Packet Loss Detection in the Data Plane

Afonso Gonçalves
afonso.corte-real.goncalves@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisors: Professors Luís Rodrigues and Fernando Ramos)

Abstract. The task of monitoring the operation of computer networks is a key component to ensure the performance of current distributed systems. In particular, we are interested in the use of monitoring to detect, in a timely manner, network anomalies such as links that exhibit excessive packet loss. In this work 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 efficiency of the monitoring tasks. Inspired by the advantages and disadvantages of related work, we propose a research road-map that aims at improving the task of localizing faulty links in networks that have a mix of programmable and non-programmable switches.

Table of Contents

1 Introduction

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, and cannot be trivially applied to networks that have a combination of programmable and non-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.

The rest of the report is organized as follows: Section 2 briefly summarizes the goals and expected results of our work; In Sections 3 we present background related with our work; Section 4 analyses the state-of-the-art solutions in this topic; Section 5 describes our proposed solution; and Section 6 describes how we plan to evaluate our results; Finally, Section 7 presents the schedule of future work and Section 8 concludes the report.

2 Goals

This work addresses the problem of detecting potential sources of packet loss in a computer network. More precisely:

> *Goals:* We plan to design, implement, and evaluate a system that can help in identifying the occurrence of excessive packet loss in a network and that can provide an indication of the subset of links and switches that can be the root cause for the observed anomaly.

To achieve this goal, we plan to extend and combine complementary techniques that have been previously proposed in the literature. In particular, we aim at leveraging the potential of programmable switches to perform anomaly detection in a passive and timely manner while, at the same time, supporting this operation in networks that can have a mix of programmable and non-programmable switches.

The project will produce the following expected results.

> *Expected results:* The work will produce i) a specification of the anomaly detection mechanisms; ii) an implementation for the Intel Barefoot Tofino switch; iii) an extensive experimental evaluation using a combination of small scale real network examples and simulations; and iv) a theoretical evaluation to determine the cost and limitations of the proposed solution.

3 Background

3.1 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 [18], 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 [46, 47, 40]. 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 [34] and programmable switches [5, 4, 21], can now be leveraged to make network monitoring more accurate and efficient.

3.2 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 [19], OSPF [33], or BGP [37]. 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.

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 [25]. However, it also brings several relevant advantages. Namely, it makes the control logic simpler 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 [34].

OpenFlow [31] 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.

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* [8] impaired the development of SDN in two ways [34]: 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) [5] 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 [4] 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. [4]. 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.

4 Related Work

In this section we make an overview of the network monitoring techniques and systems that are most relevant for 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.

4.1 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. We call this extra packets *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 [47]. 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 [28]. 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 [46]. From this perspective, active monitoring can be more effective as a complementary diagnosing tool, after an anomaly is detected by some other mechanism [47].

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 [24]. 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.

4.2 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.

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.

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 [18], 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 [17, 28]; to assemble network statistics that would unveil devices overloaded with traffic [44]; or even to collect inter-packet time statistics to identify malicious activity in the network [3]. 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 application traffic suffers network anomalies [47], 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 [47]. Although there are several techniques that aim to alleviate this limitation (Section 4.3), 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.

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 [4] 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 [46, 39, 22, 20], 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.

4.3 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.*

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.

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 [36]. 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 the network performance.

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 [36]. 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 [41].

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 [47]. 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 [12, 13, 28].

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 [47]. 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 [28]. For instance, it is possible that either every single packet or no packet at all matches an established filter.

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) [17] while others employ off the shelf compression algorithms, such as LZMA, gzip or rar [35]. This technique allows to collect more network information and to consequently achieve more accurate results at the cost of consuming more computational resources.

Sketching Sketches are space-efficient probabilistic data structures used to compute accurate network statistic estimates with low memory requirements and provable resource-accuracy tradeoffs [1, 44, 30, 29]. 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 [9], heavy hitter detection [39], distinct flow counting, [2], change detection [26], entropy estimation [27], and attack detection [3].

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 structure typically stores "heavy" traffic, often losing the "mice" flows. The the coarse-grained statistics thus obtained may lose information crucial to specific fine-grained monitoring tasks, such as anomaly location or per-flow monitoring [28, 46].

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 [44]. Others employ *universal streaming* primitives, from which it is possible to calculate several metrics [7, 6, 30]. 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.

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 [28] (further analysed in Section 4.4) 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 [10] or Bloom sketches [45], 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 [14].

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 [46]. For example, NetSeer [46] 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.

4.4 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.

PingMesh [16] PingMesh 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 [11] 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 [16]).

NetBouncer [40] NetBouncer 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 [38] "*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 for 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.

Planck [36] In contrast to the previous solutions, Planck 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 [41] since it mirrors packets directly in the Data Plane. Doing it using the switch CPU significantly reduces the mirroring throughput [36]. 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.

Everflow [47] Everflow 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 [38] 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 [46]. 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.

OpenSketch [44] OpenSketch 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 [30].

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.

UnivMon [30] Univmon is a Passive, Switch Assisted solution that employs Sketching to monitor the network.

Similarly to OpenSketch [44], 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 [6, 7] 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 [23], 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 [43] and its accuracy is below desirable [42].

FlowRadar [28] FlowRadar 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.

NetSight [17] NetSight 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 is programmable 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 [47]. 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.

NetSeer [46] NetSeer 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 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.

4.5 Discussion

Table 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.

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					✓			✓

Table 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 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 [40] 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 [17], Planck [36] and Everflow [47] are able to detect and locate packet drops, these solutions incur in unacceptable traffic overhead [46], 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 [28] 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 [46] 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 a solution that relaxes these assumptions, by considering only a small set of programmable switches, connected by other non-programmable switches or even external network that may reorder, duplicate, and drop packets. To fulfill our objective, we plan to use an inter-switch drop detection mechanism similar to NetSeer, but tolerating packet reordering. In addition, we plan to introduce a host-based drop location inference algorithm, similar to the one used in NetBouncer, to help pinpoint faulty links. Our solution required regular communication with several elements, which may generate too much monitoring traffic. One challenge is thus to find a good compromise between minimizing traffic overhead while maintaining acceptable accuracy levels.

5 Proposed solution

We propose a technique to detect inter-switch packet drops, with the goal to identify the set of links and switches where losses may have occurred. It leverages the availability of programmable switches to insert observation points inside the network, but improves over state-of-the-art solutions (namely, NetBouncer and NetSeer) in two aspects. First, we can have more observation points in the network than NetBouncer, which paves the way for more accurate results. Second, we do not require all switches to be programmable, nor to be directly connected, as in NetSeer. Third, it supports passive monitoring, avoiding signaling overhead and allowing to monitor real application traffic.

5.1 Underlying Model

This work assumes the network is composed of a mix of programmable and non-programmable switches, and where only the former have monitoring capabilities. For simplicity, switches that perform monitoring activities are denoted *m-switches*, and every other switch is denoted as a regular switch, or *r-switch*. We also define *m-neighbours* as any pair of m-switches that are connected by one or more r-switches, and call each of those paths a *virtual link*. Virtual links should cover every physical link and switch in the network and should not contain loops, i.e. each switch appears at most once in a virtual links. We assume that virtual links may drop, reorder and duplicate packets. An *m-neighbourhood* is the largest subset of m-switches which virtual connections form a complete graph. This notion is useful since it allows to analyze different neighbourhoods independently, which simplifies the problem we are trying to solve.

Figure 1 presents a network composed of 5 m-switches and 4 r-switches, organized into two m-neighbourhoods: $\{1, 2, 3\}$ and $\{2, 3, 4, 5\}$. We can see that switch 5 is m-neighbour of 2, 3 and 4, but it is not neighbour of 1, since every path from 5 to 1 traverses, at least, an m-switch. We can also note that there are three virtual links connecting 2 and 3: $(2 \rightarrow 6 \rightarrow 3)$, $(2 \rightarrow 7 \rightarrow 8 \rightarrow 3)$ and $(2 \rightarrow 7 \rightarrow 9 \rightarrow 8 \rightarrow 3)$.

5.2 Overview

The proposed monitoring technique uses three main components: The *m-switches*, the *analysers*, and a *controller*, and works as follows. Every m-switch

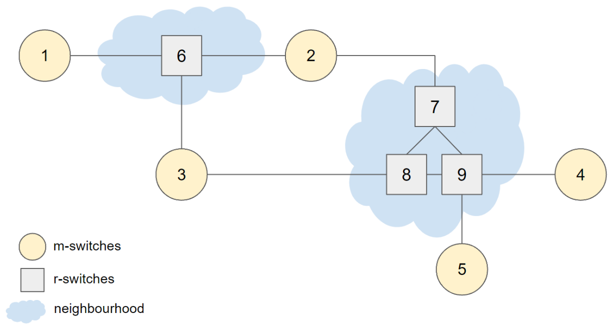

Fig. 1. Underlying model

counts the dropped packets for each incoming virtual link and regularly reports that information to an analyser. Each analyser knows the devices that form each virtual link and combines that information with the counters received from the m-switches to find the subset of devices that may be causing packet drops. The controller, in turn, is responsible for keeping the m-switches and analysers up to date with the network topology: m-switches must know which virtual links each flow uses and the analysers must know the links and switches that form each virtual link.

The ability to divide the network into multiple m-neighbourhoods allows our technique to be modular. First, instead of locating faulty devices in the entire network, the task is reduced to locating faults in individual m-neighbourhoods. Second, by assigning an analyser to each m-neighbourhood we allow the solution to scale with the network size.

5.3 Detecting Drops in Virtual Links

NetSeer [46] employs an effective technique to detect inter-switch packet drops. However, it assumes that the links connecting a pair of switches respect FIFO order; we cannot make the same assumption for virtual links. We thus plan to augment NetSeer's algorithm to tolerate packet reordering and duplication by using counters and circular buffers to track the packets that were already received. More precisely, each m-switch keeps a sequence number (*sn)* for each of its outgoing virtual links, representing the number of packets sent to the corresponding virtual link. The value of the *sn* is appended, along with the virtual link identifier, to every packet that is sent to the corresponding virtual link, regardless of the flow that packet belongs to. Each m-switch also keeps, for each incoming virtual connection, the largest *sn* received (*max_sn*) and a circular buffer (*buf*) of size N, that tracks the reception of packets $p_{max\_sn-N+1}, ..., p_{max\_sn}$. Each position i in *buf* corresponds to an expected p_i and contains a 1 if that packet has already been processed or a 0 otherwise.

Every time an m-switch receives a packet p_i with $i > max\_sn$, it updates the value of *max_sn* to i. As *max_sn* changes, say it is increased by x, the first x

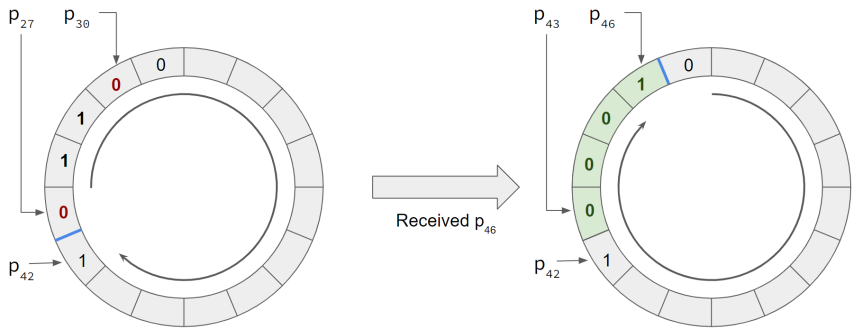

Fig. 2. Buffer update example

entries will now track the reception of packets $p_{i-x}, ..., p_i$. For this reason, the switch needs to perform two operations to these entries. First, it needs to count how many of them were still set to 0, meaning that the corresponding packets have not been received yet, and reports them as dropped packets. Second, it sets those entries to 0, to start tracking the new packets. The entry corresponding to p_i will be set to 1, since that packet was already received.

Figure 2 illustrates the procedure described above with an example with a buffer of size $N = 16$ and a *max_sn* set to 42, when receiving p_{46}. The image on the left depicts the buffer state before receiving the new packet and the blue line delimits the beginning and the end of the buffer. When receiving p_{46}, the first $46 - 42 = 4$ entries of the buffer will track p_{43} to p_{46}. We can see that the entries associated with p_{27} to p_{30} were still set to 0 (in red), meaning that those packets were not received yet, thus the m-switch will report them as dropped. Finally, the new entries (highlighted in green) were all set to 0 to track the incoming packets, except for that last one, since p_{46} was already received. The image on the right depicts the state of the buffer after these changes have been applied.

Now, consider that m-switch A sends k packets to m-switch B with *sn* p_0, p_1, ..., p_k and that B receives them in the following order: p_0, p_k, p_1, ..., p_{k-1}. When B receives p_0, it becomes the new *max_sn* and *buf* is correctly updated. If we would only store the highest *sn*, which is equivalent of having a *buf* of size $N = 1$, we would report p_1 to p_{k-1} as dropped after receiving p_k. However, if $N \geq k$, receiving p_k would not cause the buffer to override the entry of any packet that was not received yet (if $N == k$, then p_k would override the position of p_0, which was already received). On the other hand, if $N < k$, p_k would be stored in the position of p_{k-N}, meaning that packets from p_1 to p_{k-N}, inclusive, would be reported as dropped even if they arrived after p_k.

This example illustrates how the *buf* size affects this solution performance: on the one hand, having a buffer that is too small may lead to reporting dropped packets too soon, which will lead to more false positives. On the other hand, having a buffer that is too large may increase the detection latency, as it is only possible to detect the drop of p_i after receiving $p_j, j \geq i + N$. Since network traffic intensity is unpredictable, we intend to dynamically change the buffer size to one that does not cause premature drop detection nor takes too long to detect a packet drop. For instance, receiving packets that precede p_{max_sn-N} may indicate that the buffer size must be increased.

5.4 Fault Localization

This work aims to locate faulty devices by combining the drop counters received from every m-switch in a m-neighbourhood into an equation system that correlates the expected drop rate of each physical link with the measured rates in each virtual link, a technique that proved to offer good results in previous work [40]. Moreover, by having more observation points inside the network and by clustering network segments into m-neighbourhoods, we expect our technique to yield more accurate results.

5.5 Parametrization

We now discuss the following aspects: i) how to collect m-switch counters, ii) how to assure measurement consistency across different switches and iii) how often to collect those measurements.

Regarding the first issue, a strawman approach would be to make every m-switch report packet drops as soon as they are detected. Although this solution would provide a close to real time drop detection, it would incur in an unacceptable traffic overhead that could disrupt the network [46]. For this reason, several solutions [28, 44, 30, 46] accumulate statistics over a period of time (epoch) and only report those results at the end of each epoch. However, using epochs opens the possibility for different m-switches measuring different network states in the same epoch, which would induce erroneous measurements (ii). One can curtail this limitation by synchronizing the m-witches in the network and making them be in the same epoch simultaneously.

There are two main strategies to perform switch synchronization: In a time-based strategy, every switch would have a synchronized clock (e.g. by running NTP [32]) and the epochs would change at predefined time instants. At the end of each epoch, every m-switch would send its counters to the analyser and restart the measurement process. Another approach would use the analyser of each neighbourhood to determine the end of each epoch, by sending a message to each m-switch notifying the epoch change. This message would trigger a response from every m-switch, containing the measured counters of that epoch. Ideally, an epoch would end as soon as there were enough measurements to have accurate results, hence we propose a mixed solution where as soon as an m-switch collects enough information, it sends it to the analyser, which in turn contacts the other m-switches in the m-neighborhood to collect the value of their counters. This method also allows the epoch change to be initiated by the analyser, after a certain timeout.

When it comes to determine the length of an epoch (iii), we face a tradeoff. On the one hand, we want measurements to be close to real time, to detect transient anomalies quickly. On the other hand, reducing the duration of each epoch could lead to noisy results [36]. We intend to experimentally find the optimal epoch duration for a number of scenarios to be defined.

6 Evaluation

We intend to implement a prototype of the proposed solution and to evaluate its performance and limitations. This evaluation will be done in two parts. First, we will experimentally measure the performance of the drop detection and localization mechanisms. Second, we intend to theoretically determine how many m-switches are required to perform accurate detection for different network topologies.

6.1 Experimental Evaluation

We plan to evaluate the performance of our solution in four different aspects. First, we want to study how well m-switches detect packet drops, by measuring the number of undetected packet drops [1] (FN_{det}) and the number of incorrectly reported packet drops (FP_{det}). Second, we will evaluate the drop location mechanism, by measuring the number of correctly detected faulty devices (TP_{loc}), the number of healthy devices reported as faulty (FP_{loc}) and the number of undetected faulty devices (FN_{loc}). Third, we intend to calculate the traffic overhead generated by the proposed solution, by measuring the number of bytes received by the monitor. Finally, we aim to measure the time our solution takes to detect and locate packet drops.

This evaluation process will consist of several executions, for every targeted topology. Each execution will have a different configuration regarding the traffic intensity, the frequency and amplitude of packet reordering, the drop rate of each link and the number of faulty devices. Having different configurations will allow us to understand how these factors affect the performance of our solution. We will simulate a faulty link by making that link pass through a programmable switch that will be programmed to randomly drop packets, based on the hash value of the packet identifier.

During each execution, the faulty links will record, for each dropped packet, the virtual connection it belonged to, so we can compare, after each execution, the number of detected drops with the number of drops registered by the faulty links. These metrics allow to calculate the number of false negatives (FN) and the number of false positives (FP) as follows:

$$FN_{det} = \sum_{vc} max(0, real_{vc} - detected_{vc})$$

[1] Here, FN stands for number of False Negatives. Using the same logic, we have that FP stands for number of False Positives and TP stands for number of True Positives. The *det* and *loc* stand for *detection* and *location*, respectively.

$$FP_{det} = \sum_{vc} max(0, detected_{vc} - real_{vc})$$

being $detected_{vc}$ and $real_{vc}$, respectively, the number of detected and real drops for virtual connection vc.

After running the fault location algorithm, we can compare its output with the configuration of that run, to measure the TP_{loc}, FP_{loc} and FN_{loc}.

Finally, to measure the time it takes for our solution to detect a faulty device, we will set up a simple topology with no programmed faults and make the monitor send an OpenFlow message to the device we intend to make faulty, that will configure it to start dropping packets. The monitor will record the time instant that message was sent and, after receiving the data from the m-switches and detecting the faulty device, it will compute the time elapsed since the OpenFlow message was sent.

6.2 Theoretical Analysis

This work proposes a solution that allows network monitoring in networks that may have non-programmable switches in it, hence we intend to understand the minimum cost required to implement this solution in an already functioning network.

For that reason, we also aim at performing a theoretically analysis to determine, for different network topologies, i) the minimum number of m-switches required to perform fault detection and location, as a function of the network size, and ii) the maximum number of neighbourhoods that it is possible to create with a fixed number of m-switches, ideally while minimizing the variance of the neighbourhood size, i.e. to evenly divide the network into the maximum number of neighbourhoods of similar size.

7 Scheduling of Future Work

Future work is scheduled as follows:

– January 15 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
– March 30 - May 3: Perform the complete experimental evaluation of the results.
– May 4 - May 23: Write a paper describing the project.
– May 24 - June 15: Finish the writing of the dissertation.
– June 15 Deliver the MSc dissertation.

8 Conclusions

In this report we present a survey of the state of the art in network monitoring techniques, giving emphasis to approaches that can localize the source of packet drops in the network. Based on an analysis of the strengths and limitations of the different approaches, we propose a new technique to localize the source of packet drops that leverage programmable switches, but that can operate on a network that has a mix on programmable and non-programmable switches. Programmable switches are used to detect packet loss on virtual links, composed of multiple physical links connected by non-programmable switches. The information collected by different programmable switches is then correlated to pinpoint the physical link that is the source of the packet loss. The report also discusses the evaluation methodology that we plan to use to assess the merits of the proposed solution.

Acknowledgments We are very grateful to Tiago Antunes, Daniel Lopes and Daniel Seara for the fruitful discussions and comments during the preparation of this report.

References

1. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. Journal of Computer and system sciences 58(1), 137–147 (1999), `https://www.sciencedirect.com/science/article/pii/S0022000097915452`
2. Bar-Yossef, Z., Jayram, T., Kumar, R., Sivakumar, D., Trevisan, L.: Counting distinct elements in a data stream. In: International Workshop on Randomization and Approximation Techniques in Computer Science. pp. 1–10. Springer (2002)
3. Barradas, D., Santos, N., Rodrigues, L., Signorello, S., Ramos, F.M., Madeira, A.: Flowlens: Enabling efficient flow classification for ml-based network security applications. In: Proceedings of the 28th Network and Distributed System Security Symposium. NDSS'21, San Diego, California, USA (2021)
4. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D.: P4: Programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev. 44, 87–95 (jul 2014), `https://doi.org/10.1145/2656877.2656890`
5. Bosshart, P., Gibb, G., Kim, H.S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., Horowitz, M.: 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. p. 99–110. SIGCOMM '13, Association for Computing Machinery, Hong Kong, China (2013), `https://doi.org/10.1145/2486001.2486011`
6. Braverman, V., Ostrovsky, R.: Zero-one frequency laws. In: Proceedings of the Forty-Second ACM Symposium on Theory of Computing. p. 281–290. STOC '10, Association for Computing Machinery, Cambridge, Massachusetts, USA (2010), `https://doi.org/10.1145/1806689.1806729`
7. Braverman, V., Ostrovsky, R., Roytman, A.: Zero-one laws for sliding windows and universal sketches. In: Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2015). pp. 573–590. Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015), `http://drops.dagstuhl.de/opus/volltexte/2015/5324`

8. Butun, I., Tuncel, Y.K., Oztoprak, K.: Application layer packet processing using pisa switches. Sensors 21, 8010 (2021)
9. Cormode, G., Hadjieleftheriou, M.: Finding frequent items in data streams. In: Proceedings of the VLDB Endowment. vol. 1, p. 1530–1541. VLDB Endowment (2008), https://doi.org/10.14778/1454159.1454225
10. Cormode, G., Muthukrishnan, M.: Count-min sketch. (2009)
11. Dally, W.J., Towles, B.P.: Principles and practices of interconnection networks, chap. 6.3. Elsevier (2004)
12. Duffield, N., Lund, C., Thorup, M.: Estimating flow distributions from sampled flow statistics. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. p. 325–336. SIGCOMM '03, Association for Computing Machinery, Karlsruhe, Germany (2003), https://doi.org/10.1145/863955.863992
13. Estan, C., Keys, K., Moore, D., Varghese, G.: Building a better netflow. In: Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. p. 245–256. SIGCOMM '04, Association for Computing Machinery, Portland, Oregon, USA (2004), https://doi.org/10.1145/1015467.1015495
14. Goncalves, D., Signorello, S., Ramos, F.M., Médard, M.: Random linear network coding on programmable switches. In: 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS) (2019)
15. Group, T.P.A.W.: P4runtime specification. https://opennetworking.org/wp-content/uploads/2020/10/P4Runtime-Specification-120.html (2020)
16. Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., Lin, Z.W., Kurien, V.: 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. p. 139–152. SIGCOMM '15, Association for Computing Machinery, London, United Kingdom (2015), https://doi.org/10.1145/2785956.2787496
17. Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., McKeown, N.: 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. p. 71–85. NSDI'14, USENIX Association, Seattle, Washington, USA (2014)
18. Harrington, D., Wijnen, B., Presuhn, R.: An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411 (dec 2002), https://rfc-editor.org/rfc/rfc3411.txt
19. Hedrick, C., et al.: Routing information protocol. Tech. rep., RFC 1058, Rutgers University (1988)
20. Holterbach, T., Molero, E.C., Apostolaki, M., Dainotti, A., Vissicchio, S., Vanbever, L.: Blink: Fast connectivity recovery entirely in the data plane. In: Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation. p. 161–176. NSDI'19, USENIX Association, Boston, Massachusetts, USA (2019)
21. Intel: Intelligent fabric processors. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html (2019)
22. Ivkin, N., Yu, Z., Braverman, V., Jin, X.: Qpipe: Quantiles sketch fully in the data plane. In: Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies. p. 285–291. CoNEXT '19, Association for Computing Machinery, Orlando, Florida (2019), https://doi.org/10.1145/3359989.3365433

23. Kang, N., Liu, Z., Rexford, J., Walker, D.: Optimizing the "one big switch" abstraction in software-defined networks. In: Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies. p. 13–24. CoNEXT '13, Association for Computing Machinery, Santa Barbara, California, USA (2013), https://doi.org/10.1145/2535372.2535373
24. Kohavi, R., Longbotham, R.: Online experiments: Lessons learned. Computer 40, 103–105 (2007)
25. Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., Shenker, S.: Onix: A distributed control platform for large-scale production networks. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. p. 351–364. OSDI'10, USENIX Association, Vancouver, BC, Canada (2010)
26. Krishnamurthy, B., Sen, S., Zhang, Y., Chen, Y.: Sketch-based change detection: Methods, evaluation, and applications. In: Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement. p. 234–247. IMC '03, Association for Computing Machinery, Miami Beach, Florida, USA (2003), https://doi.org/10.1145/948205.948236
27. Lall, A., Sekar, V., Ogihara, M., Xu, J., Zhang, H.: Data streaming algorithms for estimating entropy of network traffic. In: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems. p. 145–156. SIGMETRICS '06/Performance '06, Association for Computing Machinery, Saint Malo, France (2006), https://doi.org/10.1145/1140277.1140295
28. Li, Y., Miao, R., Kim, C., Yu, M.: Flowradar: A better netflow for data centers. In: Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation. p. 311–324. NSDI'16, USENIX Association, Santa Clara, California, USA (2016)
29. Liu, Z., Ben-Basat, R., Einziger, G., Kassner, Y., Braverman, V., Friedman, R., Sekar, V.: Nitrosketch: Robust and general sketch-based monitoring in software switches. In: Proceedings of the 2019 ACM Conference on Special Interest Group on Data Communication. p. 334–350. SIGCOMM '19, Association for Computing Machinery, Beijing, China (2019), https://doi.org/10.1145/3341302.3342076
30. Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., Braverman, V.: 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. p. 101–114. SIGCOMM '16, Association for Computing Machinery, Florianopolis, Brazil (2016), https://doi.org/10.1145/2934872.2934906
31. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. 38, 69–74 (mar 2008), https://doi.org/10.1145/1355734.1355746
32. Mills, D.L.: Internet time synchronization: the network time protocol. IEEE Transactions on communications 39, 1482–1493 (1991)
33. Moy, J.T.: OSPF: anatomy of an Internet routing protocol. Addison-Wesley Professional (1998)
34. Peterson, L., Cascone, C., O'Connor, B., Vachuska, T., Davie, B.: Software-Defined Networks: A Systems Approach. Systems Approach, LLC (2021)
35. Politopoulos, P.I., Markatos, E.P., Ioannidis, S.: Evaluation of compression of remote network monitoring data streams. In: NOMS Workshops 2008-IEEE Network Operations and Management Symposium Workshops. NOMS 08, IEEE, Salvador da Bahia, Brazil (2008)

36. Rasley, J., Stephens, B., Dixon, C., Rozner, E., Felter, W., Agarwal, K., Carter, J., Fonseca, R.: Planck: Millisecond-scale monitoring and control for commodity networks. In: Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication. p. 407–418. SIGCOMM '14, Association for Computing Machinery, Chicago, Illinois, USA (2014), `https://doi.org/10.1145/2619239.2626310`
37. Rekhter, Y., Li, T., Hares, S., et al.: A border gateway protocol 4 (bgp-4) (1994)
38. Simpson, W., et al.: Ip in ip tunneling. Tech. rep., RFC 1853, October (1995)
39. Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., Rexford, J.: Heavy-hitter detection entirely in the data plane. In: Proceedings of the Symposium on SDN Research. p. 164–176. SOSR '17, Association for Computing Machinery, Santa Clara, California, USA (2017), `https://doi.org/10.1145/3050220.3063772`
40. Tan, C., Jin, Z., Guo, C., Zhang, T., Wu, H., Deng, K., Bi, D., Xiang, D.: Netbouncer: Active device and link failure localization in data center networks. In: Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation. p. 599–613. NSDI'19, USENIX Association, Boston, Massachusetts, USA (2019)
41. Wang, M., Li, B., Li, Z.: sflow: towards resource-efficient and agile service federation in service overlay networks. In: 24th International Conference on Distributed Computing Systems, 2004. Proceedings. pp. 628–635. ICDCS 2004, IEEE, Hachioji, Tokyo, Japan (2004)
42. Yang, T., Jiang, J., Liu, P., Huang, Q., Gong, J., Zhou, Y., Miao, R., Li, X., Uhlig, S.: Elastic sketch: Adaptive and fast network-wide measurements. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. p. 561–575. SIGCOMM '18, Association for Computing Machinery, Budapest, Hungary (2018), `https://doi.org/10.1145/3230543.3230544`
43. Yu, M.: Network telemetry: Towards a top-down approach. SIGCOMM Comput. Commun. Rev. 49, 11–17 (feb 2019), `https://doi.org/10.1145/3314212.3314215`
44. Yu, M., Jose, L., Miao, R.: Software defined traffic measurement with opensketch. In: Proceedings of the 10th Usenix Conference on Networked Systems Design and Implementation. pp. 29–42. NSDI'13, USENIX Association, Lombard, Illinois, USA (2013)
45. Zhou, Y., Jin, H., Liu, P., Zhang, H., Yang, T., Li, X.: Accurate per-flow measurement with bloom sketch. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops. pp. 1–2. INFOCOM WKSHPS, IEEE, Honolulu, HI, USA (2018)
46. Zhou, Y., Sun, C., Liu, H.H., Miao, R., Bai, S., Li, B., Zheng, Z., Zhu, L., Shen, Z., Xi, Y., Zhang, P., Cai, D., Zhang, M., Xu, M.: 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. p. 76–89. SIGCOMM '20, Association for Computing Machinery, Virtual Event, USA (2020), `https://doi.org/10.1145/3387514.3406214`
47. Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B.Y., Zheng, H.: Packet-level telemetry in large datacenter networks. In: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication. p. 479–491. SIGCOMM '15, Association for Computing Machinery, London, United Kingdom (2015), `https://doi.org/10.1145/2785956.2787483`