the respective register, and increments the number of processed packets. The entries of this table were automatically generated, and the generator code can be found in Listing A.7. Note that this step counts the number of ones instead of the number of zeros in the bit string. Since the compiler does not allow to initialize values by setting every bit to one, we were forced to set the default value of *slot_to_clean_val* to zero. Hence, instead of counting the number of remaining zeros, we use the bits set to one to indicate the packets that were dropped.

The current implementation omits the code that sends the drop reports to the Analysers. This information should be sent in *digests*, as they are an efficient mechanism to send messages from the Data Plane to the Control Plane. However, digests can only be sent in the Ingress Control. Thus, in order to send the drop reports, we would have to recirculate the processed packet and attach the report data. Then, during the Ingress processing, the digest would be generated and sent to the Control Plane.

Listing A.3: egress.p4

```
struct my_egress_headers_t {
    internal_h      internal;
}

struct my_egress_metadata_t {
    drop_t dropped_packets;
}

parser EgressParser(
        packet_in                   pkt,
    out my_egress_headers_t         hdr,
    out my_egress_metadata_t        meta,
    out egress_intrinsic_metadata_t eg_intr_md)
{
    state start {
        meta = { 0 };
        pkt.extract(eg_intr_md);
        transition parse_internal;
    }

    state parse_internal {
        pkt.extract(hdr.internal);
        transition accept;
    }
}

control Egress(
    /* User */
    inout my_egress_headers_t                          hdr,
    inout my_egress_metadata_t                         meta,
    /* Intrinsic */
    in    egress_intrinsic_metadata_t                  eg_intr_md,
    in    egress_intrinsic_metadata_from_parser_t      eg_prsr_md,
    inout egress_intrinsic_metadata_for_deparser_t     eg_dprsr_md,
    inout egress_intrinsic_metadata_for_output_port_t  eg_oport_md)
{

    // Records the number of detected drops
    Register<drop_t, vconn_id_t>(MAX_VCONNS, 0) dropped_packets_reg;

    // Records the number of processed packets
    Register<pkt_cnt_t, vconn_id_t>(MAX_VCONNS, 0) total_packets_reg;

    // Increments the number of detected drops in the current packet
    action count_drops(drop_t dropped_packets){
        meta.dropped_packets = meta.dropped_packets + dropped_packets;
    }

    // Each of the following tables converts a portion of slot_to_clean_val into
    // the number of drops in that same portion
    table get_dropped_packets_0 {
        key = { hdr.internal.slot_to_clean_val: ternary; }
        actions = { count_drops; }
        size = 256;
        const entries = {
            #include "gen/output/drop_count_entries_0.p4"
        }
    }
    table get_dropped_packets_1 {
        key = { hdr.internal.slot_to_clean_val: ternary; }
        actions = { count_drops; }
        size = 256;
        const entries = {
            #include "gen/output/drop_count_entries_1.p4"
        }
    }
    table get_dropped_packets_2 {
        key = { hdr.internal.slot_to_clean_val: ternary; }
        actions = { count_drops; }
        size = 256;
        const entries = {
            #include "gen/output/drop_count_entries_2.p4"
        }
    }
    table get_dropped_packets_3 {
        key = { hdr.internal.slot_to_clean_val: ternary; }
        actions = { count_drops; }
        size = 256;
        const entries = {
            #include "gen/output/drop_count_entries_3.p4"
        }
    }

    RegisterAction<drop_t, vconn_id_t, void>(dropped_packets_reg)
        update_dropped_packets = {
            void apply(inout bit<32> n_drops){
                n_drops = n_drops + meta.dropped_packets;
            }
        };
    action do_update_dropped_packets() {
        update_dropped_packets.execute(hdr.internal.vconn_id);
    }


    RegisterAction<pkt_cnt_t, vconn_id_t, void>(total_packets_reg)
        update_total_packets = {
            void apply(inout bit<32> n_packets){
                n_packets = n_packets + 1;
            }
        };

    action do_update_total_packets() {
        update_total_packets.execute(hdr.internal.vconn_id);
    }


    apply {
        get_dropped_packets_0.apply();
        get_dropped_packets_1.apply();
        get_dropped_packets_2.apply();
        get_dropped_packets_3.apply();

        do_update_dropped_packets();
        do_update_total_packets();

        hdr.internal.setInvalid();
    }
}

control EgressDeparser(
            packet_out                                  pkt,
    inout   my_egress_headers_t                         hdr,
    in      my_egress_metadata_t                        meta,
    in      egress_intrinsic_metadata_for_deparser_t    eg_dprsr_md
) {
    apply {
        pkt.emit(hdr);
    }
}
```

Listing A.4 contains the main code of the Drop Detection Algorithm.

Listing A.4: main.p4

```
#include <core.p4>
#include <tna.p4>

#include "headers.p4"
#include "ingress.p4"
#include "egress.p4"

Pipeline(
    IngressParser(),
    Ingress(),
    IngressDeparser(),
    EgressParser(),
    Egress(),
    EgressDeparser()
) pipe;

Switch(pipe) main;
```

Listing A.5 presents the RegisterActions used to operate on the buffer slots. For brevity, we will only display the methods to update a single Register. The RegisterAction *update_slot0* is responsible for applying the metadata *or_mask* to the slot value. *clean_slot0*, in turn, is responsible for resetting that value to zero, after assuring the value is saved in the metadata. Note that we store the negation of the register value, so that the Egress can count the number of dropped packets.

Listing A.5: buffer_ops.p4

```
    /*
     * Update / Clean Register 0
     */
    RegisterAction<buf_slot_t,_, void>(slot0_reg) update_slot0 = {
        void apply(inout buf_slot_t value){
            value = value | meta.or_mask;
        }
    };
    RegisterAction<buf_slot_t,_, buf_slot_t>(slot0_reg) clean_slot0 = {
        void apply(inout buf_slot_t value, out buf_slot_t out_value){
            out_value = value;
            value = 0;
        }
    };
    action do_update_slot0() {
        update_slot0.execute(hdr.vconn.id);
    }
    action do_clean_slot0() {
        meta.slot_to_clean_val = ~clean_slot0.execute(hdr.vconn.id);
    }
```

Listing A.6 contains the code that is used to generate the *or_table* entries. We define that the *or_mask* corresponds to a bit string of zeros, with a single one in the position of the respective sequence number. We can calculate this by left shifting 1 by *sn* bits. The function *ormask_table* is responsible to write the output values in P4 valid syntax.

Listing A.6: generate_or_table.p4

```
#!/usr/bin/env python3
import argparse
import sys, os
import math

from common import BUF_SIZE, N_SN_BITS

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("output_file")
    return parser.parse_args()

def ormask_fn(sn):
    return 1 << sn

def ormask_table():
    print(f"/* File automatically generated by {__file__} */\n")

    ternary_mask = (1 << int(math.log(BUF_SIZE, 2)))-1
    ternary_mask_str = f'0x{ternary_mask:0{N_SN_BITS>>2}X}'

    n = 0
    for sn in range(BUF_SIZE):
            value = ormask_fn(sn)
            value_str = f'{BUF_SIZE}w0b_{value:0{BUF_SIZE}b}'
            print(f"{sn:02} &&& {ternary_mask_str}: set_ormask({value_str});")
            n += 1
        print(f"\n/* ({n} entries) */")

if __name__ == "__main__":
    args = parse_args()
    with open(args.output_file, 'w') as f:
        sys.stdout = f
        ormask_table()
```

At last, Listing A.7 presents the code used to generate the *get_dropped_packets* entries. To count the number of ones in a bit string, we keep adding the least significant bit to a counter variable and divide the bit string by two, until the bit string equals to zero. The function *generate_table* is responsible to write the output in a P4 valid syntax.

Listing A.7: generate_drop_counter.p4

```
#!/usr/bin/env python3
import os, sys
import argparse
from common import ones, BUF_SIZE

N_DROP_TABLES = 4
n_bits_per_table = BUF_SIZE // N_DROP_TABLES

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("output_file")
    return parser.parse_args()

def count_ones(n):
    '''
    Returns the number of bits set to one in the
    binary representation of n
    '''
    res = 0
    while n > 0:
        res += n%2
        n = n//2
    return res

def generate_table(table):
    print(f"/* File automatically generated by {__file__} */\n")
    n = 0

    mask = ones(n_bits_per_table) << (n_bits_per_table * table)
    mask_str = f'0x{mask:0{BUF_SIZE//4}X}'
    for value in range(1, 1 << n_bits_per_table):
        value_str = f'{BUF_SIZE}w0b_{value<<(n_bits_per_table*table):0{BUF_SIZE}b}'
        drops = count_ones(value)
        print(f'{value_str} &&& {mask_str}: count_drops({drops});')
        n += 1
    print(f"\n/* {n} entries */")


if __name__ == "__main__":
    args = parse_args()
    for table in range(N_DROP_TABLES):
        path, extension = os.path.splitext(args.output_file)
        file = f'{path}_{table}{extension}'
        with open(file, 'w') as f:
            sys.stdout = f
            generate_table(table)
```