EDGAR COSTA MOLERO

IMPROVING NETWORK FAILURE DETECTION AND RECOVERY WITH PROGRAMMABLE DATA PLANES

DISS. ETH. NO. 30252

DISS. ETH NO. 30252

IMPROVING NETWORK FAILURE DETECTION AND RECOVERY WITH PROGRAMMABLE DATA PLANES

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES
(Dr. sc. ETH Zurich)

presented by

EDGAR COSTA MOLERO
MSc ETH EEIT
ETH Zurich

born on 02.08.1992

accepted on the recommendation of

Prof. Dr. Laurent Vanbever
Dr. Stefano Vissicchio
Prof. Dr. Minlan Yu

2024

Edgar Costa Molero: *Improving network failure detection and recovery with programmable data planes,* © 2024

Diss. ETH No. 30252
TIK-Schriftenreihe-Nr. 212

ABSTRACT

Since its creation, the Internet has grown exponentially in size and use cases, becoming an integral part of our society. Its seamless operation is often taken for granted; we only recognize its importance when disruptions occur. The current Internet's complexity and scale make it prone to all sorts of failures, with each minute of downtime costing companies millions of dollars and damaging their reputation.

In this thesis, we address the critical need for rapid detection and recovery mechanisms for network failures. We expand beyond conventional hard failures to explore and address the issue of gray failures in ISP networks, a subtle and poorly understood issue for which operators lack effective solutions. By leveraging advances in programmable data planes, we develop two systems to detect, localize, and recover from network failures.

First, we introduce FANcY, a novel system to detect and localize gray failures in ISP networks. FANcY utilizes programmable switches to implement a reliable synchronization and counting protocol, enabling precise packet loss detection. FANcY adapts to the limited memory capacity of modern switches with a hybrid approach: dedicated counters for high-priority traffic and a probabilistic data structure for best-effort traffic. This design ensures efficient monitoring under various conditions and future-proofs the system against constantly increasing traffic volumes. We demonstrate FANcY's capability for sub-second gray failure detection and reaction through extensive simulations and a prototype running on Intel Tofino switches.

Second, we present our work on hardware-accelerated network control planes. This research extends beyond detection, demonstrating that programmable data planes can run critical control plane functions traditionally implemented in software. Our working prototype efficiently runs diverse such tasks in the data plane including: detecting hard, gray, and remote failures, notifying other devices, executing distributed path-vector computations that adhere to shortest-path and BGP-like policies, and rapidly updating forwarding states to restore connectivity after failures. Finally, our work identifies challenges in expressiveness and scalability for programmable data planes, emphasizing that the careful selection of tasks for offloading remains a critical area for future research.

ZUSAMMENFASSUNG

Seit seiner Entstehung ist das Internet in seiner Größe und seinen Anwendungsbereichen exponentiell gewachsen und zu einem integralen Bestandteil unserer Gesellschaft geworden. Sein reibungsloser Betrieb wird oft als selbstverständlich angesehen; wir erkennen seine Bedeutung erst bei Störungen. Die derzeitige Komplexität und Größe des Internets machen es anfällig für Störungen und Ausfälle, dabei kostet jede Minute Unterbruch Unternehmen Millionen und schadet deren Ruf.

In dieser Arbeit nehmen wir uns schneller Erkennungs- und Wiederherstellungsmechanismen für Netzwerkausfälle an. Dabei widmen wir uns nicht nur den kompletten Ausfällen, sondern auch den partiellen Ausfällen, den sogenannten Gray Failures, ein subtiles und wenig verstandenes Problem, für das Netzwerk-Betreibern effektive Lösungen fehlen. Wir entwickeln zwei Systeme zur Erkennung, Lokalisierung und Wiederherstellung von Netzwerkausfällen durch Nutzung der aktuellsten Entwicklungen in programmierbaren Netzwerk-Datenebenen.

Zunächst stellen wir FANcY vor, ein neuartiges System zur Erkennung und Lokalisierung partieller Ausfälle in ISP-Netzwerken. FANcY nutzt programmierbare Switches, um ein zuverlässiges Synchronisations- und Zählprotokoll zu implementieren, das eine präzise Erkennung von Datenpaketverlusten ermöglicht. FANcY passt sich mit einem hybriden Ansatz an die begrenzte Speicherkapazität moderner Switches an: dedizierte Zähler für priorisierten Datenverkehr und eine probabilistische Datenstruktur für Best-Effort-Datenverkehr. Dieses Design gewährleistet eine effiziente Überwachung unter verschiedenen Bedingungen und sichert das System gegen ständig steigende Datenvolumen ab. Durch umfangreiche Simulationen und einem Prototypen auf Intel Tofino Switches zeigen wir, dass FANcY partielle Ausfälle im Subsekundenbereich erkennen und darauf reagieren kann.

Dann präsentieren wir unsere Arbeit zu hardwarebeschleunigten Kontrollebenen für Netzwerke. Diese Arbeit geht über die Erkennung hinaus und zeigt, dass programmierbare Datenebenen in der Lage sind, kritische Funktionen der Kontrollebene zu implementieren, die traditionell in Software implementiert wurden. Unser Prototyp setzt verschiedene Aufgaben in der Datenebene effizient um, einschließlich der Erkennung verschiedener Arten von Ausfällen (komplett, partiell und remote), der Benachrichtigung anderer Geräte, der Berechnung von neuen Pfaden, basierend auf verteilten Pfad-Vektor-Algorithmen, welche sich an kürzeste Pfad- und BGP-ähnliche Richtlinien halten, sowie der schnellen Reparatur der Datenebene zur Wiederherstellung der Verbindungen nach Ausfällen. Schließlich zeigt unsere Arbeit Herausforderungen in Bezug auf Funktionalität und Skalierbarkeit auf und betont, dass die sorgfältige Auswahl der auszulagernden Aufgaben ein wichtiger Bereich für zukünftige Forschung bleibt.

PUBLICATIONS

This dissertation is based on previously published conference proceedings. The list of accepted and submitted publications is presented hereafter.

Hardware-Accelerated Network Control Planes

Edgar Costa Molero, Stefano Vissicchio,
Laurent Vanbever.
In *ACM HotNets*, Redmon, WA, USA, 2018.

FAst In-Network GraY Failure Detection for ISPs

Edgar Costa Molero, Stefano Vissicchio,
Laurent Vanbever.
In *ACM SIGCOMM*, Amsterdam, Netherlands, 2022.

The following publications were part of my PhD research and are referenced in this thesis, but they were led by other researchers.

Blink: Fast Connectivity Recovery Entirely in the Data Plane

Thomas Holterbach, Edgar Costa Molero,
Maria Apostolaki, Alberto Dianotti,
Stefano Vissicchio, Laurent Vanbever.
In *USENIX NSDI*, Boston, MA, USA, 2019.

Canary: Congestion-Aware In-Network Allreduce Using Dynamic Trees

Daniele De Sensi, Edgar Costa Molero,
Salvatore Di Girolamo, Laurent Vanbever,
Torsten Hoefler.
In *FGCS*, 2024.

ACKNOWLEDGMENTS

This dissertation marks the end of my research journey, which would not have been possible without the support, guidance, and encouragement of my advisors, colleagues, friends, and family. I am deeply grateful to everyone who has participated in this journey. I want to express my sincere appreciation to those who have helped me make this achievement possible.

First and foremost, I would like to thank Prof. Laurent Vanbever for allowing me to join the NSG group. I truly appreciate his guidance and supervision during my master's thesis, which made me fall in love with SDN, and later on, for his guidance during my doctoral studies. He has taught me everything I know about creating outstanding research with a significant emphasis on communication and presentation skills. I would also like to thank him for all the support and understanding he has had for me during the end of my PhD while I had to deal with health issues. Thanks to his unconditional support, I was able to finish this dissertation.

Second, I sincerely thank Dr. Stefano Vissicchio, my second advisor. I feel incredibly privileged and lucky to have worked with him from almost the beginning of my thesis. Discussing low-level and high-level aspects of my research with Stefano has been a beneficial and enriching experience.

I would also like to thank Prof. Minlan Yu for being part of my dissertation committee, reading the thesis, and providing feedback.

I want to thank my colleagues from the networked systems group (NSG), who have been incredibly supportive since the beginning and throughout the entire duration of the thesis, creating a delightful working environment: Ahmed El-Hassany, Maria Apostolaki, Thomas Holterbach, Rüdiger Birkner, Tobias Bühler, Roland Meier, Alexander Dietmüller, Albert Gran Alcoz, Coralie Busse-Grawitz, Rui Yang, Ege Cem Kirci, Romain Jacob, Roland Schmid, Tibor Schneider, Yu Chen, Georgia Fragkouli, Muoi Tran, Theo von Arx, Laurin Brandner, Luckas Röllin, Valerio Torsiello. I am particularly grateful to Thomas Holterbach, my semester and master thesis supervisor and later collaborator, who was not only a great mentor but also a wonderful travel companion during our trips to the US. I extend my gratitude to Rüdiger Birkner, my office mate for most of my PhD. Rüdiger has inspired and supported me in many ways, both during our time sharing office and

later on. Without him my time here would have been more challenging; I owe you a lot. I would also like to thank Tobias Bühler, Roland Meier, and Georgia Fragkouli, for all the conversations, help, and time we shared in our office. Finally, I would like to thank again Rüdiger, Georgia, and Coralie for all the feedback I got on the thesis.

Additionally, I would like to acknowledge some external people with whom I have worked or who have helped me in one way or another. First, I would like to thank Daniele de Sensi for the time we worked together; it was a pleasure. Second, I would like to thank Vladimir Gurevich from Intel, who always (even during weekends and holidays) helped me with questions regarding Tofinos and P4. I also want to thank the TIK system admins: Stefan Schindler, Edoardo Talotti, and Niklaus Kappeler, for their patience and being so fast and reliable with any problem I had.

Of course, I am thankful to Beat Futterknecht, our department administrator, who has been extremely helpful during these years. Beat has been of tremendous help with any administrative task within the university, cantonal offices, or even finding an affordable flat. I enjoyed our random endless conversations; they were an excellent way to disconnect.

I am also profoundly grateful to my friends from back home and Zurich, who have been essential during this challenging journey. Ferran, Nil, Adrià, Gabi, Giulia, Nathalie, Margarita, Carlo, Marcos, Alvaro, Laia, George, and John, thank you for being there during my lows, for the precious moments we spent together that allowed me to step away from work, and for understanding whenever I had to cancel plans or couldn't meet up due to my commitments. Your support and friendship have been invaluable.

Finally, I am endlessly grateful to my family for their unwavering love and support. I sincerely appreciate their understanding and acceptance of my infrequent visits, constant busyness, and absence from family events and holidays. Your love and support are incredibly meaningful to me.

Edgar Costa Molero
May 2024

CONTENTS

1 INTRODUCTION

The Internet is one of history's most influential inventions, rapidly becoming essential to humans. Like the telegraph, telephone, radio, and computer, the Internet has completely reshaped how we communicate, entertain, and access information.

The origins of the Internet can be traced back to the early 1960s. During the Cold War, the US feared the Soviet Union could attack the national telephone system, the primary long-distance communication system at that time. This vulnerability highlighted the need for a more robust form of communication. In response, in 1962, scientists proposed a network to interconnect the different government research computers nationwide. Circuit switching, the prevailing communication method at the time, was slow, prone to data loss, and vulnerable to attacks. To address that, in 1965, researchers at MIT developed packet switching, a method in which transmitted data is broken into small blocks (packets), each of which may take an independent route toward its destination, making it robust to transmission interruptions. This innovation led to the formation of the ARPANET in 1969 by the US Advanced Research Projects Agency (ARPA) [5], the first network to feature node-to-node messaging between computers. In the following years, more packet-switching networks emerged; however, these networks could not communicate without a universal protocol. To solve this issue, in 1974, a common data transmission protocol and addressing scheme was introduced, marking the birth of the TCP/IP protocol suite [6], laying the foundation for the modern Internet.

The rest is history. Grounded in packet switching and TCP/IP, in the 1980s and 1990s, the Internet rapidly transitioned from an academic and military network to the general public, in part thanks to the creation of the World Wide Web (WWW) in 1989, the introduction of web browsers, and the commercialization of Internet access by ISPs [5]. Today, the Internet connects an estimated 5.3 billion users and over 29.3 billion IP devices [7], becoming present in every aspect of our daily lives, from communication, accessing news, shopping, entertainment, and work.

Given the Internet's integral role in everyday life, maintaining its infrastructure and ensuring it is always functional is critical. Its scale, complexity, and combination of various technologies from software to hardware across multiple layers, make the Internet inherently prone to all sorts of failures.

Major incidents, such as service outages or security breaches, can have an extensive impact, from affecting the customers of an entire ISP [8] to causing global disruptions in services. A recent example was the 2021 Facebook outage, in which all its services (Facebook, Instagram, WhatsApp, etc) became globally unavailable for seven hours, affecting every user [9].

While major outages are often the result of human errors (i.e., misconfigurations), they are not the only cause. As the Internet grows in scale and complexity, the reliability of its infrastructure, comprised of routers, switches, and other hardware, plays an increasingly significant role in network failures.

We often associate network failures with the so-called "hard failures". These failures typically occur when a networking device's port, the connecting link to another device, or the device itself completely fails. These failures immediately affect all the traffic, making them notorious and, therefore, relatively easy to detect by existing techniques.

In large-scale networks, however, components often fail or malfunction in unpredictable ways affecting only a subset of the traffic. These subtle failures are known in the literature as "gray failures" [10]. Gray failures are much harder to detect than "hard failures." For example, they might affect specific types of traffic, such as traffic for certain destination addresses or specific protocols, or cause random drops on a fraction (i.e., <1%) of the traffic that crosses a link [11]. Gray failures commonly result from malfunctioning hardware, software bugs or device misconfigurations.

Since gray failures can affect any random subset of the traffic at any time, detecting them requires analyzing *all* the traffic, *all* the time. Doing so is unsurprisingly difficult and goes beyond the capabilities of traditional detection mechanisms, which, to scale, rely on periodically analyzing small traffic subsets using probing [11], packet counters [12], random sampling [13], or traffic mirroring [14, 15]. Without effective detection tools, gray failures often go unnoticed by network operators until customers raise concerns, leading to service degradation for long periods. Indeed, even minor traffic losses can significantly impact the quality of Internet services and user experience [16].

Recent collaborations between researchers and industry have resulted in the development of several gray failure detectors for data centers and cloud networks [11, 17]. While these solutions are effective, their deployment necessitates end-host control or very low inter-device latencies ($\mathcal{O}(\mu s)$) to support monitoring at high bandwidths. These requirements limit their applicability in wider geographically spread networks such as ISPs and WANs. For example, in ISP networks, where inter-device latencies are typically in the order of milliseconds and link capacities can reach 100 Gbps or more, solutions proposed for data centers would require an amount of memory unavailable in today's network devices. Furthermore, as network traffic demands increase and link capacities expand to accommodate those demands, these limitations become further exacerbated. This scenario calls for the development of solutions that are not only scalable today but will remain effective in the face of ever-growing traffic demands in the future.

This dissertation addresses the critical need for practical gray failure solutions in ISP networks. Our work includes a comprehensive analysis of the current state of gray failures in ISP networks and the development of a scalable solution designed to detect, localize, and recover from gray failures in high-speed and high-delay environments, anticipating future ever-increasing traffic demands.

Recent research and industry efforts have led to the development of programmable data plane switches [18]. These switches, programmable with a domain-specific language, enable custom forwarding logic and the execution of stateful operations on billions of packets per second [19]. This technology allows network operators to implement stateful algorithms directly within the network, eliminating the need for continuous external packet analysis, a major bottleneck.

In this dissertation, we leverage the capabilities of programmable data planes to process and execute simple logic on *all* the packets that cross a network device. We demonstrate that even with limited state and compute, it is possible to run detection and recovery algorithms in-network, for general networks, and without compromising line-rate performance.

In summary, this dissertation focuses on the following research question:

> *Are gray failures a prevalent issue in ISP networks, and can programable data planes help us improve current failure detection and recovery mechanisms in high-speed and high-delay networks such as ISPs?*

This dissertation answers this research question with the following work:

A comprehensive study of gray failures in ISP networks that underscores the need for dedicated fast gray failure detection and recovery systems in ISP networks. This study includes an exhaustive analysis of hardware bug reports from the two leading networking device vendors for ISPs, revealing over a hundred bugs leading to different types of gray failures. Additionally, it includes a survey among ISP operators, which indicates that gray failures are a prevalent issue in ISP networks, that operators are concerned about them, and that most operators lack practical detection tools.

FANcY [2], a gray failure detection and localization system designed for current and future high-bandwidth and high-delay networks, such as ISPs. FANcY leverages programmable switches to implement a novel synchronization and counting protocol between directly connected programmable data planes enabling switches to precisely compare counters and detect packet losses at line rate.

Hardware-accelerated network control planes [1], a control plane implementation in the data plane, that demonstrates how programmable data planes can accelerate network response to failures. This work shows how the data plane can run a modified distance vector routing algorithm, enabling rapid forwarding state updates after detecting a failure.

Dissertation outline. This dissertation is structured as follows.

In Chapter 2, we introduce the reader to some basic background in computer networks, their evolution, and provide an introduction to programmable data planes.

In Chapter 3, we dive into network failures, gray failures, and existing detection techniques. Furthermore, in this chapter, we analyze the results from our bug study and the operator survey.

In Chapter 4, we present FANcY, our fast in-network gray failure detection and localization system, which detects gray (and hard) failures within the data plane using programmable switches.

Next, in Chapter 5, we challenge the traditional split between software-based control planes and hardware-based data planes and explore the benefits of offloading key convergence tasks to programmable data planes.

Finally, in Chapter 6, we conclude the dissertation and summarize some possible future research directions.

2 BACKGROUND

This chapter introduces the reader to important and useful concepts used throughout the dissertation. First, in Section 2.1, we present foundational networking concepts that are instrumental in the operations of every network. These include routing algorithms and protocols, as well as the different network planes. Then, in Section 2.2, we briefly define the different types of networks (i.e., Wide Area Network (WAN), Internet Service Provider (ISP), data center, etc.). Afterward, in Section 2.3, we describe the evolution of networks over recent years. Finally, in Section 2.4 we give an introduction to programmable data planes as well as the state-of-the-art hardware that enable fully programmable networks.

2.1 NETWORKING FOUNDATIONS

In this section, we provide an overview of fundamental networking concepts that are later utilized in this thesis. First, we explain what a routing protocol is and outline the main types. Subsequently, we briefly outline the network planes' abstractions, including their current functionalities, aspects that we challenge in this thesis. Finally, we describe the two predominant transport protocols, UDP and TCP, highlighting the importance of understanding their dynamics under different network conditions and the implications for designing data-driven in-network applications.

2.1.1 *Network routing*

Network routing is one of the core components of any communication network. In routing, network devices, such as routers, determine the best communication paths by utilizing different routing algorithms. To learn the best path, routers continuously exchange network state information with other network participants, and given a defined objective and algorithm, routers compute the best path to send packets to destinations. There are two main sub-families of routing protocols: distance-vector and link-state.

Distance-Vector Protocols. In distance-vector protocols, routers periodically advertise a vector with the distance to known destinations to their neighbors and run the Bellman-Ford algorithm [20] to compute the best route. In the beginning, routers only know the distance to direct neighbors. Whenever a router receives a distance vector from a neighbor, it compares it with its routing table, and if any destination can be reached at a lower cost, it updates its routing table and sends a new vector to its neighbors. The process is repeated until routers converge and learn the route to all destinations. The most common distance-vector protocols include RIP [21] and IGRP [22].

Link-State Protocols. In link-state protocols, routers first reconstruct a global view of the network topology and then locally run the Dijkstra algorithm [23] on it to compute the shortest path to every destination. To construct the global view, in link-state protocols, each router broadcasts Link-State Advertisements (LSAs) messages that contain its local view. Link status changes trigger a network-wide Link State Advertisements (LSAs) broadcast. The most utilized link-state protocols are OSPF [24] and IS-IS [25].

Distance-vector protocols, while straightforward to configure and efficient in small networks, converge slowly (i.e., it takes long until all routers achieve a consistent view of the network), especially in large networks. Additionally, they may encounter the *count-to-infinity* [26] problem, where a network failure can cause routing loops. Path-vector protocols emerged as a solution. While being similar to distance-vector protocols, they include the entire path in the routing advertisements and not just the path's length, thus avoiding loops. The Border Gateway Protocol (BGP) [27] is a notable example of a path-vector protocol. Meanwhile, link-state protocols converge quickly, and incur less overhead (i.e., messages are only broadcasted when needed), but demand more computing and storage resources than distance/path-vector protocols.

2.1.2 *Network planes*

In computer networks, specifically in IP router networks, the architectural framework is partitioned into three operational planes: the control, the data,

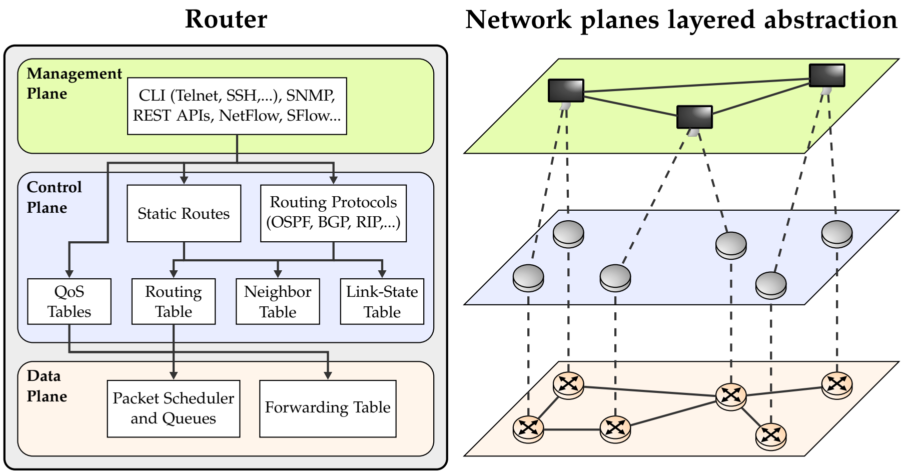

Router
Network planes layered abstraction
Management Plane
CLI (Telnet, SSH,...), SNMP, REST APIs, NetFlow, SFlow...
Control Plane
Static Routes
Routing Protocols (OSPF, BGP, RIP,...)
QoS Tables
Routing Table
Neighbor Table
Link-State Table
Data Plane
Packet Scheduler and Queues
Forwarding Table

FIGURE 2.1: The role of each networking plane (management, control, and data) in an IP router and its layered abstraction view.

and the management planes. Figure 2.1 depicts the plane split inside one IP router and its abstracted layered view at the network level. Each plane independently manages different parts within the networking stack, and together, they ensure the smooth functioning of the network device. The processes and functions involved by each plane are:

The management plane. The network device management plane is in charge of monitoring and controlling the network. It provides the necessary functions for a network operator to configure the devices, monitor and troubleshoot the state of the network, secure it, and maintain it. Monitoring and configuring are typically done through protocols such as CLIs, REST APIs, SNMP [12], NetFlow [28], and sFlow [13].

The control plane. The control plane is the "brain" of any traditional networking device. It carries out the required traffic signaling and is responsible for routing, learning topology information, and ensuring the desired Quality of Service (QoS). In short, the control plane decides where packets have to be sent. Networking devices' control planes are in charge of discovering and managing connections with other networking devices and use routing algorithms to determine the best path through the network for data packets to reach their destination based on some routing protocol

(RIP, OSPF, BGP). In traditional networks, most control plane functions are implemented in software.

The data plane. The data plane is responsible for processing incoming packets and either forwarding or blocking them based on the combination of all the different routing algorithms and logic running at the control plane. In addition, the data plane is in charge of traffic policing and queueing. In traditional networks, data plane functions are typically implemented in hardware to maintain current terabits-per-second forwarding speeds.

In short, operators define a network policy in the management plane, the control plane runs distributed algorithms to compute the needed state to ensure a given policy, and the data plane executes it by prioritizing, forwarding, or blocking packets accordingly. As shown in Figure 2.1, in traditional IP networks, all three planes reside within the router and operators configure their policies through the management plane interfaces. However, as we will show in Section 2.3, there have been proposals (e.g., SDN) to split the planes and move them to different locations to increase operators' control and protocol flexibility.

2.1.3 *Transport protocol*

The transport protocol is the fourth layer in the OSI layer model [29]. It is a fundamental component of any networking stack. It plays a crucial role in data communication. Specifically, it establishes end-to-end communication channels between hosts. The main function of the transport layer is to provide an abstraction over the best-effort IP layer and provide applications with a way to communicate with each other without necessitating direct interaction with the IP layer. Today, the two most common transport protocols in use are the Transmission Control Protocol (TCP) [30, 31] and the User Datagram Protocol (UDP) [32]. UDP, like the IP layer, operates on a best-effort end-host protocol and does not introduce substantial additional logic, which is delegated to the application. Conversely, TCP is a byte stream-based protocol that adds a level of abstraction on top of the IP layer, making it reliable and adaptable under different network conditions. The differences in design between UDP and TCP, especially their varying responses to network failures, significantly influence the design of traffic-driven network applications.

2.1.3.1 *The User Datagram Protocol (UDP)*

The User Datagram Protocol is a connectionless protocol that emphasizes simplicity and low overhead for fast communications. This focus on simplicity and speed, however, comes at the cost of reliability. UDP lacks delivery guarantees, does not handle packet re-ordering, and does not offer protection against packet duplication. Moreover, UDP clients typically do not adjust their transmission rates in response to varying network conditions, such as congestion or failures.

A key feature of UDP is its compact header. At only 8 bytes, it significantly contributes to the protocol's low overhead. The UDP header consists of only four fields; source port, destination port, length, and checksum. Note that the checksum does not provide reliability; it is only used to determine the packet's integrity.

The simplicity of UDP makes it an ideal candidate for simple request-response protocols such as the Domain Name Service (DNS) [33, 34], or time-sensitive data such as real-time audio or video, which are often transported over UDP. Furthermore, UDP's connectionless nature makes it perfect for broadcasting and multicasting, where a single packet is sent to multiple recipients without requiring a connection.

2.1.3.2 *The Transmission Control protocol (TCP)*

The Transmission Control Protocol (TCP) is a connection-oriented protocol that guarantees reliable data delivery. It achieves this through a combination of acknowledgments, retransmissions, packet sequencing, and state machine logic. In contrast to UDP, TCP requires establishing a connection between sender and receiver before data transmission can begin, a process known as the "three-way handshake." In this process, the client and server agree on the starting sequence number, starting window size, and maximum segment size.

TCP employs two mechanisms to regulate its transmission rate: flow control and congestion control. Flow control adjusts the sender's transmission rate to avoid overwhelming the receiver. This mechanism is useful when the receiver's processing capacity exceeds the sender's data generation rate. To communicate its available capacity, the receiver includes its remaining buffer capacity in the TCP header of each outgoing TCP segment. The second transmission rate mechanism, congestion control, aims to prevent

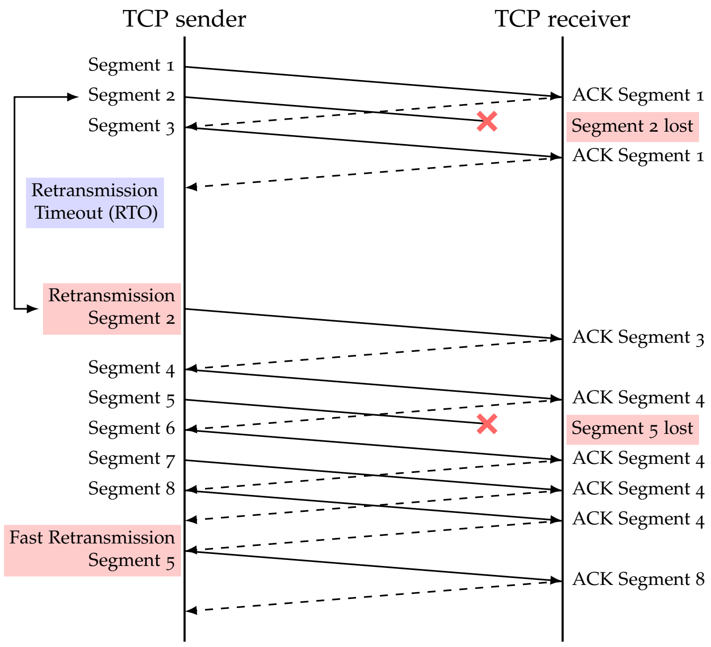


FIGURE 2.2: Simple example of TCP's segment retransmissions triggered by an RTO timeout and three duplicated ACKs.

packet congestion in the network. TCP's congestion control utilizes a congestion window to manage the sending rate. The value of the congestion window determines the number of maximum size segments (MSS) that can be sent without being acknowledged. At the start of a TCP transmission, the congestion window is set to one MSS. The window size is increased as segments are successfully delivered and reduced when congestion is detected. The amount at which the congestion window increases or decreases depends on the specific congestion control algorithm used and its configuration.

TCP's congestion control algorithms primarily use packet drops as an indicator of network congestion. There are two main methods by which a TCP sender can detect packet drops: (i) by the expiration of a timeout for an unacknowledged packet or (ii) by the reception of three duplicated acknowledgments (ACKs) with the same sequence number, triggering a fast retransmission from the sender. The TCP Fast Retransmission enhances TCP reaction in the event of packet loss. When the sender transmits data packets over the network, the receiver acknowledges each received packet by sending back an acknowledgment (ACK) packet. If a packet is lost and the sender transmits more packets after, if received, the receiver sends duplicate ACKs using the last in-sequence packet received. This process signals a gap in the sequence number to the sender, which, upon receiving three duplicate ACKs, retransmits the missing packet without waiting for the expiration of the regular packet's retransmission timer.

Figure 2.2 illustrates both scenarios. Initially, the sender transmits three TCP segments. The first is successfully transmitted and acknowledged, the second is dropped, and the third reaches the destination, triggering one duplicate ACK. In this case, with no subsequent segments arriving, a Retransmission Timeout (RTO) eventually leads to the retransmission of Segment 2. The second scenario, the fast retransmission, is depicted starting from Segment 4 in Figure 2.2. In this scenario, after Segment 5 is lost, the successful transmission of Segments 6, 7, and 8 triggers three duplicate ACKs of Segment 4. Upon receiving these three duplicate ACKs, the sender rapidly retransmits Segment 5 before an RTO timeout.

TCP congestion control algorithms react differently depending on how the packet loss occurred. Standard TCP congestion control algorithms set the congestion window to 1 MSS in case of an RTO timeout. In the case of three duplicate ACKs, the congestion window is decreased by half, and fast recovery is activated. These adjustments help to mitigate network congestion by scaling back the transmission rate in response to perceived network conditions. It is important to note that these behaviors are characteristic of standard TCP implementations such as TCP Reno and TCP New Reno. Other TCP variants might implement different strategies to adjust the congestion window.

2.1.3.3 *UDP and TCP reaction to drops*

As previously described, UDP and TCP are designed for distinct use cases. Among all their differences, the critical distinction for our purposes is the difference in how they react to packet drops. UDP traffic remains oblivious to network conditions, and its reaction to traffic drops depends on the applications built on top. In contrast, TCP uses a congestion control mechanism to dynamically adapt the sender's rate by modifying the congestion window in response to detected packet drops. In TCP, any packet drop can significantly change a flow's sending rate. As previously discussed, when TCP detects packet loss through triple duplicate ACKs, the congestion window is halved. In contrast, the congestion window is reset to one maximum segment size (MSS) if the loss is detected via an RTO timeout.

The substantial potential reduction in throughput due to packet drops underscores the need for rapid and sensitive failure detection mechanisms. Even network failures impacting a small subset of packets can cause significant disruptions in application performance, making the fast detection and fixing of these issues crucial. Understanding these fundamental differences in protocol behavior, particularly in response to network failures, is crucial for developing robust, protocol-aware network monitoring systems and failure detection mechanisms that can effectively identify and mitigate issues across diverse traffic types.

2.2 TYPES OF NETWORKS

Fundamentally, computer networks are interconnected computers and devices exchanging information. Depending on size, complexity, and functions, the Internet is composed of different network types, such as Wide Area Networks (WANs), Internet Service Providers (ISPs), Metropolitan Area Networks (MANs), and Data Center Networks. These networks range from city-wide to global coverage, facilitating enormous data transfers, enabling rapid Internet access, efficiently distributing digital content, and providing the infrastructure for massive computational processes and storage. Each network type is distinguished by various factors such as geographic coverage, bandwidth capacity, node-to-node delay, and underlying infrastructure. These characteristics have profound implications on the selection and design of network protocols, as they significantly influence the network's performance, scalability, and reliability. Understanding these characteristics is essential when deciding which protocols are best suited to run in these network environments. In this section, we briefly define the main characteristics of such networks.

Metropolitan Area Network (MAN). A network that interconnects LANs into larger geographic areas, such as cities or large campuses. It spans up to 50 kilometers, uses fiber links with high-capacity connections, and typical delays range from sub-milliseconds to several milliseconds.

Wide Area Network (WAN). A network that interconnects LANs or other bigger types of networks from different geographic locations. In most cases, connected networks belong to the same entity. Its bandwidth can vary significantly depending on the interconnected networks and technology used. Delays range from several milliseconds to hundreds of milliseconds.

Internet Service Providers (ISPs). Private organizations that provide services for accessing and using the Internet. They can interconnect WANs, and at the same time, they can be considered a type of WAN. ISPs can span the entire globe, nation, or region. At high tiers, they can reach hundreds of 100 Gbps. Node-to-node delays vary from sub-milliseconds to tens of milliseconds.

Data Center Networks. Private networks usually belonging to private organizations that interconnect many servers (i.e., hundreds of thousands). They are designed to be highly efficient, redundant, and provide high bandwidth (100 Gbps and more) between the servers in the data center. Within the data center, device-to-device one-way delays are typically below a hundred microseconds. Data center networks are excellent testing grounds for innovative networking strategies, given their inherent isolation. Such unique environment enables operators to deploy customized solutions that can be fine-tuned to their tenant's requirements and optimized to specific traffic patterns. Consequently, the majority of recent advances in network topology design [35–37], congestion control [38], load balancing [39–41], failure detection [11, 14, 17, 42, 43], and more, have been designed and deployed in the context of data center networks.

2.3 THE EVOLUTION OF NETWORKS: NETWORK PROGRAMMABILITY

Since its inception, the Internet and other computer networks have been characterized by the interconnection of many devices such as routers, switches, firewalls, NATs, load-balancers, and other network management devices. These devices are equipped with expensive, closed, and proprietary hardware and software. As pointed out by McKeown [44], the progressive reduction in the flexibility of network protocols caused by standardization has perpetuated this status quo. The hardware and software running in those devices implement network protocols that need to pass years of standardization and testing, slowing down innovation. For example, the Cisco hardware integration of VXLAN [45] took several years, which could have been achieved in weeks if implemented in software [46].

Traditional networking devices depend on vendors to implement new protocols and features. In many cases, when a software update is insufficient, it might be necessary to completely replace networking equipment with new hardware that supports the new feature. Furthermore, the dependence on vendor-specific implementations aggravates the complexity of managing these networks. Operators need to individually configure devices using heterogeneous management interfaces. These interfaces vary between vendors and sometimes even between products from the same vendor, making the process complex, tedious, and error-prone. This complexity not only risks potential errors but also has been a identified as a root cause of significant network outages, some of which have affected millions of users for several hours [47, 48].

These challenges highlighted the need for a new approach to networking. In 2008, a group of researchers addressed this need by proposing OpenFlow [49], which led to the development of Software-defined networking (SDN). SDN introduces a new networking abstraction and changes how networks should be designed and managed. First, SDN decouples the data and control planes and moves the control plane to a remote location. Second, the SDN control plane is logically centralized, capable of orchestrating an entire network of devices' data planes using policies and logic defined by the network operator. SDN controllers use a very feature-rich API called OpenFlow to instruct device data planes on how to forward traffic. OpenFlow abstracts the data plane to a set of match-action rules organized in tables that dictate how packets should be handled. An OpenFlow-enabled switch can perform various actions, including forwarding, dropping, mirroring, flooding, or modifying specific packet fields. With OpenFlow-enabled switches and controllers, network operators can write control plane applications that have a centralized and complete view of the network state and have the flexibility to make devices behave as a router, switch, firewall, NAT, or a hybrid between them.

However, while OpenFlow and SDN marked a significant stride towards improved network management and increased operators' flexibility, they are not without limitations. First, although OpenFlow match-action tables give the control plane some flexibility, packets can only match on a predefined set of headers fields, such as IP, UDP, TCP, ARP, or ICMP, as specified by the current OpenFlow standard [50]. Furthermore, the set of available actions is somewhat limited, with packet modifications restricted to only specific header fields. Second, a centralized and remote controller brings with it new challenges. It introduces a potential single point of failure, necessitating replicas to mitigate this risk. However, these replicas bring the additional challenge of synchronizing their network views. It raises scalability concerns due to the complexity of managing numerous switches. Additionally, it brings an increase in overhead and latency between the control and data planes, which could have an impact on time-sensitive systems. Finally, as OpenFlow-enabled switch vendors began to diverge in their feature implementations, the data planes grew increasingly complex and varied, breaking the initial idea of a simple control plane using a universal API.

Consequently, the fixed nature of OpenFlow data planes and lack of control of data plane packet processing has led to what can be seen as the natural evolution of SDN: making not only the control plane programmable but also the data plane. This new paradigm provides network operators complete flexibility to define how network devices should process and forward packets, the interface between the control and data plane, and which logic to run in the control plane. In the next section, we dive into the specifics of this groundbreaking advancement in network technology.

2.4 PROGRAMMABLE NETWORK DATA PLANES

Programmable data planes expand the concept of programmability beyond the control plane into the very core of the network packet processing, the data plane. This expansion offers unparalleled flexibility, enabling network operators to design customized solutions for their networks without involving device vendors or waiting for long standardization processes.

Historically, data plane programmability has been used in computer networks, primarily manifesting through software applications running on general-purpose CPUs such as x86 or ARM. These applications include software switches like Open vSwitch [51], VPP [52], and NetBricks [53], as well as userspace I/O accelerators like DPDK [54] and eBPF [55]. Although efficient and endowed with a degree of programmability, these solutions face significant challenges as they struggle to keep pace with the constant increase in high-speed links and low-latency requirements while maintaining complex processing logic.

To achieve both high speed and low latency while maintaining performance, Field-Programmable Gate Arrays (FPGAs) have traditionally been regarded as an attractive solution. These semiconductor devices, with their matrix of interconnected, configurable logic blocks, allow the designing and implementing of tailored network functions geared toward high-speed operations. A notable instance in the context of networking is the NetFPGA, an open-source FPGA-accelerated NIC, such as the NetFPGA SUME, which integrates a Xilinx Virtex 7 FPGA with four 10 Gb Ethernet ports [56]. However, even though FPGAs fulfill most requirements, programming them can be challenging. Most network operators lack the required digital systems knowledge and expertise in hardware description languages like Verilog [57] or VHDL [58], which are not targeted to network packet processing, making the approach accessible predominantly to a niche group of specialized engineers.

Over the past decade, data plane programming has been significantly reshaped by an innovative approach proposed by a consortium of leading networking experts [59]. This approach addressed the pressing demand for truly programmable data planes capable of high-speed, low-latency packet processing while remaining accessible to network operators without requiring expertise in an entirely new domain. Their contribution was the introduction of P4 [60], a domain-specific language tailored explicitly for programmable data planes. To complement this innovation, they also delineated a model for potential switching ASIC architectures that could be programmed using the P4 language. This has given birth to architectures such as the RMT [61], PISA [62], PSA [63], PNA [64], and notably, the Intel Tofino (TNA) [65] architectures. To illustrate the capabilities of these architectures, the latest Intel Tofino switch can forward an impressive 25.6 terabits per second (Tbps) and consistently handle up to 10 billion packets per second while maintaining exceptional packet-processing programmability [66].

In the following two subsections, we will describe a typical programmable architecture, its pipeline, the P4 programming language, and its associated development workflow.

2.4.1 *Programmable architectures*

Programmable ASIC architectures are organized as one or multiple pipelines of a combination of fixed, configurable, and programmable components. Figure 2.3 shows the pipeline of the PSA architecture. Setting aside minor details, it shares the same core components with the Intel Tofino TNA architecture. In the PSA or TNA architectures, when a packet enters the switch, first, a programmable parser extracts the packet into protocol headers following the parsing logic defined by the programmer and stores those headers in hardware data structures, called packet header vector (PHV).

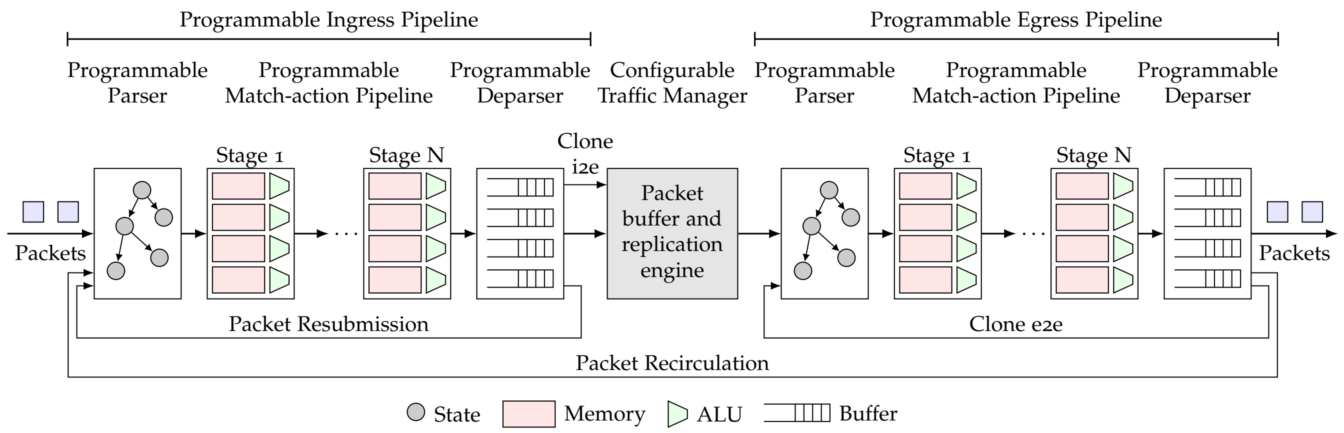


Figure 2.3: Portable Switch Architecture (PSA): each pipeline is composed of a programmable parser, a match-action pipeline, and a deparser. The ingress pipeline is connected to the egress pipeline through a configurable traffic manager. The PSA architecture also supports advanced packet processing features such as packet resubmission (allowing packets to be reprocessed by the ingress pipeline), recirculation (enabling packets to go through both pipelines multiple times), and packet cloning (creating copies of packets for multicast or monitoring purposes).

PHV fields can be read, written, and used for operations within every stage of the pipeline. Furthermore, each processed packet has access to metadata headers. Metadata headers are a set of packet-specific fields that, like parsed headers, are passed across the switch's pipeline and can be set, read, and modified. Metadata headers can be divided into user-defined or intrinsic metadata. User-defined metadata headers act like local variables that data plane programmers can use to store and carry information along packets, which can later be read or modified. For example, user-defined metadata can serve various functions: it may hold constant values, act as operands in arithmetic operations, or capture the results of computations. Intrinsic metadata comprises architecture-specific fields utilized by a switch's fixed components, such as the traffic manager. These fields carry critical information about the packet currently being processed – including ingress port, timestamps, and packet length. Additionally, the program logic running in the switch can modify the values of such metadata fields. This allows control of packet forwarding, dropping, and other architecture-specific functions such as mirroring or multicasting.

Once parsing is complete, the parsed headers, the initialized user-defined metadata, and the intrinsic metadata are passed on to the ingress pipeline, more concretely, to a set of match-action stages. In the match-action stages, packets go through a set of match-action tables (MATs). MATs implement their matching logic using SRAM and TCAM to store lookup keys and action data. The action part of MATs is implemented using arithmetic logic units (ALUs), which enable packet header and metadata modifications through a set of basic operations (i.e., addition and subtraction). Additional action logic can be implemented in most architectures using architecture-specific objects such as counters, meters, or registers.

When the packet has crossed all the pipeline stages, it is sent to the programmable ingress deparser, which serializes the modified headers, and based on intrinsic metadata, decides if the packet needs to be resubmitted back to the ingress parser for further processing, or simply has to pass it to the traffic manager and replication engine. The traffic manager and replication engine are a set of fixed functions that use intrinsic metadata fields to decide what to do with a packet, for example, to which egress port to forward it and if it has to be dropped, mirrored, or multi-casted. Furthermore, it is also responsible for packet buffering, queueing, and scheduling. Although not programmable, the traffic manager and replication engine are highly configurable through fixed control plane APIs. Next, the packet can be either sent out to the egress interface or to the egress pipeline, where packets can get further processed by the egress pipeline, which, as in the ingress pipeline, consists of a programmable parser, multiple stages of programmable match-action tables, and a deparser.

To program and configure each component of a programmable data plane, network engineers need a programming language to describe how packets should be parsed and processed by the switch, a compiler that translates that into a specific hardware configuration for each programmable hardware component, and a suite of APIs to configure fixed functions and populate the state of the match-action tables.

Standard programming languages and existing compilers are not a good fit for specialized hardware network functions. Therefore, in 2013, a group of researchers proposed P4, a domain-specific programming language tailored for data plane programming while being abstract enough to be used with many different architectures (targets). At the time of writing, P4, or Programming Protocol-Independent Packet Processors, is the most widely used data plane programming language.

2.4.2 *The P4 programming language*

The P4 programming language operates on a high level of abstraction, enabling the definition of packet-processing functions for any type of switching target. As described in the original paper [59], P4 was developed with three main objectives in mind:

Reconfigurability. The dynamic nature of network requirements demands adaptability. Thus, through P4, targets' forwarding logic should be able to be reconfigured rapidly and easily through an external API or controller. Furthermore, some packet processing logic should be redefined by configuring target parameters or state on demand.

Protocol independence. P4 is network protocol agnostic. P4 programmers must describe how the target switch has to process packets by describing how the packet parser has to extract bytes into protocol headers for each packet. This allows network operators to remove unused protocols, and add others that suit rapidly changing needs.

Target independence. The P4 language is designed to be device independent, meaning that the same program can be compiled for many types of architectures, such as FPGAs, network processors, ASICs, software models, or even CPUs. These different architectures are usually called P4 targets or target switches.

The first version ($P4_{14}$) and language specification [67] was released in 2015. However, to address many of the $P4_{14}$ limitations, in 2016, a new version of the language ($P4_{16}$) was drafted and released in 2017 [68]. $P4_{16}$ introduced support for different targets and architectures by splitting the language core components (described in the language specification [60]) from the architecture-specific ones, which now must be provided by the target vendor, together with a target-specific compiler. Furthermore, $P4_{16}$ introduced strict typing, expressions, many data types, nested data structures (i.e., stacks), and constructs to allow more modular programming. In brief, the $P4_{16}$ programming language provides a way to describe and configure any programmable device' packet processing pipeline (see 2.4.1).

The core components of the $P4_{16}$ language are described below. For a detailed definition, refer to the specification [60].

1. **Data types:** since $P4_{16}$ is a typed language, it includes some of the most common data types already existing in other programming languages. It includes bool, signed and unsigned bit integers, strings, enumeration, headers, headers stacks, header unions, tuples, and lists. Furthermore, it allows you to define your own types with the keyword *type*.
2. **Headers:** special data type used to define packet protocol headers, e.g., Ethernet, IPv4 or TCP. A header is composed of a list of one or multiple-bit fields. Protocol packet headers are typically initialized and filled during parsing, but also used and modified in the match-action pipeline.
3. **User-defined metadata:** data structures similar to headers that are associated with each packet and carried across the pipeline.
4. **Parsers:** a finite state machine (FSM) defined in $P4_{16}$ that describes how incoming packets have to be parsed and deserialized into headers based on the described parsing logic. For example, after parsing Ethernet's header and using the *EtherType* field, the parser logic can decide which protocol to parse next, e.g., MPLS, IPv4, IPv6, or any other protocol.
5. **Actions:** small code fragments that describe how packet header fields and metadata are manipulated. When executed as a result of a match-action table, actions can receive parameters supplied by the control

plane as action data. Most expressions available in the P4 core and architecture library can be used in actions. Action code will be executed sequentially or in parallel depending on the target, its compiler, and variable dependencies.

6. **Match-action tables:** a match-action table associates user-defined match keys with a list of actions. The match can consist of one or more keys each with an assigned *match type*. The $P4_{16}$ core library defined three types: *exact*, *ternary* and *longest prefix match (LPM)*, however, each architecture can have its own *match types*. The list of actions includes the names of all the actions that can be executed by this MAT. When defining a MAT, additional attributes include the maximum entry size, the default action to be executed upon a miss, and constant entries. P4 match-action tables can be seen as a generalized form of traditional switch tables such as forwarding tables, access-control lists, L2 learning tables, etc. MATs are declared in a P4 program. However, table entries are populated at run-time through control plane APIs.
7. **Control blocks:** contain the main logic of a P4 program. In control blocks you can declare variables, execute expressions, actions and sequences of match-action tables following the programmer's logic. Basic program statements such as as *if*, *else*, *switch*, and *exit* can be used to achieve that.
8. **Deparsers:** define which packet headers and in which order they have to be reassembled together with the packet payload. In some architectures, e.g., TNA, mirroring, resubmission, and recirculation are described at the deparser.

The main specific components of a target's architecture are explained below. The reader can refer to TNA [69] or v1model [70] architecture P4 definitions to see examples of actual architecture descriptions.

1. **Intrinsic metadata:** special metadata provided by the target's vendor and usually defined in the architecture part of P4. Like regular metadata, intrinsic metadata is a set of header fields associated with each packet and that can be read and written across the pipeline. Intrinsic metadata fields have mainly two uses: (i) store useful information provided by the target, e.g., ingress port, timestamps, queueing info, or (ii) to control the packet's flow or trigger some actions in the switch, e.g., deciding the egress port, mirroring, resubmission, recirculation, port queue, etc.

2. **Externs:** architecture-specific construct that extend the P4 core language. Like intrinsic metadata, they must be defined in the architecture file. Each target can implement a different set of externs. Some examples are counters, hash functions, meters, registers, and checksums.

Thanks to its well-considered design, the P4 language has become the de-facto standard for data plane programming. Its influence is evident not only in the networking research community, where it has gained significant momentum in recent years, particularly with the advent of the Intel Tofino switch, but also among network operators that use P4 to optimize their networks. Companies that own large data centers, like Facebook, Alibaba, and Google, have adopted P4 to implement custom functions, including network monitoring. Moreover, an increasing number of switch manufacturers are now offering devices that support P4 programmability [68].

2.4.2.1 *The P4 programming workflow*

The existing P4 programming tool workflow is one of the keys to P4's success. Figure 2.4 illustrates the typical P4 programming workflow. Manufacturers provide the programmable hardware (i.e., Tofino) or software (i.e., bmv2) target, the architecture definition, and the P4 compiler for the specific target. P4 programmers provide a P4 program that implements the desired packet processing logic using the specific P4 target architecture. The provided program is compiled with the manufacturer compiler, which produces two outputs. First, the compiler generates a target-specific binary, which it loads into the target to program it to achieve the programmer's desired packet processing logic. Second, it generates a data plane API that to interact with data plane objects such as match-action tables or externs.

Furthermore, to configure the target at runtime, the P4 programmer can provide control plane code implemented by leveraging the program-specific generated data plane API and the target's static fixed-function API. Through the APIs, the programmer can dynamically read and write the state of data plane objects (e.g., tables, registers, and counters) and configure or read statistics from fixed target components (e.g., port setup, mirroring, and packet scheduler). Furthermore, many targets allow direct control plane to target communication through internal PCIe or ethernet ports. These interfaces can be used to exchange entire packets or small digests of information.

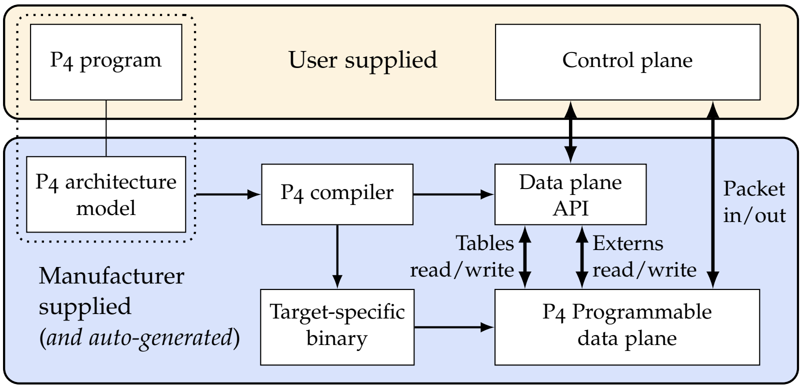


FIGURE 2.4: Diagram of a P4 workflow: the P4 program and control plane code are typically provided by the user. The other components are either manufacturer supplied or auto-generated by the compiler.

2.4.2.2 *Learning and prototyping P4*

Recognizing the barriers that potential data plane programmers or those eager to learn P4 might face without access to specialized hardware, and acknowledging the flexibility and convenience software-based environments offer, the P4 Language Consortium provides and maintains a set of software-based tools. The P4 Consortium established a dedicated GitHub repository [71], serving as an invaluable resource, especially for beginners in P4, and functioning as a central hub of materials. Among the key resources available are:

Behavioral Model (bmv2). Bmv2 [72] is a C++ software target enabling P4 programs on x86 architectures. It eliminates the necessity of owning a physical hardware target for executing of P4 programs and enables an easy integration with Linux network virtualization tools.

P4C Compiler. P4C [73] is the reference compiler for the P4 language, the p4c compiler can synthesize P4 code for an array of software targets. Its modular design allows hardware vendors to create their own target-specific backends for their proprietary P4 compilers.

Tutorials. To provide a more hands-on understanding, the repository includes tutorials that offer practical examples.

In the course of this thesis, we have made significant contributions to the P4 ecosystem through the development of a P4 prototyping framework and by expanding existing learning resources:

P4-utils. The Swiss Army knife for P4 development, p4-utils [74], is a Python package built on top of the Mininet [75] network emulator. It offers users an intuitive platform to conceptualize, create, and debug virtual networks. These networks can comprise P4 switches, controllers, hosts, conventional switches, and routers. Furthermore, for those desiring a plug-and-play experience, a Virtual Machine and its configuration are provided, ensuring a seamless transition to P4 prototyping.

P4-learning. Taking the spirit of learning and sharing forward, we have created a repository named P4-learning [76]. The repository is a collection of educational materials, including slides from the ETH Zürich Advanced Topics in Communication Networks course, and an extensive list of laboratory exercises and simple P4 program examples. The exercises and examples, implemented using p4-utils, showcase the features of the P4 language and the software switch while implementing advanced network functions such as Layer 2 learning, RSVP, load-balancing, routing, and packet loss detectors.

3 NETWORK FAILURES AND EXISTING DETECTION TECHNIQUES

In this chapter, we explore network failures and existing detection mechanisms. In Section 3.1, we provide an introduction to network hard and gray failures and their impact on ISP networks. First, in Section 3.1.1, we describe the various types of gray failures and demonstrate their impact on networks and applications performance. In Section 3.1.2, we provide evidence that networking devices from leading vendors are not exempt from bugs that may cause gray failures. At the end of the first part, in Section 3.1.3, we describe the results of a survey we conducted on the NANOG [77] mailing list. In this survey, we asked operators about the characteristics of their networks, the frequency and impact of gray failures, and how they deal with them.

In Section 3.2, we explore existing failure detection techniques. First, in Section 3.2.1, we provide an overview of the most relevant existing failure detection methods available in current networking devices and why they are insufficient to detect gray failures. In Section 3.2.2, we examine state-of-the-art failure detection systems, reviewing the latest failure detection techniques primarily designed for data center networks and highlighting why they can be ineffective in high-bandwidth and high-delay networks such as ISPs.

3.1 NETWORK FAILURES

Large-scale distributed networks, such as the Internet, are complex systems prone to various types of failures. Outages caused by DNS server errors [33], a problem with a DHCP server [78], a routing protocol (e.g., BGP) misconfiguration, or even power losses are common examples of failures that can occur. Such total or partial failures can lead to substantial service disruptions, which inevitably impact many entities, from end users and service providers to online retailers. A recent Amazon.com outage illustrates this well. A mere 49-minute downtime resulted in a staggering loss of around

$4 million in revenue [79]. Hence, speed is a high priority when detecting and recovering reliably from a downtime, as every second incurs a high cost. In addition, fast recovery is essential for mitigating financial impact and preserving customer trust in the service.

While failures can manifest in any part of the Internet, in this thesis, we focus on failures occurring at the network's core within networking devices (e.g., routers or switches), at their interfaces, or in the links that connect them.

Traditionally, when considering network failures, one might probably consider the so-called "hard" network failures. Hard network failures manifest through obvious issues such as malfunctioning device ports, complete link failures, or total device malfunctions and are characterized by their sudden and total impact on traffic. Despite the difficulty in predicting, preventing, or sometimes restoring from them, their immediate effects on the network make them relatively easy to detect.

However, network devices can also malfunction in more subtle ways beyond the obvious. For example, imagine, for some non-trivial reason, a device successfully transmits all traffic but drops packets that have a specific value in the header (e.g., a protocol or a port number), or even worse, randomly drops a tiny percentage (e.g., 0.1%) of packets without any apparent reason. These types of failures are known as gray failures. Gray failures are usually caused by faulty hardware components, software bugs, or device misconfigurations. The elusive nature of gray failures makes them complicated to detect, leading to prolonged service degradation and, in some cases, undetected outages. For example, a simple detection mechanism, such as sending ICMP ping probes between two points in the network, becomes ineffective if the gray failure does not impact the probe packets.

In the following Subsection 3.1.1, we dive deeper into gray failures. We will see which types exist, the different root causes, and how they affect traffic.

3.1.1 *Types of gray failures*

As noted earlier, gray failures refer to any hardware or software malfunction that results in non-persistent packet loss on a subset of traffic forwarded by any networking device. Gray failures manifest as service disruptions or application performance degradation, and can be classified into different categories based on their causes and how they affect traffic. This section summarizes the most common types of gray failures and their impact on traffic. Our understanding of these failures largely stems from previous studies conducted by data center operators and from a survey we conducted on the NANOG mailing list, which is primarily composed of ISP operators (see Section 3.1.3).

Most gray failures can be classified into one of the following two categories: random packet drops and packet blackholes. Within these categories, there are many variants that depend on the frequency of drops, the manner in which traffic is affected, and the root source of the problem.

Random packet drops. Random packet drops are gray failures in which a percentage (higher than 0% and lower than 100%) of packets get unexpectedly dropped. They often originate from faulty hardware but can also be caused by software bugs. Random packet drops often affect all the traffic uniformly, but can also affect specific subsets of traffic. For example, in a study by Microsoft in one of their data centers, they found random packet drops affecting all the traffic and specific source and destination pairs, which experienced a 1% random packet drop. As reported in previous studies [11, 14], random packet drops tend to be silent. Silent drops are typically not reported as drops by the switch, making them invisible to traditional detection techniques (e.g., SNMP [80]). Random packet drops can be induced by switching ASIC faults, CRC errors, poorly placed line cards, bent or damaged fiber, transmitter or transceiver issues, among others [11, 14, 16]. Zhuo et al., in their study "Understanding and Mitigating Packet Corruption in Data Center Networks", observed that random packet drops are relatively stable over time and that the loss rate does not depend on the link's utilization.

Packet blackholes. In contrast to random packet drops, packet blackholes (100% loss) are deterministic gray failures typically affecting specific traffic. For example, packet blackholes may affect one destination address or prefix, specific protocols, packet sizes, or combinations of them. Like random packet drops, packet blackholes can also manifest silently, meaning the drops do not register in the switch's counters or, for prefixes, in forwarding table drop counters [14]. Packet blackholes may occur when a switch TCAM entry gets bit-flipped or corrupted, but they can also happen due to more general device memory corruption or misconfigurations caused by software bugs or human errors.

3.1.2 *Characterizing gray failures through vendor bugs*

To better understand how gray failures happen in practice, we analyzed bug reports published by Cisco and Juniper, the leading routing vendors in the ISP market. We found roughly 150 bugs resulting in different types of gray failures.

Methodology. To find the relevant bugs leading to packet drops, we primarily utilized the bug portals of Cisco [81] and Juniper [82]. Access to these portals requires an account. We used a series of keywords to conduct a thorough search, as listed in Table 3.1. These keywords were selected based on their relevance to packet drop scenarios and were used individually or in combination with other words. This approach allowed us to search through the bug database systematically.

Search Keywords

dropped	drop	drops
packet	packets	silent
silently	loss	lost
unexpected	observed	corruption
corrupted	memory	TCAM
CRC	error parity	bit flip

TABLE 3.1: List of keywords used to find network device bugs. We used the keywords alone or in combination with other keywords.

The complete list of analyzed bugs is available in Appendix A.1. We categorized each identified bug according to its estimated root cause and provided a succinct description of the incident and its impact on traffic. It is important to note that our classification relies solely on the details available in the bug report and our best capacity to interpret them. Consequently, while our classification provides valuable insights, it is inherently subject to the limitations and level of detail provided in the reports. Our main observations are as follows:

Impact on traffic. Our analysis corroborates that the bug reports align with our prior characterization of how gray failures affect traffic; they can be specific to forwarding entries (e.g., IP prefixes) or more widespread, generally affecting all entries, often impacting certain ports or line cards. Within these classifications, the impact on packets varies; some packets might experience random drops, while others lead to complete entry blackholes. In Table 4.1 of the next Chapter 4, we present a selection of bugs categorized according to these findings.

Root causes. Out of the 149 analyzed bugs, software bugs, and unintended configuration or misconfigurations are the predominant root causes, accounting for 43% (65 bugs) and 27% (41 bugs), respectively. Hardware malfunctions and memory corruption problems contribute to 21% of the bugs, with the remaining 7% attributable to various other causes. Although classified as software bugs or misconfigurations, some issues are intertwined with hardware problems. For example, poor software-hardware integration could lead to incorrect hardware states. In misconfiguration scenarios, several cases involved unintended configurations (i.e., unanticipated by the device manufacturers), leading to unexpected device behaviors and gray failures.

3.1.3 *Operators survey: gray failures in ISP networks*

Failure detection research in data center networks has extensively documented the prevalence and frequency of gray failures [11, 17, 83]. To complement these studies and evaluate the relevance of gray failures across a broader spectrum of networks, we conducted a survey on the North American Networks Operators' Group (NANOG) mailing list. The objective was to determine whether gray failures present a significant issue in ISP networks and, if confirmed, identify the strategies operators use to address them.

The anonymous survey comprised twelve questions and was designed to take five to ten minutes to complete. Table 3.2 lists these twelve questions, divided into two categories. First, we asked operators about the type of networks they operate, such as type, bandwidth, and link delays. Second, we asked about their experiences with gray failures, including detection capabilities and the impact on traffic.

Before participating in the survey, we provided operators with a brief overview of gray failures and references to related studies in data center networks. For further details, the full thread from the NANOG mailing list is available online [77].

No.	Question
What kind of network(s) do you operate?	
1	What kind of network do you operate?
2	What link capacity do you commonly use to interconnect most of your network devices?
3	What link capacity do you commonly use to interconnect your high-capacity network devices (e.g., devices sitting in the core of your network)?
4	What is the typical (one-way) delay of most of your links?
5	What is the typical one-way delay of your high-capacity links?
What is your experience with gray failures?	
6	How often do you have to diagnose gray failures in your network?
7	Are dedicated gray failures detection techniques deployed in your network?
8	When you become aware of the existence of a gray failure in your network (independently of how), how do you typically determine the root cause?
9	When you become aware of the existence of a gray failure in your network (independently of how), how long does it take you on average to determine the root cause of the failure?
10	Have you seen gray failures affecting traffic in the following ways in the past? (see Table 3.3 for options)
11	What were the root causes of the gray failures you managed to debug in the past?
12	To finish, could you share with us some anecdotes of interesting (or perhaps particularly frustrating) gray failures you experienced in the past?

Table 3.2: Set of questions asked in the 2022 NANOG survey. Questions are divided in two categories: network characteristics and gray failures.

In the following subsections, we summarize the key findings learned from the answers. The results are based on responses from 46 anonymous network operators. It is important to note that, we excluded unclassifiable responses, such as "I do not know," slightly reducing the total number of respondents for some questions.

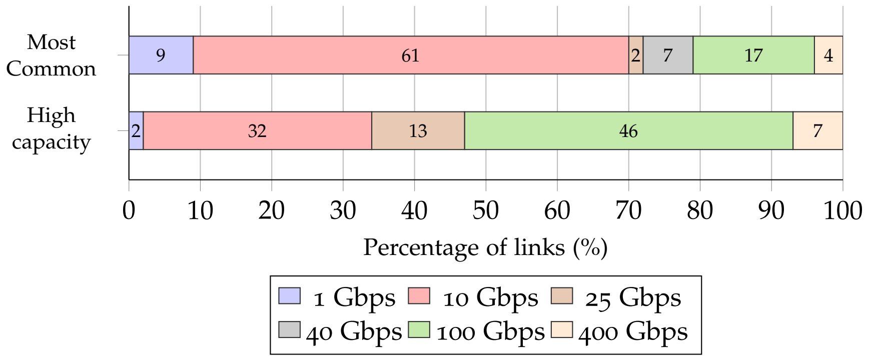


FIGURE 3.1: ISPs capacities distribution (1 Gbps to 400 Gbps) for most common links and high-capacity links.

3.1.3.1 *What kind of networks do they operate?*

Understanding the characteristics of ISP networks is crucial to determine the impact of failures and inform the design of an effective detection mechanism, as we will detail in Section 4.1.3. To this end, we asked operators about the types of networks they manage and the characteristics of their links (Questions 1-5).

Most network operators operate a WAN. Nearly 80% of the respondents operate a WAN network. Just over half report operating a stub ISP (54.3%) and a data center network (58.7%), while 41.3% are responsible for a transit ISP or an enterprise network. Notably, 76% of network operators indicated that they operate two or more network types, with 56% operating three or more.

Over 50% of ISPs' high-capacity links have a bandwidth of 100 Gbps or more. As illustrated in Figure 3.1, these high-capacity ISP links predominantly consist of 100 Gbps links (46%) and 400 Gbps links (7%). When considering the set of the most common links in all the ISP network together, 61% have a capacity of 10 Gbps. The proportion of links with a capacity of 100 Gbps or more reduces to 21% when evaluating all link types.

Approximately 40% of ISPs' high-capacity links have a one-way delay of 1 ms or more. Figure 3.2 presents the one-way delay for all ISPs' most common and high-capacity links. Delays of at least 0.1 ms are observed in 91% of the most common links and in 73% of the high-capacity links.

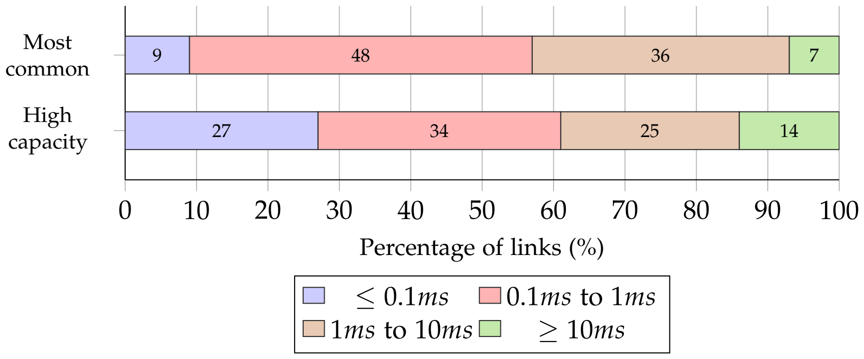


FIGURE 3.2: ISPs one-way delay distribution (0.1 ms to 10 ms) for most common links and high-capacity links.

Furthermore, in both categories, around 40% of links (43% among the most common links) have delays of 1 ms or more, with delays of 10 ms or more for 14% of high-capacity links. Notably, while ISP links are less likely to have short connections like those in data center networks, 27% of ISPs' high-capacity links have a one-way delay of 0.1 ms or less, compared to only 9% for the most common links. Although some ISP link delays can go down to 0.1 ms, there remains a substantial latency gap between ISPs and data center networks. For perspective, the average propagation latency within a data center fabric is typically around 1 µs [84].

3.1.3.2 *What do ISP operators say about gray failures?*

One of the main goals of our survey was to quantify the severity and significance of gray failures from the perspective of ISP operators, similarly to how previous studies have been done in data center networks. For that, we asked operators regarding the frequency they experience and need to diagnose gray failure-related problems, their detection mechanisms, and how they manifest on the faulty devices and traffic (Questions 6-12).

Gray failures are a significant problem for ISP operators. Indeed, gray failures present a real problem for ≈90% of the operators. Among these, as depicted in Figure 3.3, 15% experience gray failures every day, 17% weekly, 46% monthly, 78% semiannually, and 95% at least annually. A number of operators acknowledge that they typically investigate gray failures in response to customers' complaints.

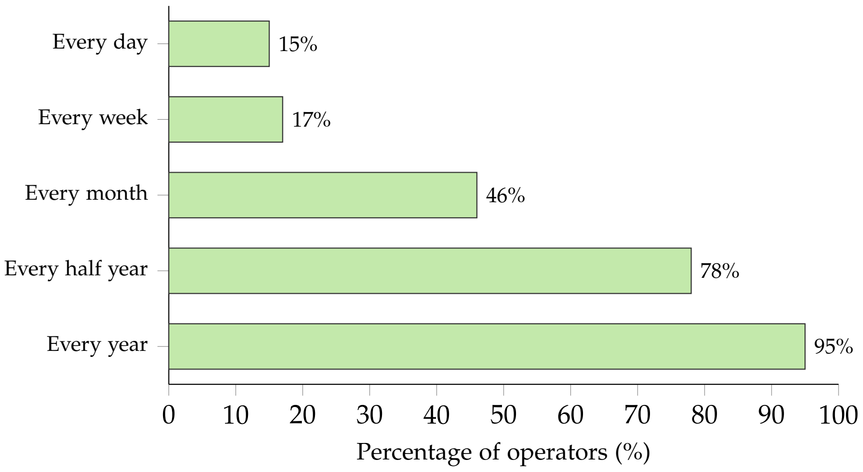


FIGURE 3.3: Frequency ISP operators experience and need to diagnose gray failures in their networks.

The reported data, while concerning, likely still underestimate the actual true number of gray failures occurring in ISPs. Notably, 73.9% of the operators admit that they lack a specialized gray failure detection tool, which suggests that numerous gray failures likely remain undetected. The remaining 23.9% of operators that do use detection tools mostly rely on rudimentary methods such as regular switch counters (e.g., SNMP) or endpoint packet probes. Only two operators reported the use of internally developed tools.

ISP operators struggle to diagnose gray failures. The consensus is clear: if detected at all, gray failures are difficult, time-consuming, and frustrating to debug.

Only 13% of the operators can diagnose them within minutes, while for 35% it takes hours, days for 20%, and weeks for another 20% of the operators. The remaining operators either do not know or measure how long it takes. This problem is partially due to the absence of effective tools to detect and localize gray failures. Commonly, troubleshooting consists of manual and sequential elimination of hypotheses. Given the hardware-specific nature of gray failures, operators often need to involve vendors in this process, slowing the process even further. Worse yet, a considerable number of gray failures go undiagnosed because they appear intermittently.

How is traffic affected by gray failures?	
A Failure dropping *all* packets for *some* forwarding entries	22 (48.9%)
A failure dropping *specific* packets (e.g., only small TCP packets) for *all* forwarding entries	19 (42.2%)
A failure dropping *specific* packets (e.g., only small TCP packets) for *some* forwarding entries	23 (51.1%)
A failure *randomly* dropping packets for *some* forwarding entries	24 (53.3%)
A failure *randomly* dropping packets for *all* forwarding entries	22 (48.9%)

Table 3.3: The answers to the survey's question #10 show that all possible ways traffic can be affected by gray failures are experienced by ≈50% of operators.

Furthermore, one of the operators raised an interesting point. On the one hand, customers are usually the first ones to notice anomalies. However, they have very little information about what is going on in the network. They are left wondering whether the issue is local, congestion, or a gray failure – they simply have no network visibility. On the other hand, despite having better insights, operators struggle to correlate between immediate network events happening *within the network* and the problems experienced *at the edge*.

Roughly 50% of the operators have experienced gray failures of all types. We asked operators whether they observed some of the most common forms of gray failure and their impact on traffic. Table 3.3 shows that for every different way traffic can be affected, roughly 50% of operators have experienced such pattern. The least frequent observed failure pattern was "A failure dropping *specific* packets (e.g., only small TCP packets) for *all* forwarding entries" which 42.2% of operators reported.

As described in Section 3.1.1, the root cause of most gray failures can typically be traced back to software bugs, misconfigurations, TCAM or memory corruption, parity errors, or malfunctioning line cards. Surveying operators on past gray failure root causes revealed software bugs as the predominant issue for 71.7% of the operators, as shown in Figure 3.4. Misconfigurations and malfunctioning line cards were each reported by 58.7% of operators. A quarter identified TCAM and memory corruption

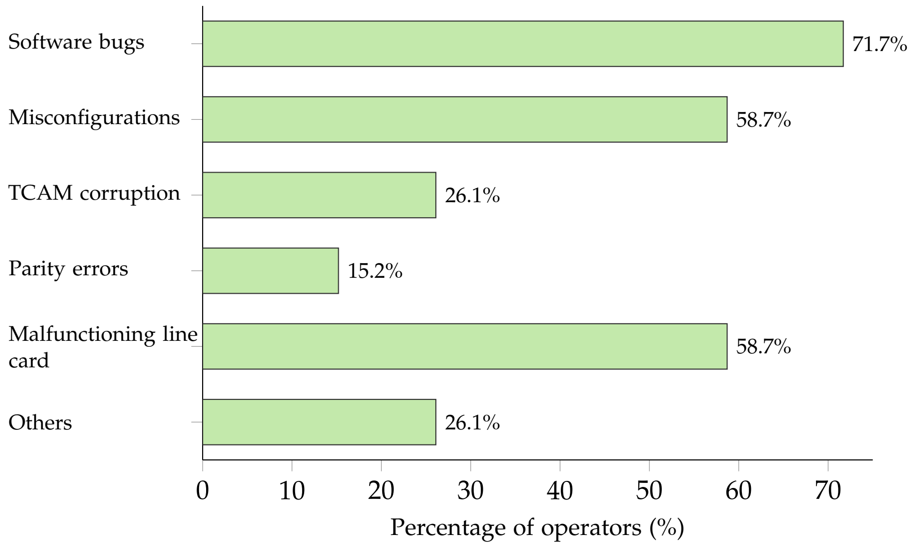


FIGURE 3.4: Percentage of operators that encountered each of the most common root causes of gray failures: Software bugs, misconfigurations, and malfunctioning line cards are the most common.

as the causes of gray failures. The least commonly observed root cause of packet drops was parity errors, reported by 15.2% of operators. Finally, 26.1% of operators have encountered other less frequent root causes such as MTU problems, faulty optics, or damaged cables.

In summary, this survey confirms the significant challenge gray failures present in ISP networks. Operators agree that gray failures occur frequently and are particularly challenging to detect. In most cases, operators only become aware of an issue when a customer complains. Additionally, when a problem is detected, determining the root cause can be a lengthy process. It often involves extensive dialogue with hardware vendors, which can be both cumbersome and time-consuming.

3.2 NETWORK FAILURE DETECTION TECHNIQUES

Network failure detection has been a long-standing problem since the beginning of computer networking. This problem has led to the proposal and development of numerous detection mechanisms. While some mechanisms are already deployable out of the box with cutting-edge networking devices (e.g., hardware BFD), while others are vendor-dependent or have been developed and tested only in controlled environments, such as data center networks. This section aims to summarize these mechanisms and discussing their capabilities and limitations. Furthermore, we explore why existing techniques may not suffice in high-delay and high-bandwidth networks, such as those operated by ISPs.

3.2.1 *Existing vendor network failure detection techniques*

Due to the significant impact of network failures and the necessity for rapid detection, network vendors have developed a wide range of failure detection mechanisms. Failure detection mechanisms are typically implemented on most modern network devices across the first three OSI network model layers [29]. When designing a network, operators select which detection mechanism to use based on their needs, weighting factors such as detection speed, overhead, available features, and complexity.

Physical layer detection. At the lower layer of the stack, failures can be rapidly detected with techniques such as built-in Ethernet auto-negotiation, carrier delay, loss of light, and link fault signaling. While these techniques offer simplicity and a fast response time, they are limited to major link problems. Importantly, they cannot be used to detect unidirectional issues, problems only visible at a higher layer, or issues caused by cables that result in spurious transmission errors.

Data link layer detection. Layer-2 technologies also provide failure detection mechanisms, including technologies such as Link Aggregation Control Protocol (LACP) [85], Unidirectional Link Detection (UDLD) [86], or link Operations, administration, and management (OAM) for error frame and CRC checks [87]. While these Layer-2 mechanisms are generally easy to configure and widely supported across devices, they rely on the control plane and, therefore, tend to be slow. For example, UDLD detection time is in the order of tens of seconds. In the case of LACP, timers can be set to one second, providing a minimum detection time of 3 seconds. Moreover, these mechanisms are designed only to detect issues between directly connected Layer-2 devices rather than end-to-end network issues. Furthermore, like physical layer mechanisms, data link layer solutions can only detect hard cable failures.

Network layer detection. The network layer employs HELLO messages in nearly all routing protocols for basic link failure detection. Protocols such as OSPF [24] and EIGRP [88] periodically send HELLO messages to all neighbors, whereas BGP [27] utilizes KEEPALIVE messages to ensure peer reachability. Although effective for identifying link issues between layer-3 devices, HELLO methods are inherently slow, with detection times in the order of tens of seconds. To achieve quicker detection times, they require increasing the frequency of HELLO messages. However, this approach cannot achieve sub-second failure detection even with the lowest timer setting. Additionally, lowering the timers would increase device CPU usage, potentially affect the routing protocol and other processes, and, in the worst case, result in false positives. To overcome these limitations, Juniper Networks introduced bidirectional forwarding detection (BFD) [89], a low-overhead and routing protocol agnostic failure detection protocol designed for sub-second failure detection. BFD can run either in the control plane's software or directly within the data plane's hardware. However, the second option is limited to high-end network devices. BFD has two operation modes: asynchronous and echo. In asynchronous mode, routers running BFD periodically send packets to all neighbor control planes, which process them to ensure liveliness. While more efficient than routing protocols' HELLO mechanisms, BFD asynchronous mode can still incur a high CPU overhead. Echo mode reduces the CPU load by having the prober send packets with itself in the destination address. This makes the remote BFD neighbor forward them back along the same path without involving the CPU. The echo mode is particularly beneficial as it lowers the receiver's CPU usage and tests its forwarding path. BFD can be offloaded to the device's line card to further reduce CPU utilization, achieving detection times as low as 50 ms.

Despite BFD's efficiency in rapidly detecting network hard failures and its potential for hardware offloading, BFD still relies on periodic health checks, limiting its ability to detect failures that only affect a subset of the traffic (i.e., gray failures). Furthermore, while offloading BFD to a device's line card offers significant performance benefits, this does not entirely mitigate the potential bottleneck in the control plane during the post-detection response phase. In the subsequent section, we evaluate the control plane's influence

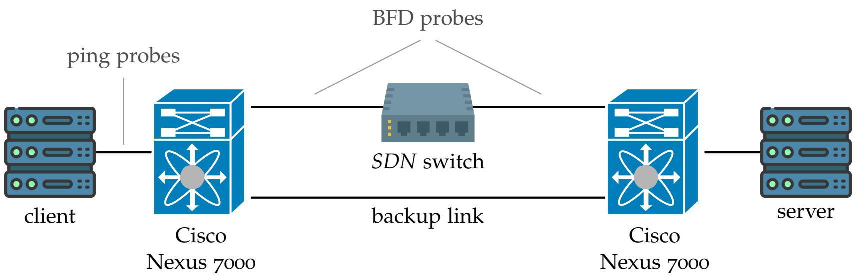


FIGURE 3.5: BFD experiments lab setup: Two Cisco Nexus 7000 switches connected via an SDN switch to simulate link failures, with a backup link to reroute traffic upon detection. Two servers are used for convergence time measurements and to advertise routes.

on overall convergence time following a failure, even when BFD is running offloaded in the line card.

3.2.1.1 *Measuring BFD's performance under different control plane loads*

To evaluate the efficiency of Bidirectional Forwarding Detection (BFD) under different control plane loads, we conducted a series of controlled experiments in our laboratory. Figure 3.5 depicts our experimental setup, which consists of two interconnected Cisco Nexus 7K switches and a layer-2 Software-Defined Networking (SDN) switch placed between them to simulate link failures. By using the layer-2 switch to drop packets, we ensure that built-in physical layer detection mechanisms, such as carrier delay, cannot detect the failure. As a backup, we included a direct cable connection between the Cisco Nexus 7K switches. Furthermore, we connected a server to each switch; the server connected to the layer-2 switch activates and deactivates the drop commands, while the servers connected to the Cisco Nexus 7K switches are used for both measurements and route advertisements.

These experiments were designed to thoroughly assess BFD's performance under various conditions, including different timer intervals, routing protocols, and control plane loads. Our Nexus switches, equipped with hardware-offloaded BFD features, were set to operate in echo mode for this evaluation. We tested BFD with varying timer intervals: 15 ms, 30 ms, 50 ms, 100 ms, and 200 ms. BFD utilizes a multiplier, which we configured to

3. This setting implies that a failure is detected only after three consecutive probe losses.

In the experiments, we measured the total convergence time. The convergence time includes the detection time (done by BFD), routing protocol computations, and subsequent data plane updates. To this end, we used the fping [90] tool, configuring it to send one probe every 1 ms to facilitate accurate measurement of convergence time. We then defined the convergence time as the duration between the failure and the arrival of the first successful fping probe.

To determine the influence of the running routing algorithm and control plane load on the convergence time, we ran the experiments under three different scenarios:

Static routing scenario. We designated the upper link as the primary, with the lower link set as a backup that activates upon BFD, notifying the control plane of a failure.

OSPF scenario. We ran OSPF on all router interfaces and adjusted the cost of the upper link to make it the preferred path for switch-to-switch traffic.

BGP scenario. We positioned the servers in different Autonomous Systems (AS) and established peering relationships between the first switch and two additional ASes. These two ASes advertise 10,000 prefixes to the switch. When a failure is detected, a considerable computation gets triggered to recompute and update the forwarding entries for these 10,000 prefixes. This scenario aimed to underscore the potential impact of a busy control plane CPU on the total convergence time, even when a fast hardware detection mechanism is in use.

For robustness and to ensure statistical relevance, we ran 60 iterations of each experiment, covering every BFD timer and routing protocol mix, resulting in 15 distinct experiment sets and a total of 900 runs. Figure 3.6 illustrates the convergence results of our BFD experiments. Each row corresponds to a different routing protocol scenario, while each column represents a different BFD probing time in ascending order.

With Static routing, detection times appear to follow a normal distribution centered around the multiplication of the probing speed by the multiplier. This suggests that the convergence time is nearly identical to the detection time due to negligible control plane load.

In the OSPF scenario, we notice a similar trend. However, when lowering the probing timers, we see an increase in worst-case scenarios compared to static routing. Additionally, the center of the normal distribution is shifted slightly, indicating longer average detection times. Although the control plane overhead impacts some runs, given the simplicity of OSPF computation in our scenario, the overall effect remains limited.

When we incorporate BGP and overload the control plane by forcing it to recompute and update the forwarding for 10,000 prefixes, the contribution of detection time to total convergence time becomes less pronounced. For the smallest probing time, 15 ms, where we would typically expect convergence times of approximately 50 ms, we encounter a control plane overhead exceeding 100 ms in most cases. This trend persists with larger timers, although the proportion of overhead in the total convergence time becomes less noticeable. This observation raises an interesting point: in scenarios where the control plane is loaded, or many forwarding rules need to be updated, the control plane becomes the most significant contributor to the total convergence time. This underscores the control plane as another critical area for improvement. We explore this idea in Chapter 5.

In this subsection, we have conducted an in-depth analysis of vendor-specific failure detection methods, whose applicability varies based on the unique demands of network design and operational requirements. Our research particularly emphasizes the advantages of BFD for its speed, simplicity, and efficient resource utilization compared to other alternatives. Despite the effectiveness of these methods, they primarily address the detection of "hard" network failures, offering limited results for gray failures. Moreover, our experiments highlight how control plane bottlenecks can significantly impact total convergence times. This underscores the need for advanced detection systems for gray failures and innovations to accelerate control plane's subsequent steps during the convergence process. In the next section, our focus shifts to advanced detection mechanisms capable of detecting a broader range of failures.

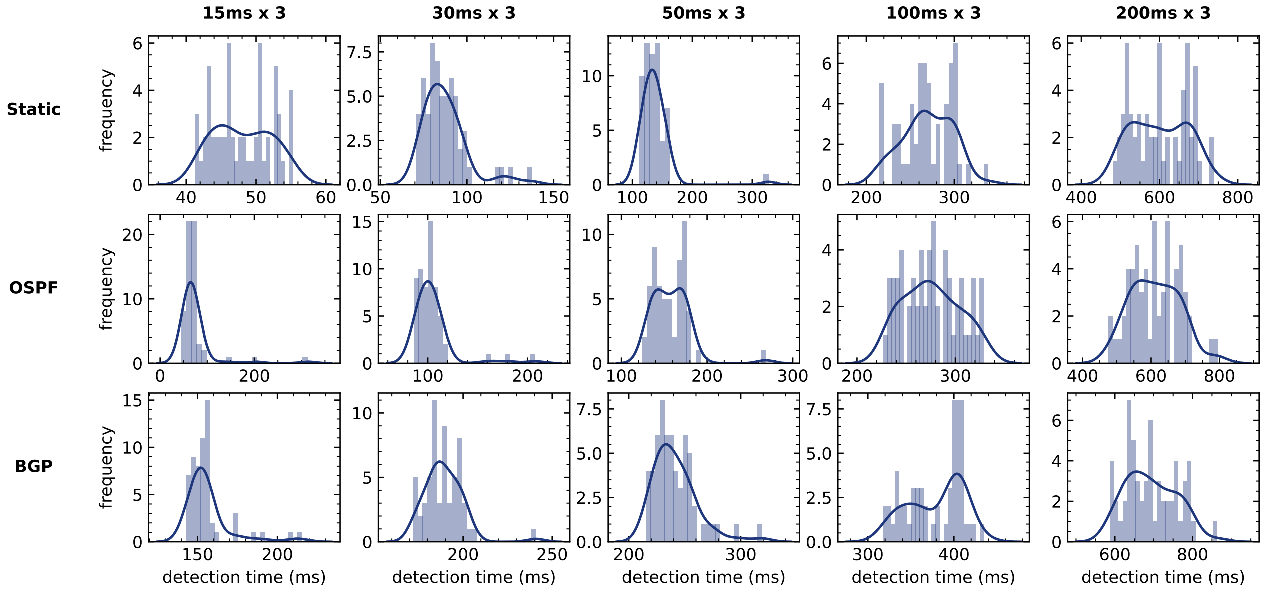


FIGURE 3.6: Comparison of convergence times under varying control plane scenarios and BFD (with hardware-offload) probing timers. The figure shows histograms and fitted probability density functions for detection times across three routing protocols (Static, OSPF, and BGP) and five BFD probing intervals (15 ms, 30 ms, 50 ms, 100 ms, and 200 ms), each with a multiplier of 3. The results illustrate how different routing protocols and control plane loads affect convergence times in relation to BFD timer settings.

3.2.2 *Advanced network failure detection techniques*

In the previous section, we explored the network failure detection techniques available in standard networking equipment. We explained their advantages and disadvantages and highlighted their primary focus on detecting hard rather than gray failures.

This section introduces advanced failure detection and monitoring techniques suitable for both hard and gray failures. Although we call these techniques *advanced*, we explore detection techniques that go from straightforward packet counting mechanisms, such as polling switch counters (e.g., SNMP) or aggregated and sample counter statistics (e.g., NetFlow or SFlow), to state-of-the-art research solutions that utilize end-hosts, advanced switch capabilities, and external controllers that hold a global view of the network. We conclude the section by explaining why most of these solutions do not work in high-bandwidth (>=100 Gbps) and high-delay (>=1 ms) networks such as ISP networks.

3.2.2.1 *Host-based monitoring techniques*

End-host based techniques fall into two categories: passive and active. Generally, these methods are most applicable to data center networks, as they require sending traffic probes or monitoring traffic from hosts or virtual hosts through a hypervisor.

Passive Monitoring. Passive approaches are often integrated within the host's network stack or in data centers in the server's hypervisor [91–97]. These methods focus on tracking flow statistics, such as TCP parameters, over time. By analyzing data locally and globally, passive monitoring can infer various types of network issues but struggle to localize them accurately.

Active Monitoring. In contrast, active monitoring techniques inject traffic probes to assess the current state of the network and to investigate how specific traffic gets affected [11, 98–100]. Some systems combine active and passive methods, creating a hybrid approach. These systems initially use passive monitoring to identify anomalies and then switch to active probing for a more detailed examination of specific issues.

While end-host based detection systems can detect network failures, device malfunctioning, and even pinpoint which traffic or hosts are being affected, their reliance on inference limits the ability to accurately determine the problem's location and root causes. These systems generally suffer from slow detection speed due to the necessity of maintaining a low probing frequency and typically rely on a central controller or agent for the export and analysis of host statistics. Furthermore, they contribute to CPU overhead and, in the case of active probing, additional bandwidth consumption, which is usually compensated by trading it by detection coverage (i.e., monitoring fewer traffic entries). Finally, host-based monitoring techniques are impractical in ISP networks, where network operators lack control over end hosts.

3.2.2.2 *In-network controller-centric approaches*

Unlike end-host based solutions, in-network controller-centric methods prioritize the monitoring and detecting issues through data collected directly within network devices, like switches. This data is then exported to a centralized controller for analysis and interpretation.

In-network controller-centric detection systems range from simple counter-based techniques, such as SNMP [12], to more sophisticated techniques that use probabilistic data structures, such as sketches or invertible bloom filters [101].

Packet and Flow-based counters. A straightforward method for detecting gray failures is using packet counters within network switches. These devices are typically equipped with counters to track various metrics, including packet and byte counts for both incoming and outgoing traffic on each port, as well as packet drops resulting from unusual switch activities. These basic packet counters are accessible from an external controller via SNMP [12]. While packet counters can offer a broad view of what is happening in the network, they fall short of providing the information required for detecting gray failures or debugging specific issues. Moreover, as demonstrated by Pingmesh [11], hardware defects or memory corruption may lead to SNMP counters inaccurately reporting normal operations amidst actual packet losses.

For enhanced monitoring capabilities, many networking devices also support flow-based counters that aggregate statistics on TCP and UDP traffic flows. Protocols like NetFlow [28] and sFlow [13] are commonly employed for this purpose. Contrary to basic SNMP counters, NetFlow and sFlow provide flow-level information, enabling operators to record data such as source and destination IP addresses, port numbers, and protocols types, offering a more comprehensive view of the network's traffic. However, despite their advantages over simple counters, flow-based counters still face challenges when used to detect gray failures. First, the data is often exported in intervals ranging from several seconds to minutes [28, 102], leading to delayed detection. Second, to maintain line-rate data recording, these systems often resort to packet sampling [13], which significantly compromises detection accuracy. Finally, accurately identifying packet drops requires perfect synchronization between exported counters [103], a need these techniques lack.

Packet mirroring. It is a controller-centric method that involves mirroring (i.e., making copies) packets from switches to a controller in a coordinated and synchronized way to detect packet losses between devices. By examining traffic mirrored from all switches, the controller can identify which and where packets were lost. Although highly effective for gray failure identification, packet mirroring becomes impractical in high-bandwidth settings like ISP networks. This technique would require sending a copy for every packet that crosses a switch to effectively detect gray failures, significantly increasing bandwidth demands relative to the path's link count and bandwidth. In ISP networks, this would translate into hundreds of Tbps of traffic to store and analyze, making it an impractical solution.

Previous work has explored more practical packet mirroring options to circumvent scaling issues. However, the constant increase of network bandwidths and trade-offs required for scaling makes those systems impractical for gray failure detection. Planck [102] introduces sampling to reduce the burden, but this makes mirroring non-deterministic and thus ineffective for loss detection. NetSight [104] truncates the mirrored packets and only sends the packet header to the controller. However, such efforts are insufficient as with only headers the aggregated load in the network remains very high and the number of packets to process remains the same. Alternatives like Stroboscope [15] and Everflow [14] reduce the load by only mirroring a subset of packets according to a budget, however, decreasing the coverage and requiring operators to configure what to monitor, compromising their utility for detecting gray failures.

In-band Network Telemetry. The emergence of programmable data planes has revolutionized network telemetry through In-band Network Telemetry (INT) [105, 106]. Unlike conventional out-of-band methods such as mirroring or counter collection, INT allows real-time collection of network device state information such as queue depths, delays, port statistics, and path information. As packets traverse the network, programmable data planes embed custom metadata into packet headers, allowing operators to tailor the information added. Upon reaching the destination or a predetermined sink node, this metadata is extracted and relayed to a telemetry collector for analysis INT offers immediate, detailed insights into network performance, latency, and other operational characteristics. Despite being an effective way to get rich in-network metadata, INT-based solutions like mirroring solutions, must navigate the balance between coverage and operational cost, again, limiting its effectiveness for gray failure detection.

Sketches. Sketches offer an alternative to methods like counters and packet mirroring. They involve instructing switches to store packet measurements in probabilistic data structures called sketches. These data structures provide a way to store compressed measurement data, which can subsequently be extracted, decompressed, and analyzed by a centralized controller. While many existing sketch solutions [107–116] are effective for approximating flow sizes and packet counts, they are generally not well suited for specific tasks such as packet loss detection due to their trade-off between accuracy and memory efficiency.

In recent years, with the emergence of data plane programmability, specialized sketches designed explicitly for failure detection have emerged [42, 103, 117].

FlowRadar [103], a networking monitoring system, leverages Invertible Bloom Filters (IBF [101]) to encode per-flow counters, which are periodically pulled and decoded at a remote controller. The decoding process the IBFs from across the network to extract flow counters, enabling loss detection by comparing the counters from successive hops. FlowRadar has two main limitations: it scales with the number of active flows, and requires all the IBFs to be collected in a synchronized way which results in a coarse-grained detection timescale [42].

LossRadar [42] is a packet drop detector that uses IBFs like FlowRadar. However, to scale with the number of packet losses and not with the number active flows, LossRadar periodically XORs the IBFs of two directly connected switch ports. With this operation, all the successfully transmitted packets get removed from the IBF, leaving only the lost packet for decoding. Consequently, this enables precise identification of both the location and the 5-tuple details of packets lost in transit between the two IBFs. Alternatively, ChameleMon [117] proposes a hybrid between the two (FlowRadar and

LossRadar requirements

		Average loss rate			
Switch	Metric	0.1%	0.2%	0.3%	1%
100 Gbps 32 ports	memory size*	× 0.21	× 0.42	× 0.63	× 2.1
	read speedup†	× 0.7	× 1.4	× 1.9	× 4.5
400 Gbps 64 ports	memory size*	× 1.7	× 3.4	× 5.1	× 16.9
	read speedup†	× 3.7	× 6.6	× 9.5	× 29.5

* LossRadar req. memory / memory available per hardware stage

† LossRadar req. read speed / available hardware read speed

Table 3.4: Even for registers' (64 bits) and packets' (1500 B) sizes minimizing memory reading time, LossRadar exceeds the capabilities of state-of-the-art switches (see red numbers).

LossRadar) and introduces the FermatSketch an IBT-based data structure that dynamically allocates its memory to only track "victim" flows and thus does not suffer from the same scalability problems as FlowRadar.

Despite being effective, controller-heavy based systems, have a common problem: controllers have to continuously pull large stateful data structures from the data plane. This requirement makes these solutions impractical for high-bandwidth networks such as ISPs.

To illustrate that, we consider LossRadar [42], one of the few sketch-based systems able to detect gray failures. In LossRadar, to ensure a quick failure detection and avoid the pollution (i.e., by too many packets encoded) of the IBFs, the IBFs need to be dimensioned accordingly, and they must be extracted from the data plane to the controller very often (i.e., every 10 ms).

Table 3.4 shows the results of measurements we performed on a state-of-the-art switch [18]: current switches do not read memory fast enough for LossRadar to support average loss rates higher than 0.15% in 100 Gbps switches with 32 ports. LossRadar limitations worsen for higher bandwidth and port number counts.

Note that switches' memory size and reading speed constraints limit the options available to sketch-based approaches: extracting measurements less frequently requires larger data structures, which however exacerbate hardware limitations. For example, in LossRadar, gathering IBFs less frequently is counter-productive because it requires increasing their sizes to deal with the higher number of packets lost during larger intervals for the same loss rate; yet, larger IBFs further reduce the loss rates detectable by LossRadar.

Our results show that LossRadar fundamentally cannot detect gray failures efficiently within current and future ISPs (with constantly increasing bandwidths), unless a major technological breakthrough enables switches to support significantly more memory and read it much faster than today. Other sketches may make a more parsimonious use of switches' memory, but they are likely to suffer similar scalability limitations with respect to the tracked traffic. In addition, a recent study [118] shows that all sketch-based solutions have a significant accuracy drop (up to 94$\times$) compared to theoretical expectations due to the delays in retrieving the data plane state. Those limitations lead us to design an in-switch failure detection system.

3.2.2.3 *In-network computation-centric approaches*

In-network computation-centric approaches are programmable data plane based detection methods that heavily leverage deep in-network programmability to run most of the detection in the data plane, and thus requiring less data plane to control plane communication. This approach directly addresses the bottleneck issue prevalent in sketch-based systems by enabling a majority (if not all) of the detection process to occur within the data plane itself. Three techniques have recently been proposed to detect failures within switches and without much control plane involvement: Blink [3], NetSeer [17], and dDrops [119].

Blink [3] focuses on detecting failures that affect all flows crossing a remote failed link. It selects a few flows (e.g., 64) per prefix and checks if the majority of them retransmit within an 800 ms window. Blink fundamentally cannot detect a gray failure that does not affect most of the flows crossing a link; in those cases, Blink does not monitor enough affected flows. For cases in which Blink could select more than 32 affected flows, gray failures increase the likelihood that retransmissions are spread over time, beyond 800 ms windows, since only a subset of the packets is lost, which would again prevent Blink from detecting the failure. In fact, as the loss rate decreases (i.e., lower %), Blink's accuracy also drastically decreases. Extending Blink to detect gray failures is also challenging as (i) monitoring more flows is impractical, given switches' computational and memory resources, and (ii)

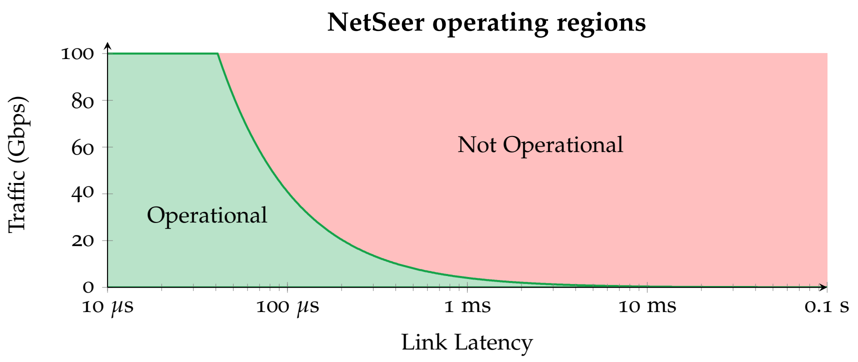


FIGURE 3.7: NetSeer can only report absolute packet losses in typical ISP link delays (i.e., $\geq 100\mu s$) and traffic volumes. The plot considers 1024B packets and a 1000-cell NetSeer's buffer: smaller packets or buffers further decreases NetSeer's applicability.

inferring failures from the retransmissions of fewer flows would lead to many false positives.

NetSeer [17] is an in-switch system designed to detect a variety of events, including gray failures, in data center networks. It includes mechanisms to identify packet drops internal to switches (e.g., caused by congestion), as well as an inter-switch protocol for detecting the most general class of gray failures.

NetSeer effectively detects internal losses that are accurately logged by switches. Its inter-switch protocol utilizes switch memory proportional to the link's bandwidth and device-to-device delay. While this approach is efficient in environments with minimal delay, such as data centers, it would struggle to accurately detect gray failures within ISP networks. Indeed, in NetSeer's inter-switch protocol, each upstream switch stores a signature of sent packets in a buffer, adds a sequence number to sent packets, and receives NACKs from neighbors whenever any such packet is lost, which is detected by the downstream switch checking for any gap in received sequence numbers. Fundamentally, NetSeer's packet buffers have limited size, and in ISPs, they are likely to be overridden before NACKs are received, because of ISPs' traffic volume and link delays. Whenever this happens, we say that NetSeer is not operational since it has no visibility on losses per entry and, therefore, cannot localize the corresponding gray failures.

Figure 3.7 shows the operational regions of NetSeer at 100 Gbps or lower, considering buffer sizes of 1000 packets per port, in which each cell requires 13 bytes (flow 5-tuple) and assuming a packet average size of 1024 bytes. Results shown in Figure 3.7 are computed analytically and confirmed by experiments we conducted in ns-3. Note that smaller average packet and buffer size further decreases NetSeer's applicability. Additionally, the assumption of a 1024-byte average packet size is somewhat optimistic, given that ISP networks often report smaller averages [120]. The parameters selected for the analysis align with those recommended and used in NetSeer's own evaluations [17], and in total they require 800 KB of SRAM memory. With the given fixed buffer allocation, NetSeer's buffer gets completely overwritten when the switch receives more than 1000 packets before a NACK from the downstream is received. Therefore, NetSeer's operational area directly depends on the rate at which packets are received, and the delay between upstream and downstream switches.

Figure 3.7 shows that at 100 Gbps, NetSeer is only operational if the link delay is 40μ or lower. For 10 Gbps interfaces, the maximum tolerable delay goes up to 0.4 ms. These findings underscore NetSeer's operational viability within data center environments, characterized by minimal delays. However, it would not be operational in most common ISP settings where traffic per link exceeds 100 Gbps and link latency is on the order of milliseconds, as confirmed by the operator survey we conducted.

Figure 3.8 shows the required memory for NetSeer to be operational for different switch settings and inter-switch link latencies. For instance, to remain operational at a 1 ms delay, NetSeer requires 20 MB of memory for a switch equipped with 64 ports at 100 Gbps and 80 MB for a 400 Gbps switch. Therefore, as Figure 3.8 shows, the required memory increases proportionally as delay or bandwidth increases. For example, with a 5 ms delay, the required memory is five times more than for 1 ms.

In ISP environments, where link delays typically span from 1 ms to over 10 ms, the memory required by NetSeer (i.e., hundreds of MBs) exceeds the available memory in today's switches. Indeed, current state-of-the-art switches offer about 12-15 MB of memory per pipeline, with 4-8 pipelines in total [121]. Moreover, this memory is distributed across the stages of each pipeline [122, 123], meaning that an in-switch application is, in practice, constrained by the maximum per-stage memory. In addition, per-pipeline per-stage memory is shared across all in-switch applications, further reducing the memory available to each application.

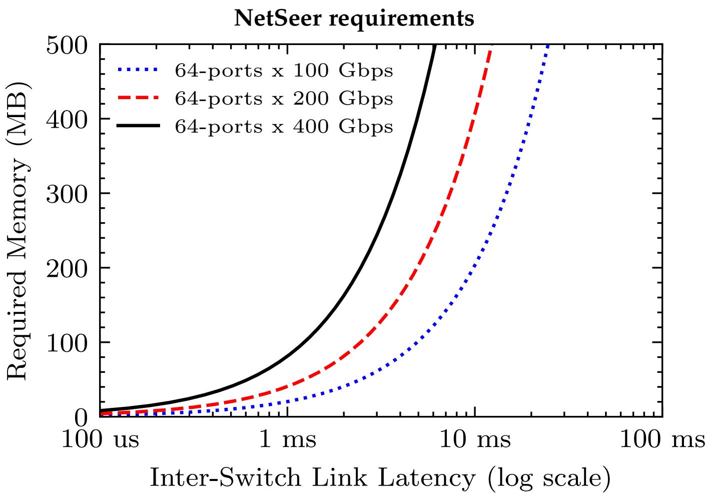


FIGURE 3.8: The required memory for NetSeer to be operational increases proportionally as the delay or bandwidth increases. The plot considers 1024B packets and 13 bytes per packet.

It's important to note that NetSeer's limitations are not easily addressed in the future. This is because traffic forwarded by ISPs is expected to increase over the years at a much faster rate than the growth of hardware resources (e.g., memory) in switches. This trend would make NetSeer less and less suitable for future ISPs.

In this section, we have explored advanced detection mechanisms, extending our analysis beyond vendor-specific methods. These detection mechanisms range from simple packet counting to sophisticated solutions that leverage end-host capabilities, external controllers, or programmable data planes. Despite their potential, these methods face challenges in high-bandwidth ($\geq$ 100 Gbps) and high-delay ($\geq$ 1 ms) environments like ISP networks. These findings highlight the need for a solution capable of operating in such challenging environments. Consequently, in the next chapter, we introduce FANcY, our novel gray failure detection system specifically designed to function effectively within ISP network environments.

4 FAST IN-NETWORK GRAY FAILURE DETECTION FOR INTERNET SERVICE PROVIDER NETWORKS

In this chapter, we present FANcY, a fast in-network gray failure detection and localization system aimed at high-bandwidth and high-delay networks such as Internet Service Providers (ISPs). FANcY complements previous gray failure detection systems, which are mainly tailored for low-delay networks such as data center networks and are not suitable for ISP networks.

ISPs serve as the backbone of our increasingly interconnected society. Thus, ensuring the reliable and efficient transmission of data packets is not only a technical requirement but an economic imperative. Even at marginal rates, packet losses can significantly impact the quality of Internet services [16]. Thus, avoiding packet loss is so critical to ISPs that research and industry efforts focus increasingly on ensuring minimal downtime upon failures (e.g., [3, 124, 125]). A major result of past efforts is that "hard" failures affecting all packets crossing a link or node are typically detected quickly, thanks to existing vendor detection techniques (see Section 3.2.1) such as the BFD protocol [89].

In practice, however, malfunctioning hardware often causes packet losses only for subsets of packets sent over a link. As introduced before (Section 3.1.1), in this dissertation, we call a *gray failure* any hardware malfunction that causes non-transient packet loss on a subset of the traffic forwarded by any packet-forwarding device – which we generally call a switch. Consistent with our definition, we do not classify congestion as a gray failure.

Table 4.1 shows representative examples of device bugs causing gray failures. As shown in the table, malfunctions might cause random packet drops or packet blackholes for one or some packet types (e.g., same prefix or port) or even for all types. Additional examples include misplaced line cards and bent or dirty fibers [16].

Our survey (see Section 4.1.1) confirms that ISP operators consider gray failures a significant concern and lack effective techniques to detect and locate them. Indeed, they often become aware of gray failures only when customers complain about the failure-induced packet loss, which they end up troubleshooting for days or weeks.

As described in the previous chapter, in Section 3.2.2, existing state-of-the-art techniques are ineffective because detecting and localizing gray failures in ISP networks requires analyzing *all* the traffic, which is something they are not capable of doing. Hello-based protocols, including Bidirectional Forwarding Detection (BFD), do not work because most gray failures do not impact messages from these protocols. Packet and flow counter tools, such as NetFlow [28] or sFlow [13], do not help either. They rely on random packet sampling for scalability, and are unable to support fine-grained traffic analyses (as also shown in [15]), which would be needed to spot gray failures.

Gray failures are not specific to ISPs, and recent research aimed at developing gray failure detectors for data center and cloud networks. Those detectors' designs, however, do not match the peculiarities of ISP networks: they either require control of end hosts [11, 43, 93, 98], assume low packet loss rates and extremely high-speed interfaces between the control and data planes (e.g., [42]), or require limited links delay and traffic volumes (e.g., [17]).

Finally, mechanisms internal to switches, such as deflection on drop [17], do not capture several failure cases, including those where the drop flag is not correctly set on packets because of memory corruption, and link-level failures.

Therefore, in this chapter, we introduce FANcY, a system that can locate and detect gray failures in ISP networks.

Vision. We aim at designing an accurate and fast gray failure detector for ISPs. Similar to BFD, the immediate application of such a detector would be to support *selective fast rerouting on gray failures* – i.e., rerouting traffic only for the disrupted traffic, as fast as possible. We also envision that in the future, a gray failure detector may assist operators in finding the root cause of gray failures, and enable new control- and data-plane applications, such as automated failure repair through ad-hoc forward error correction mechanisms or hardware reconfiguration (e.g., [126]).

Problem statement. We focus on the following question:

> *Can we build an ISP-targeted system able to detect and localize intra-domain gray failures in seconds?*

Real examples of ***unwanted*** traffic drops affecting...	...***some*** packets	...***all*** packets
...***one*** or ***some*** IP prefixes	Neighbor Solicitation [127] or BGP [128] packets	Packets sent from a specific line card [129] Specific IP prefixes [130]
...***all*** IP prefixes	With specific sizes [131] With IP ID field `0xE000` [132] With wrong CRC [135, 136]	Traffic on certain ports [133, 134] Interface flaps [137, 138]

Table 4.1: Representative examples of gray failures plaguing major routing devices (from Cisco and Juniper bug reports).

By localizing, we mean identifying both the switch port suffering from a gray failure and the affected traffic. Note that our problem statement does not directly target root cause analysis, nor automated remedies to the detected failures.

FANcY. We present FANcY, a gray failure detector tailored to ISPs. FANcY relies on an inter-switch protocol enabling data planes to synchronize packet counters and detect packet losses by comparing the values of those counters. Counters provide the minimal information needed to localize gray failures; frequently exchanging them provides detection speed and scalability (e.g., consumed memory).

The chapter is organized as follows. In Section 4.1, we show the impact of gray failures in ISP networks and why existing solutions are not applicable. In Section 4.2, we provide an overview of FANcY, giving a high-level idea of how it operates, its inputs and outputs. In Section 4.3 we dive deep into FANcY internals; its reliable protocol, counter exchange synchronization, and the different types of counters giving an emphasis on hash-based trees and their parametrization. In Section 4.4, we do a sensitivity analysis using real traffic traces to select the best hash-based tree parameters to use in our evaluation. In Section 4.5, we evaluate FANcY's ability to capture gray failures through extensive simulations using our ns-3 [139] implementation. Since FANcY is traffic-driven, we first assess the minimal traffic requirements that allow it to quickly and accurately localize failures. We then experiment with real traffic traces, and confirm FANcY's potential to work well in real ISPs. In Section 4.6, we introduce FANcY's prototype implementation in P4 on an Intel Tofino Switch. We use this implementation to demonstrate that FANcY enables sub-second selective fast rerouting. Finally, in Section 4.7, we conclude the chapter with a brief summary.

4.1 GRAY FAILURES IN ISP NETWORKS

We now revisit why detecting and localizing gray failures in ISP networks is an important and open research problem. To confirm this, we first summarize the main relevant findings from the operators survey introduced in Section 3.1.3.2 and the analysis of hardware bugs and the different ways they affect traffic, as shown in Section 3.1.2. Finally, we detail why prior work falls short in detecting gray failures in ISP networks using today's hardware.

4.1.1 *Gray failures are a problem for a majority of operators*

The key takeaway from the anonymous survey, which we detail in Section 3.1.3.2, is that gray failures are a common problem in ISP networks. Most operators ($\approx$90%) have to deal with these issues quite often. For some, it is a daily problem, and for almost half, it is at least a monthly issue. The real challenge is not just how often these problems happen but also how hard they are to find and fix, taking anywhere from hours to weeks.

Most operators say they need better tools to detect gray failures. In fact, in our survey, we show that 74% of them do not have any specialized tool for this, highlighting an acute need for practical solutions. Furthermore, operators often find out about gray failures through customer complaints, suggesting that many of these failures may go unnoticed.

Another key finding from our survey is that when discovered, gray failures are difficult, time-consuming, and frustrating to debug. The absence of specialized tools adds to the difficulty, often requiring manual step-by-step analysis. Additionally, since gray failures tend to be hardware-specific, which require the involvement of vendors, further delays the process.

4.1.2 *What is the impact of gray failures in ISPs?*

To further assess the impact of gray failures in ISPs, we refer to the analysis of vendor bugs shown in Section 3.1.2.

We classify gray failures according to *(i)* the affected forwarding entries (i.e., all or some IP prefixes) and *(ii)* the affected traffic (all or some packets per affected entry). Our classification focuses on the effects of the gray failures (i.e., *what* is dropped, which is visible to operators), rather than their causes (i.e., *why* packets are dropped, which is usually harder to estimate and known by vendors only).

Table 4.1 lists a representative selection of examples of gray failures for each class. It shows that gray failures come in all shapes and forms, some leading to complete blackholes while others induce drops of very specific packets only (i.e., affecting one or a few entries). You can find the full list of bug reports, its estimated root cause, and a short summary in Appendix A.1.

Our survey (Section 4.1.1) confirms that our classification is representative: operators state that they have observed at least one gray failure of each type. For more details about our bug study, refer to Section 3.1.2.

4.1.3 *Why is prior work not applicable in ISPs?*

We now discuss why prior gray failure detection approaches do not work in ISPs, and motivate the need for a new in-switch design.

In Section 3.2.2, we detailed all the different types of advanced failure detection techniques, we classified them by approach (i.e., host-based) and technique (i.e., passive monitoring of TCP stats) and explained their strengths but also why most of them are not suited for ISPs.

In this section, we will revisit that idea and explain in general terms why most existing solutions do not work in ISP networks and what are the requirements to detect gray failures in them.

In general, to detect and localize gray failures between two points, we have two requirements: (i) to be able to *collect* packet information (e.g., headers, counters, etc.) on *all* the traffic, as any packet might be lost, and

(ii) a mechanism to compare that packet information and communicate between the two collection points.

Existing ISP monitoring techniques do not collect statistics on *all* the traffic. Heartbeat-based protocols such as BFD can only detect failures affecting the heartbeat packets, solutions that use probe-based [11, 98] mechanisms are only able to detect problems if probe packets get impacted. Packet or flow-based counter mechanisms either monitor only basic things such as port counters (e.g., SNMP) or need to use packet sampling in order to keep operating at line rate [13, 28]. Similarly, packet mirroring techniques [14, 15] can only afford to mirror slices of traffic due to the impossibility of mirroring the aggregated traffic of today's networks.

Existing data center solutions fall short in ISPs. While some data center gray failure detection techniques *collect* packet statistics on all the traffic and do some sort of *comparison* or *analysis* on them, they cannot operate in ISPs networks. We can easily discard all end-point based solutions, as ISP operators have either non or very limited end-host control. This leaves us with either in-network controller-centric approaches or in-network computation-centric approaches. Given today's typical link bandwidths (e.g., $\geq$ 100 Gbps), these systems can only manage to fulfill the requirements in low latency networks, such as data centers, where link delays can be as low as a few microseconds. In contrast, in ISPs, the latency between devices is in the order of ms (as shown in Section 3.1.3.1), while also having very high-bandwidth links of 100 Gbps and constantly increasing. In fact, with consumer demands continually rising, especially for video content, it is expected that service providers and cloud providers adopt Ethernet technologies reaching 1.6 Tb/s [140]. The evolving characteristics of current and future ISPs make failure detection systems designed for data centers impractical in ISP networks.

To understand the significant impact of link bandwidth and device latency on the memory requirements for a gray failure detector, let's revisit the two aforementioned requirements. To successfully detect gray failures, we need a system that is able to first collect or store packet information for all the packets, and after some time, compare the information either directly in the data plane or in a control plane. Therefore, the memory required by switches during the collection phase depends on the collection rate, the collection complexity, and the minimum required time to perform a comparison or export the collected data. In more detail:

Collection rate. The amount of packets per unit of time for which the switch needs to store information. The collection rate directly depends on the link bandwidth and average packet size.

Collection complexity. The amount of information bits that need to be stored in the switch per packet. The collection complexity depends on the detection technique used. For example, NetSeer's [17] collection complexity is the packets 5-tuple (13 bytes or 104 bits).

Minimum compare time. The minimum amount of time the detection system must be collecting data before it can free its memory. For systems that use a controller, this is determined by the device-to-controller latency and reading speed. For in-network systems where devices exchange information or control signals directly in the data plane, the compare time is bound by the link delay.

On the one hand, controller-centric approaches capable of detecting gray failures, such as LossRadar [42], which already face limitations due to the control plane reading speed (see Table 3.4), would be further limited by the typical ISP delays between devices and controllers. Note that increasing the allocated memory to compensate with these delays, would paradoxically increase the reading time further, exacerbating the initial problem.

On the other hand, in-switch designs such as NetSeer [17], which is also able to efficiently detect gray failures in data center networks, the high collection rate and increased delay between devices makes it completely impractical in ISPs: the required memory by the inter-switch packet loss detection exceeds by a great margin the available memory in today's switches (see Figure 3.8).

These limitations are unlikely to be easily resolved in the near future. In fact, in the next years, the amount of traffic forwarded by ISPs is expected to significantly increase outpacing the growth in switch memory capacity and controller-to-device reading speeds [140]. Additionally, the delay between devices in an ISP is bound to their physical distances which remains rather constant. Thus, in order to detect gray failures in ISP networks we need a system whose required memory does not fully depend on the traffic rate and minimum comparison time.

4.1.4 *What about simple designs?*

Based on our observations in Section 4.1.3, we conclude that an effective gray failure detector for ISPs should operate in-switch, avoid sampling flows, and minimize the duration of storing per-packet information. At a glance, it may seem that we can use simple designs matching those constraints by just exploiting the ability of switches to count packets. Unfortunately, this is not the case.

First, we cannot count traffic at per-link granularity: this simply does not provide enough information to localize the gray failure. Also, we cannot sample traffic to count; otherwise, gray failures affecting small fractions of traffic would probabilistically take a long, possibly indefinite time to be even detected. Similarly, we cannot count traffic only for some entries because gray failures can impact one or a few entries that we do not know in advance – see Table 4.1.

Conceptually, we instead need to count *all* packets for each traffic entry. Once again, however, simply having one counter per entry does not work in ISPs, as it exceeds the memory available on the switches. For example, if we consider entries to be all the /24 IPv4 prefixes ($\approx$ 16 M), covering the Internet routing table with a conservative estimate of 32 bits per counter would require about 512 MB per port. This naive solution would require an order of magnitude more memory than SRAM available in today's switches to cover gray failures for only one port.

In the next Section 4.2, we present the design of our in-network gray failure detection system. This system aims at scaling per-entry packet counters and circumventing the problems existing solutions face in high-bandwidth and high-latency environments. Additionally, our design addresses practical challenges, including distinguishing gray failures from transient drops, such as those caused by congestion, and ensuring synchronized packet counting across switches.

4.2 FANCY OVERVIEW

Figure 4.1 illustrates FANcY's interface and its role within our envisioned in-network reaction approach. FANcY takes two inputs: (i) the specification of entries that the operator or applications using FANcY want to monitor, and (ii) the memory budget per switch. Whenever FANcY detects packet

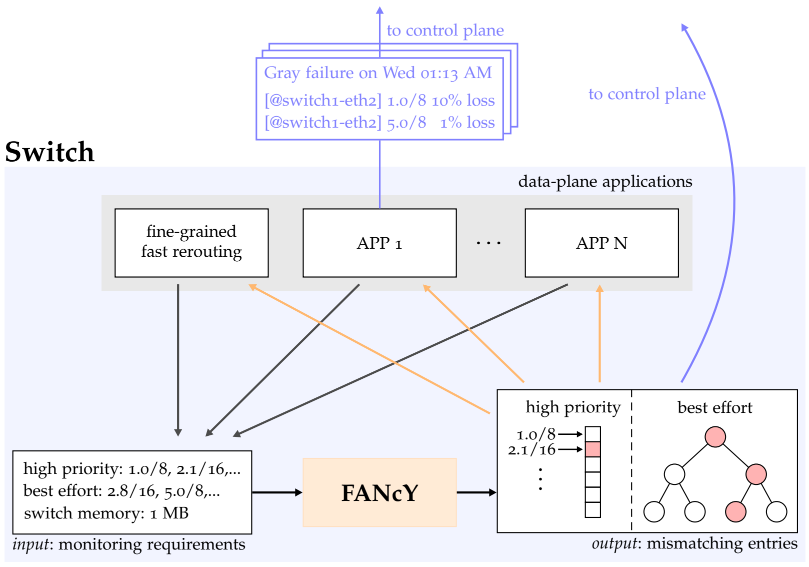

to control plane
Gray failure on Wed 01:13 AM
[@switch1-eth2] 1.0/8 10% loss
[@switch1-eth2] 5.0/8 1% loss
to control plane
Switch
data-plane applications
fine-grained fast rerouting
APP 1
...
APP N
high priority
best effort
1.0/8
2.1/16
high priority: 1.0/8, 2.1/16,...
best effort: 2.8/16, 5.0/8,...
switch memory: 1 MB
FANcY
input: monitoring requirements
output: mismatching entries

FIGURE 4.1: High-level view of a FANcY switch.

drops induced by a gray failure, it flags the entries and ports affected by the failure.

In FANcY, an *entry* indicates a subset of the header space defined by a match rule on packets. For example, Figure 4.1 shows that operators can specify destination prefixes as entries, which would be reasonable if they aim to support selective fast rerouting in destination-based routed networks. However, we remark that future applications can dynamically define the entries monitored by FANcY, for example, for root cause analyses – e.g., to assess losses per packet size or per value of specific IP fields.

Since fundamental limits constrain how many entries can be monitored with the limited memory available on switches, FANcY offers two levels of accuracy for entries to be monitored: high priority, and best-effort. Each high priority entry is tracked with a *dedicated counter*. Best-effort entries are collectively monitored with a *hash-based tree*. Contrary to existing sketches, the hash-based tree stores aggregated counters without compressing information, and is decoded *in hardware*, at runtime, to identify faulty (i.e., experiencing gray failures) entries, in-switch and at line rate.

FANcY's interface, for example, allows operators to monitor all destination prefixes, while also maximizing accuracy and reactivity for the ones

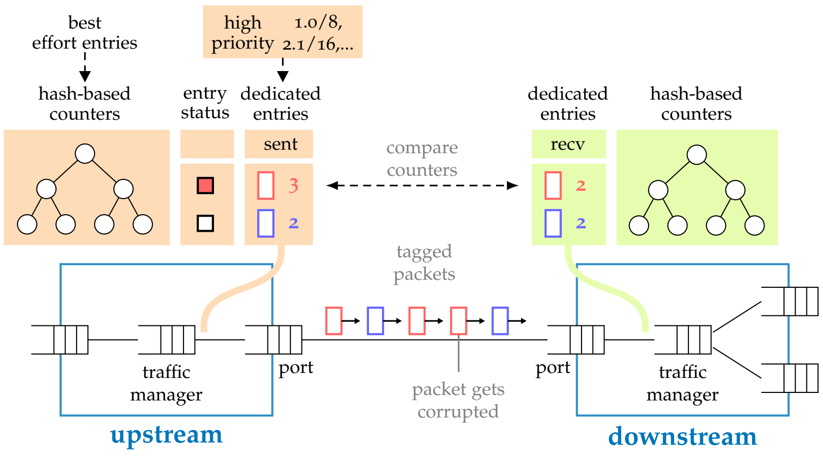


Figure 4.2: High-level view of a FANcY switch to switch running the counting protocol.

driving most Internet traffic, which are typically few [141]; we assume this to be a common goal for ISP operators. If operators want to monitor a more limited set of entries, they can also specify all entries as high priority. The system returns an error, if the set of high-priority entries cannot be supported with the memory budget specified in input.

Figure 4.2 shows a pair of switches running FANcY's counting protocol. As shown, FANcY works at a per-link granularity, reporting losses separately for each switch port. To detect and localize gray failures affecting input entries, each upstream FANcY switch sending packets to a downstream FANcY switch establishes synchronized counting sessions with the downstream. A new session is opened as soon as the previous one closes.

During each counting session, the upstream tags packets to be counted by the downstream with an identifier of the counter to be increased, so that both switches consistently count the same subset of packets with the same counters.

At the end of each session, the downstream sends back its counters to the upstream, which compares the counters and starts a new session immediately after. When it detects discrepancies between its counters and the downstream ones, the upstream switch flags the mismatching counters by populating local registers.

FANcY counters are carefully positioned to avoid recording packet loss due to congestion. Within any switch, congestion typically occurs at the traffic manager (TM), which implements the actual switching logic – i.e., redirecting packets from the ingress pipeline to the configured egress pipeline. In FANcY, as shown in Figure 4.2, packet counters are therefore placed after the TM of the upstream switch and before the TM of the downstream one.

We designed FANcY's counting protocol to be resilient to packet loss while also using minimal memory on switches. To provide good accuracy for best-effort entries, we rely on a zooming algorithm that allows switches' data planes to dynamically explore hash-based trees at runtime. This reduces FANcY's memory consumption on switches, thus allowing each switch to maintain counting sessions with all its downstream switches. We detail the design of FANcY internals in Section 4.3, and analyze its accuracy, speed, and resource consumption in Section 4.5 and Section 4.6.

4.3 FANCY INTERNALS

We now describe the most important FANcY components: the counting protocol (Section 4.3.1), the hash-based tree data structure (Section 4.3.2), some hash-based tree properties (Section 4.3.3), and the system interface and deployment details (Section 4.3.4).

4.3.1 *Counting protocol*

When designing FANcY's counting protocol, we need to balance accuracy (i.e., how often we count packets), reliability (i.e., how to guarantee that counters are successfully exchanged), and scalability (i.e., how much memory is needed on switches). We first show that maximizing accuracy leads to high memory consumption and sub-optimal reliability. This motivates us to trade some accuracy for much better reliability and scalability.

Strawman: continuous counting with in-packet session IDs. Ideally, we would like to continuously count all the packets at the upstream and downstream switches. To achieve that, the upstream can tag packets with a session ID, and start a new session by just changing the packets' tag. For example, increasing it by one for a new session. Upon receiving a packet with a different tag, the downstream would then send its counters back to the upstream.

Unfortunately, this counting approach requires the upstream switch to allocate memory for at least two sets of counters, respectively, for the current and previous sessions. The upstream indeed has to wait for the counters from the downstream switch before it can check for packet drops in the previous counting session. In addition, the above protocol does not achieve reliability. If a counter sent by the downstream is lost, all the measurements for that session are also lost – i.e., a link cannot be monitored if a failure affects the reverse direction of the traffic. To ensure reliability across k sessions, both the upstream and the downstream must then keep $k-1$ historical counters' values, and consume k times the memory required for a single session. Furthermore, when using such counting protocol, packet reordering would trigger unwanted counting events – i.e., the downstream sending a counter before the session has ended. Such premature counter transmissions from the downstream would result in counter mismatches, undermining the reliability of the counting mechanism.

FANcY protocol. To achieve reliability with minimal memory, FANcY adopts a protocol akin to stop-and-wait. In FANcY, every counting session is opened by the upstream switch through a *Start* control message, and closed after a *Stop* message. After sending a *Start* (resp. *Stop*) message, the upstream switch waits for a *Start ACK* (resp. *Report*, including the downstream counters) response from the downstream switch, and it keeps retransmitting *Start* (resp. *Stop*) messages if it does not receive responses before a timeout.

At any time, FANcY's counting protocol requires storing a single set of counters, for the current session, at both the upstream and downstream switches. Furthermore, FANcY's stop-and-wait protocol ensures that both counting ends are perfectly synchronized counting packets for the same session without needing the intervention of the control plane. Its downside is that counting is stopped when control messages are exchanged. The time not counting is directly dependent on the delay between the upstream and downstream switches. We make this choice because FANcY focuses on systematic packet drops (e.g., see Table 4.1), and hence stopping counting for short times may affect detection speed, but does not prevent us from detecting gray failures, as Section 4.5 confirms.

An important parameter of FANcY's counting protocol is the frequency of counters' exchanges. This parameter influences the accuracy, the detection

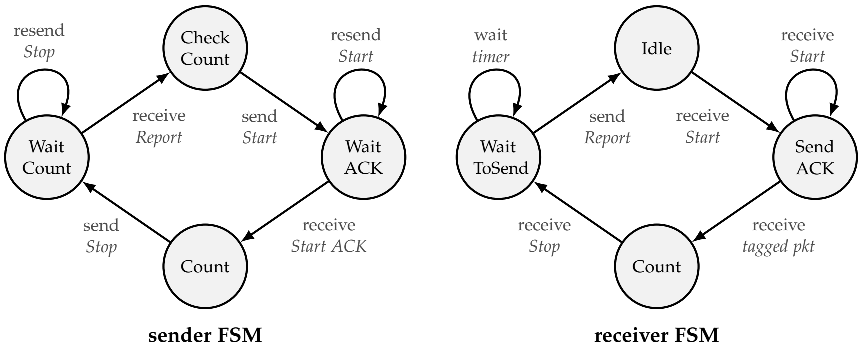


FIGURE 4.3: Finite state machines run on any pair of upstream (left) and downstream (right) FANcY switches.

speed, and the overhead in terms of additional control traffic generated by FANcY. We discuss reasonable counters' exchange frequency values in Section 4.5.

FANcY Finite State Machines (FSMs). FANcY switches implement their counting protocol by running FSMs directly in the switches' hardware. We now detail FANcY's FSMs. We further describe the implementation and evaluation of those FSMs within an Intel Tofino switch in Section 4.6 and Section 4.6.3.

Let A be an upstream switch, and B be the downstream one. To detect losses of packets sent by A to B, A implements FANcY's *sender FSM*, while B runs the *receiver FSM*. Figure 4.3 displays both FSMs, and Figure 4.4 illustrates how the FSMs transition from one state to another during a typical counting session.

To start a new counting session, A resets all its counters for the $A \rightarrow B$ link, and sends a *Start* message to B. Since it is critical that A and B start counting from the same packet, A then enters the `WaitACK` state where it doesn't increase any counter but waits for an acknowledgment from B. When it receives a *Start* message, B indeed resets its counters, and replies with a *Start ACK* message. The *Start* phase in FANcY is essential for synchronizing the FSMs states and enabling an efficient reset of hardware data structures without the need for external controllers. This critical synchronization step ensures both reliability and operational efficiency in FANcY's FSM processes.

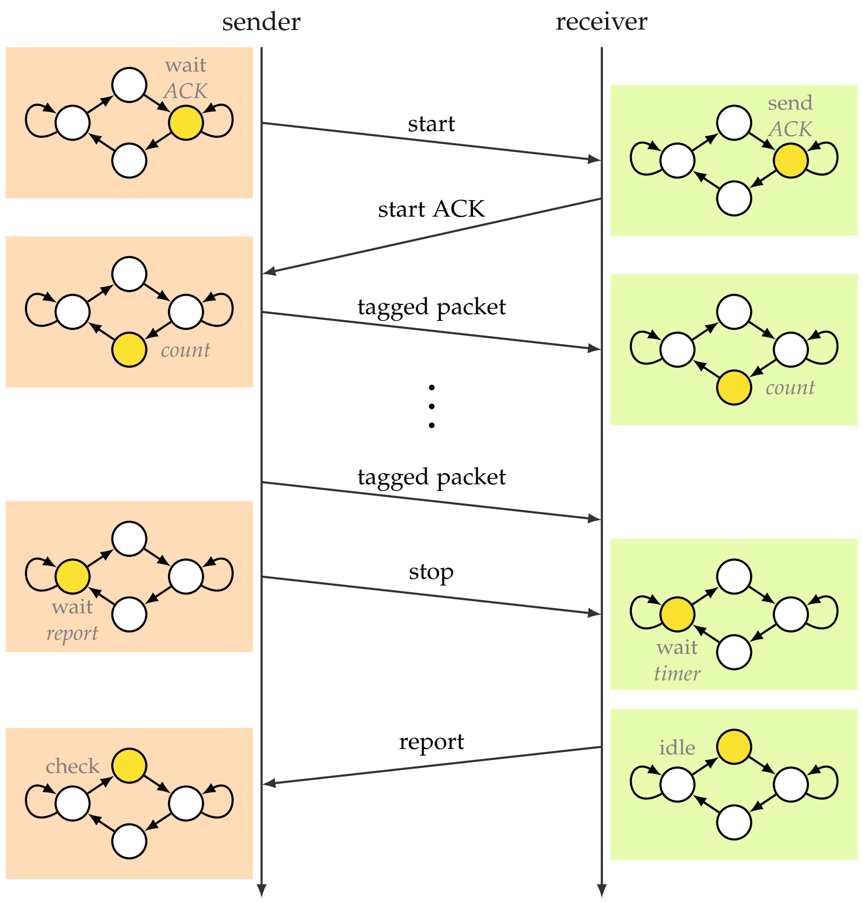


FIGURE 4.4: Time sequence diagram showing the implementation of a counting session with FANcY state machines.

If after a given time T_{rtx}, A does not receive a *Start ACK*, it sends the *Start* message to B again. If A does not receive responses from B after X attempts (with X=5 by default), A reports a link failure.

Upon successfully receiving a *Start ACK*, the sender FSM transitions to the `Counting` state, where A counts and tags each packet it sends over the $A \rightarrow B$ link. The first tagged packet received after the *Start* message makes B transition to its own `Counting` state.

Packets are tagged by A and counted by B until A sends a *Stop* message to B. At that point, A moves to the `WaitCounter` state until it receives a *Report* message from B, with B's counters.

A stop-and-wait approach, similar to the one implemented for the session setup, is used to work around possible losses of *Stop* and *Report* messages. In contrast to session opening, however, the downstream switch does not send the *Response* message immediately after receiving the *Stop* one. Upon receiving such a message, the receiver FSM indeed transitions to the `WaitToSendCounter` state, where it can keep counting tagged packets for a short time interval T_{wait}. This timeout accounts for delayed or reordered packets. In theory, it should not be possible for packets to be reordered if they follow the exact same path from the sender FSM to the receiver one – e.g., if A and B are neighbors. We keep the `WaitToSendCounter` state in the receiver FSM to avoid making assumptions on the path from A to B. After T_{wait}, the receiver FSM sends the counter back to the upstream, which upon successful reception, checks for counter discrepancies, and if needed reports the error or flags the entry as faulty. If no counter report is received at A after T_{rtx}, A sends a *Stop* message to B again. As before, if A does not receive responses from B after X attempts (with X=5 by default), A reports a link failure. Finally, upon a successful counter exchange and checks, A starts a new counting session.

So far, we have shown FANcY FSMs primarily for synchronizing and exchanging single packet counters between switches. However, the versatility of our FSMs extends beyond this singular use. We can generalize their application to facilitate the exchange of more advanced data structures, enabling the implementation of more complex algorithms. In the next Section 4.3.2, we show an instance of that. It's important to note that extending FSMs for exchanging information beyond packet counters only requires to tweak the semantics that switches associate to packet tags, and adjust the content of the *Report* messages.

4.3.2 *Hash-based trees*

In the previous section, we introduced the FANcY protocol. Its dedicated counter version uses a pair of counters to monitor traffic entries individually, a method that's simple and effective but only suitable for high-priority traffic due to its poor scalability. For instance, tracking 1 million entries

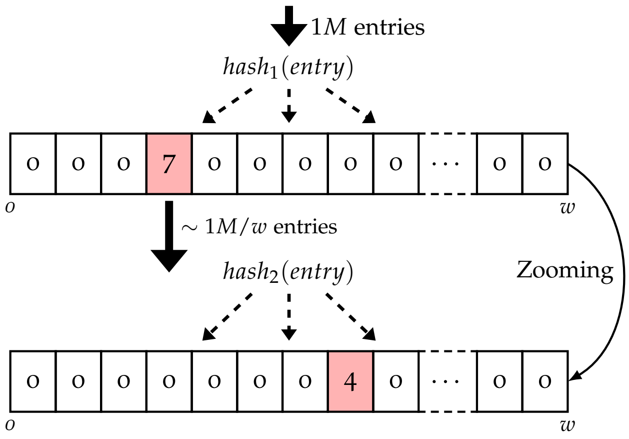


Figure 4.5: Two-level hash-based counter array illustrating the distribution of 1 million entries into w cells and the subsequent zooming on mismatched counters to further isolate aggregated entries at the first level.

requires 1.25 GB of memory for a 64-port switch, with each pair of state machines and counters consuming 160 bits.

Recognizing that gray failures are typically sparse – the majority of entries often do not experience errors at the same time – we identified an opportunity for an efficient data structure that instead of using dedicated resources for each traffic entry, which can be inefficient, aggregates multiple traffic entries into one counter. This approach optimizes memory usage and is realized through hash-based counter arrays, akin to counting bloom filters [142]. To count with these arrays we use a hash function on the traffic entry identifier, such as an IPv4 prefix, to determine which counter index cell belongs to each traffic entry. Like in the dedicated counter FANcY protocol, packets are counted in a hash-based array at both upstream and downstream switches. The counts are exchanged and compared at the end of every counting session. However, this time, cells with mismatching counters in the arrays can represent multiple traffic entries, necessitating a multi-round approach to accurately pinpoint the specific faulty entry or entries. Figure 4.5 illustrates the process for a two-level hash-based counter array example monitoring 1 million entries.

Although hash-based arrays provide a scalable way to detect gray failures, they can only be used to zoom in one cell at a time. To solve that, in FANcY

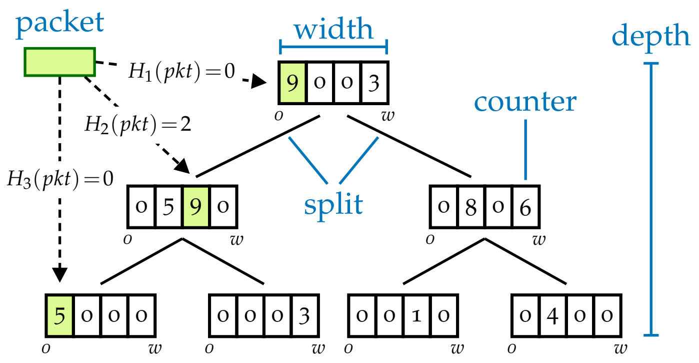


Figure 4.6: An example of a hash-based tree implemented within FANcY switches: each node is an array of counters, and packets are mapped to counters at each level through a level-specific hash function.

we introduce the hash-based tree data structure. In FANcY hash-based trees, each tree level stores an array of counters associated with a subset of best-effort entries. Counter arrays at higher levels of the tree map to larger sets of entries, while the tree's leaves map to one or few entries.

Utilizing hash-based trees allows for improved accuracy and scalability at the cost of slower detection speed, proportional to the tree's levels. This is particularly effective for large numbers of entries, where a single counter array would either be too large or prone to collisions and false positives. By employing hash-based trees, switches can dynamically explore and zoom in on counters at lower levels upon detecting mismatches at higher levels. In the following, we dive into the details of the hash-based trees' data structure and the zooming algorithm.

Hash-based trees' data structure. Figure 4.6 pictures a small hash-based tree as implemented in FANcY switches. Its nodes are fixed-size arrays of counters. Each counter is mapped to a specific set of packets through hash functions. To better define which packets increment which counters, we first introduce some terminology.

Any FANcY hash-based tree is a balanced k-ary tree, characterized by three parameters: width, depth, and split. For a given tree, its *width* w is the number of counters per node, its *depth* d is the length of any path from the tree's root to a leaf, and its *split* k is the number of children per node. For example, $w = 4$, $d = 3$, and $k = 2$ for the tree in Figure 4.6.

Every packet belonging to a best-effort entry maps to one counter per tree's level through a distinct hash function per level, as also shown in Figure 4.6. Consider a counter c_i. A packet p is mapped to c_i if and only if $H_j(p) = i$, where H_j is the hash function applied at the level j to which c_i belongs, and $H_j(p)$ is a value between 0 and $w - 1$ obtained by applying H_j to the fields of p used in p's entry (e.g., the destination address in destination-based routing).

We define the *hash path* of a packet as the list of counter IDs the packet maps to, ordered from the root to the leaf. For instance, $\boxed{0} \rightarrow \boxed{2} \rightarrow \boxed{0}$ is the hash path of the packet shown in Figure 4.6.

Note that any sequence of counters at consecutive levels forms a *partial hash path*, and corresponds to a number of entries inversely proportional to the length of the sequence: the shorter the sequence, the bigger the number of associated entries.

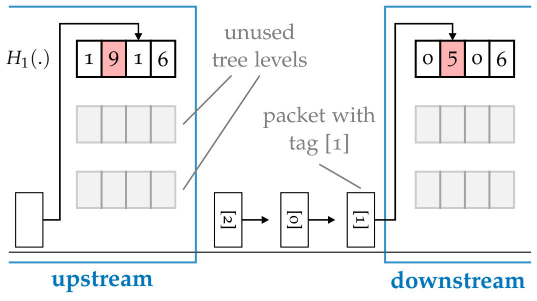
FANcY zooming algorithm. To detect best-effort entries affected by a failure, this algorithm incrementally builds partial hash paths of increasing length for counters affected by a failure. As described before, at every iteration, the algorithm indeed increases by one the length of the partial hash path affected by packet loss, and hence it reduces the set of candidate failed entries.

To understand the zooming algorithm better, let us consider a pair of FANcY switches. Assume for now that the switches maintain trees of split 1, a counter array of width 4 and a depth of 3, as shown in Figure 4.7.

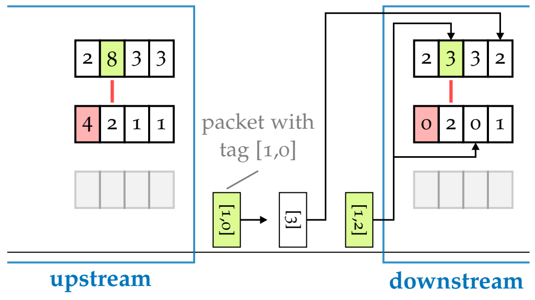
In the absence of losses, the two switches only update root-level counters. During each counting session, the upstream switch tags every packet with the index of the counter to which the packet maps according to the root-level hash function (h_1), and the root-level counters are consistently updated on both switches, as displayed in Figure 4.7a. At the end of the session, if no drops have happened, the upstream switch detects the congruence of its counters with the downstream ones, and starts a new session.

Suppose now that a gray failure occurs and packets for a specific traffic entry start to get dropped. At the end of the session, the upstream switch checks its counters against the downstream ones. If it detects mismatches for more than half of the counters, it flags the failure as a uniform random one – i.e., "localizing" it to all entries. Otherwise, the switch computes the root-level counter c_i with the maximal difference between the local and the

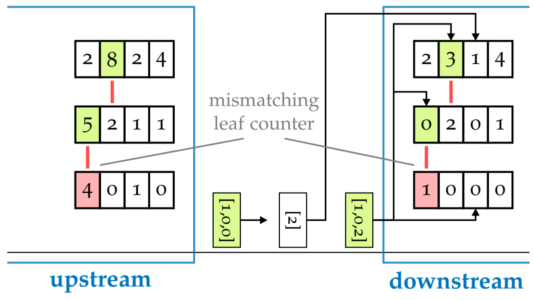
Hash-based tree zooming stages



(a) before any counter mismatch



(b) counting session after first zoom



(c) successful zooming completion

FIGURE 4.7: Illustration of the zooming algorithm on hash-based trees of width 4, depth 3 and split 1. The algorithm zooms into one tree layer at each stage until possibly detecting a mismatching counter at the leaf layer.

downstream values.[1] In our example, after the first counting session, the upstream computes the counter difference as: $\boxed{1}\boxed{9}\boxed{1}\boxed{6} - \boxed{1}\boxed{6}\boxed{1}\boxed{6} = \boxed{0}\boxed{3}\boxed{0}\boxed{0}$, detecting a mismatch of 3 packets at c_1. In the following session, the upstream switch then tags packets if and only if they hash to c_1, effectively zooming in the set of entries with the highest drop rate.

Packet tags carry information about the hash path of the counters packets map to. This way, the downstream switch knows which packets to count and which counters to increase without having to hash packets consistently with the upstream. In our example, after detecting a mismatch on counter c_i, the upstream would therefore tag every packet that maps to c_i with the path $[i, m]$, where i is the index of c_i in the root-level node and m is the index of the second-level counter c_m to which the packet is mapped – see Figure 4.7b.

The above procedure is repeated until a leaf node is reached. In our example from Figure 4.7c, after all the displayed packets in the link have been transmitted, the leaf counters difference is computed as: $\boxed{4}\boxed{0}\boxed{1}\boxed{0} - \boxed{2}\boxed{0}\boxed{1}\boxed{0} = \boxed{2}\boxed{0}\boxed{0}\boxed{0}$. At that point, FANcY reports a failure for *every* mismatching leaf counter. This, for example, enables the upstream switch to immediately start rerouting packets whose hash path (i.e., $\boxed{1}\rightarrow\boxed{0}\rightarrow\boxed{0}$) corresponds to any of those counters. Note that FANcY technically detects a failure when it starts zooming in any root-level counter, but reports the failure only after reaching the tree's leaves in order to increase accuracy.

For multi-entry failures, FANcY adopts a pipelining approach that increases failure detection speed. To achieve that, it simultaneously zooms in counters at different levels of the tree. Consider, for example, a failure that affects two root-level counters c_1 and c_2. At the end of the first counting session after the failure, the upstream switch observes packet losses on both c_1 and c_2, and selects the one with the maximum packet difference, say c_1. In the following session, the upstream then instructs the downstream to populate counters at the second level of the tree for packets hashing to c_1 *in addition* to increasing root-level counters. At the end of this second session, the upstream observes again packet loss for both c_1 and c_2. Since it is already zooming in c_1, it starts zooming in c_2 this time. So, in the third session, the upstream and downstream increase root-level counters for all the packets, second-level counters for packets hashing to c_2, and third-level

1 Selecting the counter with maximum losses is instrumental to prioritize failure detection for most traffic. We however envision that future FANcY implementations can take operators' policies into account at this step.

counters for packets hashing to c_1 and the second-level counter with the maximum mismatch in the previous session.

A generalized version of the above algorithm is used in trees with split $k > 1$. At the end of every counting session, the generalized algorithm zooms in k mismatching counters rather than only one. In the presence of multi-entry failures, the algorithm therefore can explore *in parallel* up to k hash paths per counting session, and hence supports the simultaneous exploration of k^{d-1} different paths in d counting sessions.

4.3.3 *Properties of hash-based trees*

n this section, we analyze how different hash-based tree parameters impact accuracy and detection speed. Further, we provide formulas to compute collision probability and memory requirements for generic hash-based trees based on their width, depth, and split.

4.3.3.1 *Parameters analysis*

Since hash paths identify entries affected by failures, the total number and length of hash paths influence the number of entries that can share a counter. Both factors depend on the tree's depth and width, as the number of hash paths is equal to w^d, and their length is bounded by d. Increasing width and depth increases accuracy by making it more likely that leaf counters map to a single entry (see below 4.3.3.2 for details). Increasing depth and width, however, comes at the cost of higher memory utilization (see 4.3.3.3 for details). Additionally, higher depths increase the number of counting sessions needed, slowing down failure detection and making the failure harder to detect, thus potentially decreasing accuracy when there is not enough traffic. As such, width and depth regulate the trade-off between accuracy on one side and memory and detection speed on the other. In Section 4.4, we show how the depth, width, and split affect accuracy and detection speeds with simulations using real traces.

In general, the detection speed for entries mapped to the tree depends on the performance of the zooming algorithm. As described in Section 4.3.2, the algorithm explores up to k^{d-1} paths in d counting sessions. Thus, the detection speed also depends on the split value k: higher split values speed up the detection of multi-entry failures (by a factor proportional to k), but it also requires more memory (i.e., to store a bigger tree) and implementation logic.

The duration of counting sessions, which we also denote as *zooming speed*, affects detection speed, too: it is quite intuitive that shorter counting sessions tend to make detection faster. However, decreasing the zooming speed can also impact FANcY's accuracy, as it reduces the probability of observing packet losses during d consecutive counting sessions. In Section 4.5, Figure 4.10 shows that a low zooming speed might negatively affect detection accuracy (as less traffic is monitored).

4.3.3.2 *Collision probability*

Hash-based tree counters offer a scalable method for monitoring a vast quantity of traffic entries utilizing minimal memory resources. This efficiency, however, comes with the trade-off of potential collisions. Such collisions manifest as false positives for the traffic entries that did not experience losses but have the same hash path as faulty ones.

To determine the collision probability within our system, we can apply the theoretical framework of Bloom filters employing a single hash function, as detailed by Broader et al. [143]. In this context, the number of potential hash paths corresponds to the size of our conceptual Bloom filter. We compute the number of hash paths (m) as: $m = w^d$, where w is the number of cells in a bucket, and d represents the depth of the hash-based tree.

Given that we are solely concerned with collisions occurring in cells that contain faulty entries, the probability of a collision depends on the number of total cells (m) and active faulty entries (n). Consequently, the collision probability (p) can be calculated using the following expression:

$$p = (1 - e^{-n/m}) \tag{4.1}$$

The expected number of collisions (or false positives) is influenced by the total number of distinct traffic entries (t) that cross the hash-based tree while there are faulty entries. This expected number can be formulated as:

$$E(t) = p \cdot t \tag{4.2}$$

Consider a hash-based tree where each bucket contains $w = 100$ cells, and the depth is $d = 3$. This results in $m = w^d = 100^3$ possible hash paths.

Imagine a scenario with 10^5 different traffic entries, among which there is one entry suffering from packet loss.

The probability of a collision, as described by Equation 4.1 with $n = 1$ active faulty entry, is given by:

$$p = 1 - e^{-\frac{1}{m}} = 1 - e^{-\frac{1}{10^5}} \approx 9.999995 \times 10^{-7}.$$

Employing Equation 4.2, the expected number of collisions, $E(t)$, when all 10^5 traffic entries are active while one entry is faulty, is calculated as:

$$E(10^5) = p \times 10^5 \approx 9.999995 \times 10^{-7} \cdot 10^5,$$

which leads to an average of approximately 0.1 collisions during the presence of one faulty entry.

4.3.3.3 *Tree nodes and memory*

The memory footprint of a hash-based tree is determined by its configuration parameters: width (w), depth (d), split factor (k), and the operational mode. Specifically, if the tree operates in pipelined mode—as detailed in Section 4.3.2—FANcY needs to allocate memory for the tree's entire structure. In non-pipelined mode, FANcY optimizes memory usage by allocating only the memory necessary for the last layer of the tree (the tree leaves), which corresponds to the memory needed for the last zooming stage. The memory allocation for tree nodes can be quantified as follows:

- **Pipelined Mode:**

$$\text{nodes}_{\text{p}}(k, d) = \begin{cases} \frac{k^d - 1}{k - 1} & \text{if } k > 1 \\ d & \text{otherwise} \end{cases}$$

- **Non-Pipelined Mode:** $\text{nodes}_{\text{np}}(k, d) = k^{d-1}$.

To compute the total memory required for a hash-based tree (excluding state machine resources, for more info see Section 4.6.2) we only need the product of the number of counter cells (w) at each node, the number of layers in the tree (d), the split (k), the number of bits per counter (b), and the number of nodes ($\text{nodes}(k, d)$). Thus, the memory formula (in bits) is given by:

$$Memory(k, d, w, b) = b \cdot w \cdot \text{nodes}(k, d)$$

If we consider the tree from the example above, with $w = 100$, $d = 3$, a $k = 2$, pipelining and 32-bit counter cells, the required memory for the hash-based tree can be calculated as:

$$\begin{aligned} M(2,3,100,32) &= 32 \cdot 100 \cdot \text{nodes}_{\text{p}}(2,3) \\ &= 32 \cdot 100 \cdot 7 \\ &= 22400 \text{ bits} \end{aligned}$$

Note that this is the memory required for a single tree, FANcY requires two trees per session, one at the upstream and another at the downstream. Furthermore, FANcY installs one pair of trees at each port.

4.3.4 *Practical considerations*

We now describe how FANcY design is instantiated.

Input translation. FANcY switches first allocate one dedicated counter for each input high-priority entry. Each counter occupies 80 bits in total (both at upstream and downstream), including the required state for the counting protocol.

Switches then dimension the hash-based tree based on the input memory minus the amount consumed by dedicated counters. Each node of the tree requires at each side of the session 32 bits times the width of the tree, plus 88 bits to support the counting protocol and the zooming algorithm. Sections 4.3.3.3 and 4.6.2 provide details to compute the number of nodes and total memory required by any given hash tree.

The question then is how to decide the width, depth, and split of the tree, which also influences the number of nodes. In the previous Section 4.3.3.1, we have described how each tree parameter impacts the performance of FANcY. Further, in the next Section 4.4, we show how those parameters impact the performance of FANcY's hash-based tree when detecting gray failures on real Internet traces. Our analysis indicates that setting split to 2 and depth to 3 provides a good trade-off between memory consumption, accuracy, and detection speed. Hence, our FANcY implementation uses those values and adjusts the tree's width based on the available memory. ISP operators can customize FANcY's trees by applying a similar analysis on their networks' traffic traces, requirements and available memory.

FANcY returns an error if the memory needed for dedicated counters and hash-based trees with the above parameters' values exceeds the input memory.

Output. FANcY uses two additional data structures to flag the entries affected by packet loss: a 1-bit register array with one register for each dedicated counter, and a 2-register Bloom filter associated with the hash-based tree. When mismatching values are detected for a dedicated counter, the corresponding register in the 1-bit array is updated. When a counter in the hash-based tree reports a failure, the hash path for that counter is stored in the Bloom filter.

Deployment. FANcY is designed to be deployed at every switch, so that it can monitor all links, one by one; this maximizes accuracy of failure detection and localization.

We however note that FANcY keeps working when deployed at remote switches. In this case, FANcY is able to detect gray failures on the *path* between the two switches[2], although losing the ability to precisely pinpoint the failure location along the path. This enables practical use cases in partial and incremental FANcY deployments. For example, if deployed at the border switches exchanging high volumes of traffic, FANcY provides support for near real-time detection of gray failures along the internal paths carrying most traffic: no tool currently available to ISPs offers a similar capability.

4.4 SENSITIVITY ANALYSIS OF FANCY'S PARAMETERS

In this section, before FANcY's evaluation 4.5, we perform some experiments to show the impact of changing the values of the hash-based tree's parameters. We use FANcY's software implementation (see more details in the next section) and compare different trees using real Internet traces.

As previously described, hash-based trees can detect the vast majority of affected traffic entries (especially those carrying most of the traffic) at scale. The goal of this sensitivity analysis is to determine the impact of increasing or decreasing memory usage on the detection speed and false positives, as well as how much the split impacts detection speeds and true positive rates

2 Note that systematic failures can be distinguished from congestion even in partial deployments of FANcY by monitoring queue sizes on all devices, and discarding all measurements collected during periods where queue sizes were excessively long.

symbol	depth	split	width	nodes	hash paths	memory
●	3	3	205	13	8.6M	1 MB
■	3	2	190	7	6.9M	500 KB
▲	3	3	100	13	1M	500 KB
▼	4	3	32	40	1M	500 KB
◀	3	2	100	7	1M	250 KB
▶	4	2	44	15	3.7M	250 KB
◆	3	1	110	3	1.3M	125 KB
⬢	4	2	28	15	0.6M	125 KB

Table 4.2: Configurations of the eight hash-based tree used for the sensitivity analysis.

depending on the failure burst size (i.e., how many entries fail at the same time). In the next section, we will use what we have learned to pick the parameters of our hash-based trees for FANcY evaluation.

Methodology. We have selected eight different FANcY hash-based tree configurations ranging from approximately 125 KB to 1 MB of required memory for a 32-port switch (see details in Table 4.2). To compare them, we have used the Internet trace with the most active prefixes from the CAIDA dataset ($\approx 560K$, ID 4 4.3) such that we can see how that impacts the number of false positives with respect to the total number of hash paths supported by the tree. During the simulation, which lasts 30 seconds, we fail (100% loss) either 10 or 50 prefixes at the same time. We use multi-failure scenarios to assess better the impact of the tree split on detection speed and True Positive Rate (TPR). The process is averaged over 10 runs, each with a different set of randomly selected prefixes. Note that we only fail prefixes that can be detected at the zooming speed and depth used by the system under test ($\approx 120k$ prefixes). We use the pipelined version of FANcY; Consequently, we need to reserve memory space for all nodes in the tree. As shown in the legend of Figure 4.8, system configuration names are defined as follows: $depth/split/width$ $(Memory)$. For this experiment, we do not use dedicated counter entries, therefore, we allocate 100% of the memory to the hash-based tree.

Figure 4.8 shows the result of our simulation. The upper row pertains to the experiments with 10 failures, whereas the lower row pertains to those with 50. The plots on the left side depict the average mean detection speed (of each run) against the average true positive rate, while plots on the right

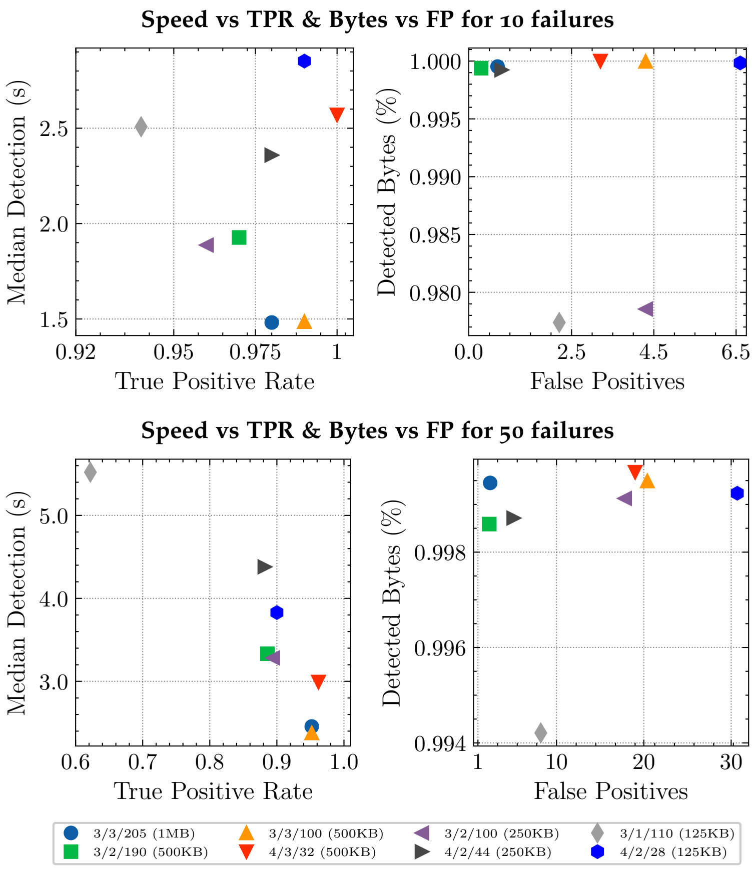


FIGURE 4.8: Comparison of the eight different hash-based tree configurations when 10 (upper) and 50 (lower) failures happen at the same time.

illustrate the average total bytes detected against the average count of false positives. In the following, we describe the main findings.

Split increases TPR and reduces detection speed. Designs with a higher split value have the best true positive rates and lowest median detection speeds, especially in cases affecting many entries. As shown in Figure 4.8, on the left side, the most rapid and accurate designs have a split of 3. This pattern is particularly evident with 50 simultaneous failures (lower left), where the three configurations with a split of 3 (●, ▲, ▼) have the lowest detection times and highest TPR. Conversely, the tree with a split of 1 (◆) has the worst detection speed and the lowest TPR.

In general, depth increases detection time with a slight decrease in TPR. When the number of concurrent failures is relatively low (i.e., 10), trees with greater depth exhibit the highest detection times. This is evident on the top left side of Figure 4.8, where systems with a depth of 4 (▼, ▶, ⬢), or a small split (◆) show the longest detection times. However, in scenarios where numerous traffic entries are impacted, the tree split becomes crucial as it enables parallel failure detection. For example, the detection speed of ▼ relatively improves with respect to other trees under 50 concurrent failure, as seen in the lower left side of the figure. Furthermore, while increasing depth influences the TPR, this increase does not lead to a drastic reduction in the rate.

Memory can be traded by speed without sacrificing TPR or increasing FPs. We have identified cost-effective designs, such as ▶, which achieve a decent TPR, and small FP (due to its 3.7M hash paths) while being one of the cheapest designs. However, that comes at the price of an increased detection time.

Most large traffic entries, which contribute significantly to overall traffic volume in terms of bytes, are detected by all the evaluated trees. As depicted on the right side of Figure 4.8, in both failure scenarios, most systems successfully detect failures for large traffic entries, which account for most bytes. This is normal as all non-detected entries send very few packets.

The number of hash paths directly determines the number of FPs. As previously explained and detailed in Section 4.3.3.2, the number of false positives in our system is directly influenced by the total number of hash paths supported, w^d. Consequently, trees with a higher number of hash paths, such as ●, ■ and ▶, as listed in Table 4.2, tend to have fewer false positives.

4.5 EVALUATION

We evaluate FANcY against its goal of detecting and localizing gray failures accurately, quickly, and scalably. Since FANcY is a data-driven solution, its accuracy and detection speed depend on the amount of traffic it receives for the entries affected by gray failures. We therefore assess FANcY's performance depending on the packet loss rate per disrupted entry. We do not compare against gray failure detectors proposed in previous work because they are incompatible with the network characteristics of ISPs, as already detailed in Section 4.1.3.

To evaluate FANcY, we simulate different gray failures and measure FANcY's accuracy and speed to localize each of them. We use synthetic traffic to quantify the minimal requirements for FANcY to properly work (Section 4.5.1), and CAIDA traces [144] to evaluate the system-wide performance on real traffic (Section 4.5.2). Finally, we analytically assess FANcY's scalability in terms of traffic overhead (Section 4.5.3).

In our evaluation, we consider a 64-port FANcY switch which is given the following input: high-priority entries covering the 500 prefixes driving the most traffic, best-effort entries for all the remaining traffic, and memory of 1.25 MB (i.e., 20 KB per port). Accordingly, FANcY uses 500 dedicated counters and a hash-based tree of depth 3, split 2, and width 190.

When evaluating FANcY's accuracy, we mainly refer to its true positive rate (TPR), which is defined as the fraction of the correctly identified failed entries. Hence, the TPR measures FANcY's ability to detect and localize failures. We focus on the TPR because the true negatives are the complement of the true positives in our case, and the false positives (i.e., entries detected as failed despite the fact that they are not) do not depend on traffic conditions. Indeed, the false positive rate (FPR) is always zero for any dedicated counter. Also, for the hash-based tree, the FPR depends on the probability that multiple entries are stored in the same leaf node, and one of them experiences losses; otherwise, it is zero too. This probability is a function of the tree's width and depth (as detailed in Section 4.3.3), and it is very low for reasonably dimensioned trees. In fact, for traffic extracted from CAIDA traces, the average number of FANcY's false positives is 1.1 (resp., 0.59) for 100% (resp., 1%) packet loss in the challenging case of 100 entries failing at the same time.

We measure FANcY's detection speed as the difference between the time a gray failure is introduced in an experiment and the time FANcY localizes it. Note that this is slightly unfair to FANcY as it may have to wait sometime before a packet affected by the failure is received, especially if the corresponding entry drives little traffic or the gray failure has a low packet drop rate per entry.

To show that FANcY works in large ISPs, we set the inter-switch delay to 10 ms in all the experiments. We also experiment with lower link delays, for which FANcY's accuracy slightly increases for low-drop scenarios, and failure localization speeds up. For example, for 1 ms links, detection speed doubles for dedicated counters, and increases by $\approx$ 15% for hash-based trees.

Experiments in this section are packet-level simulations performed with ns-3 [139]. Simulated networks are composed of nodes running our software implementation of FANcY– i.e., $\approx$ 8,000 lines of C++ code implementing a custom ns-3 switch that closely mimics all the data-plane components (parsers, ingress, egress, metadata fields, etc.) of a P4 switch.

4.5.1 *Benchmarking* FANcY

First, we experimentally demonstrate that FANcY requires an amount of traffic per entry which is realistic to assume in ISP networks.

We are especially interested in the *minimum* amount of traffic needed for FANcY to detect different types of failures. We, therefore, evaluate FANcY's accuracy and speed for synthetically generated traffic of increasing size: in separate experiments, we generate traffic with a different number of TCP flows per second and bitrate per flow. All simulated flows have a duration of $\approx$1 second in the absence of losses, and a retransmission timeout of 200 ms. Of course, failures can significantly increase the duration of flows.

Within the first two seconds of each experiment, we simulate a failure by instructing a switch to drop a certain percentage of packets for some or all entries. We then run each experiment for 30 seconds. When we do not detect any failure across all the repetitions of an experiment, we report a TPR of 0 and a detection time of 30 seconds. We repeat every experiment 10 times, randomly changing flows' starting and failure times.

In the following, we first consider gray failures affecting a subset of entries monitored by FANcY, such as in the cases shown in the first row of Table 4.1. We do so separately for the dedicated counters (Section 4.5.1.1) and the hash-based tree (Section 4.5.1.2). We then evaluate FANcY's performance upon

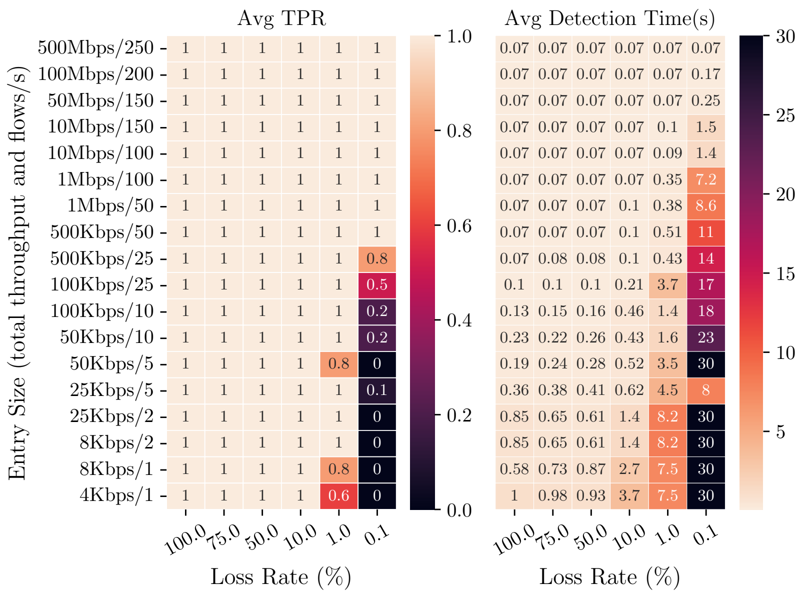


FIGURE 4.9: Accuracy and detection speed of dedicated counters for different gray failures and traffic volumes.

failures affecting all entries (Section 4.5.1.3), such as link-level problems or bugs exemplified in the second row of Table 4.1.

4.5.1.1 *Dedicated counters*

We assess the performance of dedicated counters by simulating single-entry failures only, because those counters work independently of each other.

We first evaluate the impact of the exchange frequency of counters. In principle, such a frequency may affect FANcY's accuracy because packet losses are not detected when counting sessions are opened and closed. Our simulations, however, indicate that FANcY's accuracy is not significantly impacted unless counters are exchanged extremely often. Accuracy results are indeed very similar whenever counters' exchange frequency ranges between 50 and 100 ms. This also means that the counters' exchange frequency just affects overhead and detection speed: increasing the exchange frequency speeds up failure detection but increases the overhead. Hereafter, we report results for a frequency value of 50 ms.

We now focus on FANcY's performance for different traffic volumes and loss rates. Results are depicted in Figure 4.9.

Accuracy. As displayed in the left part of Figure 4.9, FANcY's dedicated counters detect almost all gray failures whenever the induced packet drop rate is $\geq$1%, or the affected entries drive at least 500 Kbps of traffic.

Accuracy decreases for very low drop rates (e.g., 0.1%) of entries attracting little traffic ($\approx$100 Kbps or less). However, this is not an intrinsic limitation of FANcY, but mostly an artifact of our experiments. Indeed, very few packets are generated during these experiments, and chances are low that any packet is dropped if the loss rate is $\leq$0.1%. For example, in 80% of those experiments, no packet is actually dropped during the 30 seconds of the experiment. Only in the remaining 20% of the cases at least one packet is dropped. In those latter cases, FANcY fails to detect simulated failures because packets are dropped while FANcY closes a counting session or opens a new one. In real deployments, operators can reduce those cases by decreasing the counters' exchange frequency, which would trade detection speed for higher accuracy in very low drop-rate scenarios.

Detection speed. For dedicated counters, we expect a failure to be detected just after the first post-failure counters' exchange. The right part of Figure 4.9 shows that this is the case as long as the failed entries drive enough traffic (e.g., at least 500 Kbps). In fact, the top-left part of the right heatmap shows that the average detection time is $\approx$70 ms, which is approximately the counters' exchange frequency (50 ms) plus counting sessions' opening and closing.

Results may look less intuitive in the bottom part of the heatmap, where the average detection time increases to $\approx$600-1000 ms for blackholes, and to several seconds for lower packet-drop rates. Again, this does not directly depend on FANcY. Instead, packets affected by a failure tend to appear sometime after the failure is introduced for low-traffic entries and low drop rates. For example, if an entry drives one packet per second, on average the first packet for that entry is received by FANcY 500 ms after the failure is introduced.

4.5.1.2 *Hash-based tree*

Contrary to dedicated counters, the performance of hash-based trees generally depends on the number of entries failing simultaneously. In fact, the

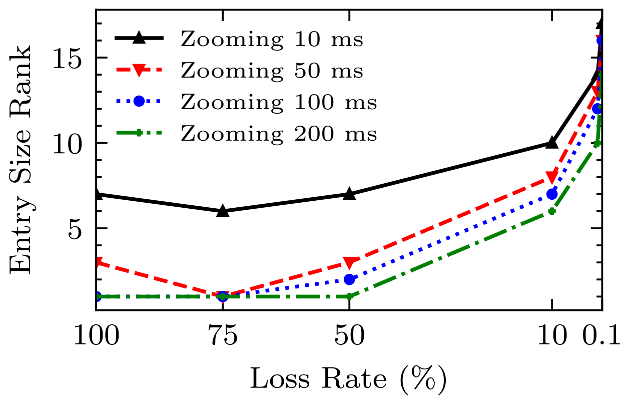


FIGURE 4.10: Minimum entry size for which FANcY has a TPR ≥95% for different zooming speeds. The y-axis ranks entries according to the traffic they drive: lower ranks correspond to smaller traffic.

detection of one failed entry may be delayed or even overlooked when FANcY zooms in the counters for another entry. We therefore evaluate both single-entry and multi-entry failure scenarios.

As the first step, we need to decide the duration of the counting sessions, which we denote as *zooming speed* for brevity. To do so, we measure the minimum prefix size required to get a TPR of at least 95% when we vary the loss rate and zooming speed. Results are plotted in Figure 4.10. All zooming speeds between 10 and 200 ms reach high TPR values, even for low loss rates (up to 0.1%), as long as the prefixes drive a reasonable amount of traffic. Additionally, requirements for traffic per entry are very similar across zooming speeds higher than 50 ms.

We conclude that FANcY's accuracy is not very sensitive to the tree's zooming speed between 50 ms and 200 ms. In the following, we show the results obtained using 200 ms as zooming speed, as it matches the typical value of TCP flows' retransmission timeout. We note that operators can fine-tune FANcY's zooming speed according to their specific requirements, as faster zooming speeds tend to decrease detection time but also increase overhead.

We now focus on FANcY's performance, comparing failures affecting only one entry (Figure 4.11) with those impacting 100 entries at the same time (Figure 4.12).

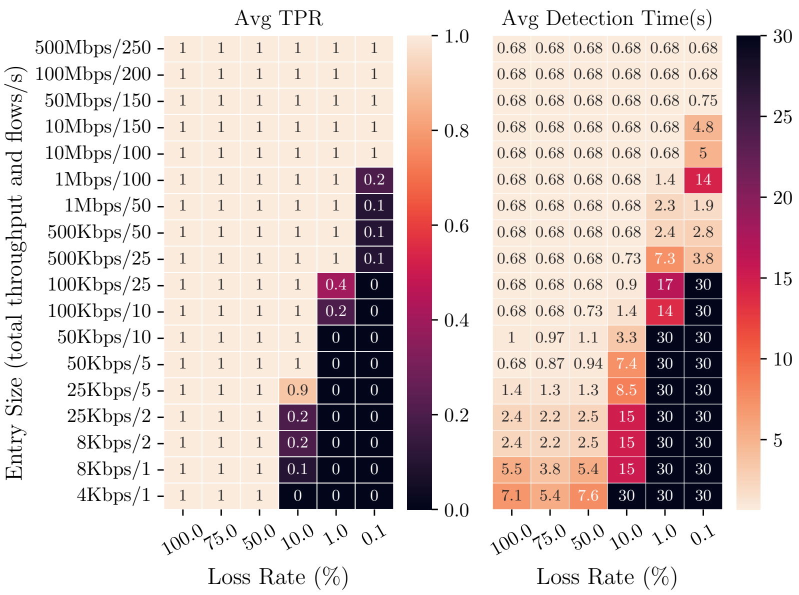


Figure 4.11: Accuracy and detection speed of FANcY's hash-based tree for single-entry failures and different traffic volume.

Accuracy. For single-entry failures, FANcY always identifies the failed entry as long as the packet loss rate is higher than 10%. For lower loss rates, FANcY's accuracy worsens for low-traffic entries. This is a direct consequence of our design: FANcY fully detects a failure after observing packet loss in three consecutive counting sessions, which becomes unlikely if it receives a few failure-affected packets. Indeed, in 97.5% of the experiments where FANcY fails to detect simulated failures, at no time are packets dropped during three consecutive counting sessions. We expect entries with those characteristics to collectively account for a limited percentage of real ISPs' traffic (see also Section 4.5.2), which makes this limitation not critical in real deployments.

For multi-entry failures, TPR values are consistent with those for single-entry failures, as evident when comparing the left part of Figure 4.12 with the left part of Figure 4.11. TPR decreases only for very low-traffic entries (e.g., 4-8 Kbps). For 80% of the runs in which FANcY fails to detect failures, no packets are dropped during three consecutive counting sessions – with

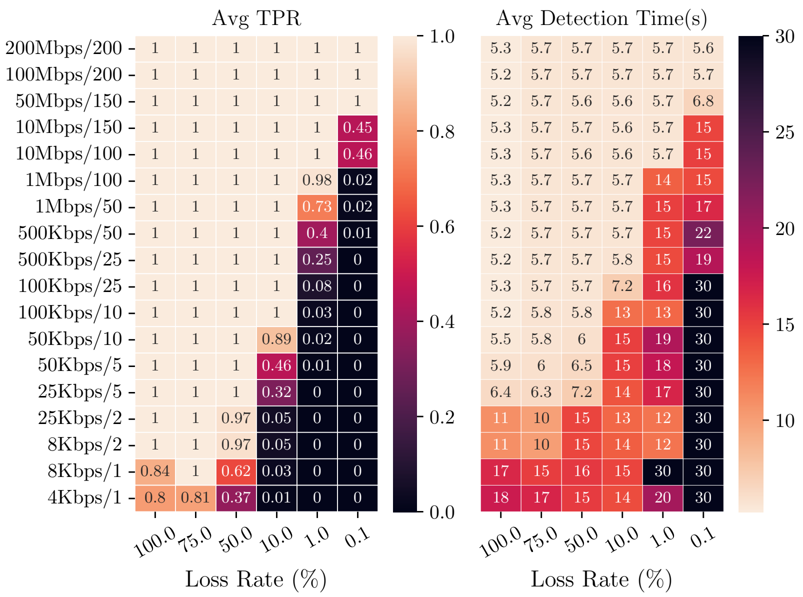


FIGURE 4.12: Accuracy and detection speed of FANcY's hash-based tree for 100-entry failures and different traffic volume.

packets lost while FANcY zooms in another entry in the remaining 20% of the experiments. Again, we expect entries attracting so little traffic to be not critical for ISPs. Those results thus suggest that FANcY's trees should be able to cover the practically relevant entries in ISPs' switches, even for failures simultaneously affecting a hundred entries.

Detection speed. FANcY is fast to detect failures of single entries with a reasonable amount of traffic and high loss rates: as shown by the right part of Figure 4.11, single-entry failures are typically detected in 680 ms, which roughly matches the lower bound of three times the selected zooming speed (i.e., 200 ms). FANcY detection slows down lower-traffic entries and low loss rates: for example, it changes from sub-second to a few seconds for single entries attracting $\leq$50 Kbps of traffic.

Increasing the number of failed entries has a more significant impact than the entry size. For 100-entry failures, the average detection time increases from 600 ms to about 5.3-5.7 seconds for high-loss high-traffic entries. This increase is motivated by the fact that FANcY zooms in a limited number of counters in each counting session – e.g., one root-level counter per session.

This choice enables FANcY to scale, but intrinsically degrades detection speed for many-entry failures, which we believe are relatively uncommon. We also stress that for all the scenarios where FANcY has high accuracy, the detection speed remains around 5-10 seconds, which is significantly much faster than the days or weeks currently needed by most operators.

4.5.1.3 *Uniform failures.*

We finally simulate failures affecting all entries simultaneously, such as random packet losses over a link, or bugs affecting all IP prefixes in Table 4.1. To be realistic, we simulate a network with 100 Gbps links, and assign traffic to entries mimicking a Zipf distribution. We experiment with packet loss rates per entry between 100% and 0.1%.

In all our experiments, FANcY detects the introduced failures and correctly identifies them as uniform random drops. Its average detection time matches one zooming interval (200 ms). This is consistent with the procedure used in FANcY to detect uniform failures, which is based on checking if the majority of root-level counters in the hash-based tree have mismatching values (as detailed in Section 4.3).

4.5.2 FANcY *on real traffic traces*

We now evaluate FANcY on CAIDA traces. The goal is to assess the traffic coverage provided by the whole system, combining the dedicated counters and hash-based tree, when traffic per entry follows a realistic distribution.

For that, we selected four different traces captured on different links and days. Further, we made sure that the selected traces exhibit different traffic conditions such as bit rate, packet rate, flow rate and even average packet size. Table 4.3 lists them and some of their characteristics.

ID	Link	Date	Bit Rate	Packet rate	Flow rate	Trace Size	Duration
1	caida-equinix-chicago.dirB	19-06-2014	6.25 Gbps	759.1 Kpps	28.3 Kfps	163 GB	3719 s
2	caida-equinix-nyc.dirA	19-04-2018	3.86 Gbps	557 Kpps	26.4 Kfps	125 GB	3719 s
3	caida-equinix-nyc.dirB	16-08-2018	5.79 Gbps	2.03 Mpps	104.5 Kfps	465 GB	3719 s
4	caida-equinix-nyc.dirB	17-01-2019	4.72 Gbps	1.56 Mpps	90.7 Kfps	345 GB	3720 s
Total						1.1 TB	4.1 h

Table 4.3: List of CAIDA traces [144] that we use to evaluate FANcY.

We stress that CAIDA traces constitute a challenging test for FANcY that we do not expect to be matched in real ISPs, for two reasons. First, the overall traffic rate (4-6 Gbps) in CAIDA traces is two orders of magnitude lower than typical rates in ISPs' links. Second, we assume that FANcY switches hold one forwarding entry for each /24 prefix observed in the trace (on average ≈250K), because IP addresses in the traces are anonymized at the /24 prefix granularity [145]. However, this assumption artificially inflates the number of entries with little traffic, which are exactly the ones more challenging for FANcY (see Section 4.5.1). As a reference, ≈ 60% (versus the 100% in our experiments) of the prefixes currently advertised on the Internet are /24s, according to public BGP data [146].

We rely on CAIDA traces because we are not aware of better publicly available ISP traffic traces. We however expect that in current ISPs and even more in future ones, FANcY's performance will be better than the already good results it achieves in the below experiments because FANcY's accuracy and speed generally improve with higher traffic per entry – see also Section 4.5.1.

Methodology. For each CAIDA trace, we assign a dedicated counter to each of the 500 prefixes with the most bytes during the entire trace (1h), mimicking an allocation based on historical data. Then, we randomly select a 30-second slice from each trace. Note that the prefixes carrying more traffic during each slice do not generally coincide with those covered by dedicated counters.

We then implement a traffic generator that closely reproduces any input slice. In the absence of failures, the generator re-injects all the packets of each flow exactly when they appear in the slice, preserving the bit rate, packet rate, and RTT of flows. The generator relies on ns-3's TCP implementation, enabling us to run closed-loop experiments, with TCP sources reacting to packet losses.

We use each slice to perform experiments simulating the failure of the top 10,000 prefixes (which carry $\geq$ 95% amount of the total traffic in the entire trace), one by one, at a random time. For each prefix and loss rate, we repeat the experiment 3 times, with the time of the failure changing in each repetition. As for the simulations in Section 4.5.1, there is no guarantee that packets for the failed entries are actually dropped within the duration of the experiment, especially for low drop rates.

Loss	TPR Bytes			TPR Prefixes					Detection Time		
Rate	Total	Dedicated	Hash-Tree	Total	Dedicated	Hash-Tree	Top 1K	Top 5K	Total	Top 1K	Top 5K
100%	91.3%	57.1%	34.2%	84.5%	100%	83.6%	89.2%	87.8%	2.03 *s*	0.67 *s*	1.38 *s*
75%	96.0%	57.1%	38.9%	90.9%	100%	90.3%	95%	94%	2.59 *s*	0.86 *s*	1.78 *s*
50%	98.7%	57.1%	41.5%	93.1%	100%	92.6%	98.3%	97.3%	2.65 *s*	0.81 *s*	1.86 *s*
10%	96.5%	57.1%	39.4%	72.8%	100%	71%	97.6%	85.9%	4.96 *s*	0.73 *s*	3.48 *s*
1%	77.5%	57.1%	20.4%	19.5%	98.9%	14.7%	85.4%	33.4%	8.91 *s*	4.19 *s*	8.49 *s*
0.1%	56.6%	55.9%	5.30%	5%	86.7%	0.1%	39.6%	0.084%	6.29 *s*	6.13 *s*	6.29 *s*

Table 4.4: Average accuracy and detection speed of FANcY over four CAIDA traces (See Table 4.3 for more information on the used traces).

FANcY's performance. As shown in Table 4.4, FANcY detects between 91.3% and 98.7% of affected bytes in 2-5 seconds when the loss rate is ≥10%. For the same failure scenarios, the TPR in terms of detected entries is 72.8%-93.1%, a bit lower than the TPR in terms of bytes: this happens because traffic per prefix is very skewed in CAIDA traces.

For loss rates ≤1%, FANcY's accuracy is significantly impacted (5%-19.5%), mainly because the hash-based tree's TPR decreases sharply, in line with the results presented in Section 4.5.1. The main reason for those low TPR rates is the lack of packet drops during three consecutive counting sessions, which directly prevents FANcY's failure detection in ≈80% (resp., >99.8%) of the experiments with a loss rate of 1% (resp., 0.1%). Those results further stress the importance of the hash-based tree in our design: FANcY covers only 56.6% of the bytes affected by failures when the tree's TPR is close to zero versus ≈ 99% when the tree's TPR is high.

It may seem surprising that FANcY does not perform at best when traffic is blackholed (100% loss rate). This is because FANcY measures packet loss on the observed traffic, and a hard failure immediately slows down *all* the TCP flows, reducing all affected traffic to just retransmissions. Namely, for each flow, FANcY receives the first retransmission after the expiration of the TCP retransmission timeout (typically 200 ms), and further retransmissions at exponentially increasing times. In other words, TCP congestion control makes it more likely for FANcY not to receive packets for the failed entries in three consecutive counting sessions, thus reducing the tree's TPR. In contrast, FANcY performs very well when the loss rate is around 50%, where TCP reduces the flow rate much less significantly and less abruptly.

Comparison to baselines. We compare FANcY's results with the simpler designs outlined in Section 4.1.4: a single counter per link, and one dedicated counter for each prefix.

Both designs achieve a slightly higher accuracy than FANcY: their TPR for prefixes is ≈97-99.6% for a loss rate ≥10%, ≈84% for a loss rate of 1%, and ≈35% for a loss rate of 0.1%. Their accuracy is not 100% because switches may not receive traffic for the failed entries before our experiments terminate, and may not detect packet losses when exchanging counters.

However, a single counter cannot localize any failure; the number of false positives in each experiment is the total number of prefixes minus the failed ones – i.e., ≈250K. In contrast, the solution with one dedicated counter per entry has zero false positives, but it requires 320 MB (including support for the counting protocol) versus the 1.25 MB consumed by FANcY in total. Note that the memory required by one dedicated counter per entry is expected to be ≈4 times in real ISPs holding the full BGP table (i.e., ≈900K prefixes).

We then consider two additional alternatives compatible with FANcY's memory usage. The first alternative is to allocate only dedicated counters but without exceeding FANcY's memory budget. With 1.25 MB, we can allocate a maximum of 1,024 dedicated entries per port. This approach is accurate and fast for the covered prefixes, but detects no failure for any of the remaining ≈249K ones, which carry ≈40% of the traffic in the considered CAIDA traces. As the second alternative, we consider allocating all the memory to a counting Bloom filter. The TPR of such a Bloom filter is largely consistent with the single-counter approach. However, for each detected single-entry failure, the Bloom filter reports ≈100 false positives versus the ≈0.03 of FANcY. Once again, we expect that the number of false positives for the Bloom filter solution to be much higher in real ISPs, where switches typically hold significantly bigger routing tables.

Takeaways. Our results confirm FANcY's ability to detect different types of gray failures, covering the vast majority of the real-world traffic, while also achieving a much better trade-off between accuracy, speed, and scalability than simple designs.

We expect FANcY to perform significantly better when deployed in actual ISPs. Indeed, CAIDA traces contain unrealistically low traffic per entry with respect to current and future ISP settings – a condition unfavorable to FANcY as already demonstrated in Section 4.5.1.

Results in this section are consistent with those for synthetic, non-bursty traffic, described in Section 4.5.1. They also provide consistent indications on the limits of FANcY: tiny failures of entries driving little traffic tend to be very hard to detect with hash-based trees. If operators want to protect specific entries from low loss rates, one option within FANcY's design is to specify them as high-priority entries in the FANcY's input.

4.5.3 *Overhead analysis*

We now show that FANcY's overhead is minimal on ISP-scale links. In FANcY, we have two overhead components: control packets (including counters) and packet tags added by FANcY switches.

We first consider the overhead of control packets. For dedicated counters, FANcY sends five minimum-size packets (e.g., 64 B Ethernet frames) for each link and each counting session. With 500 dedicated counters exchanged every 50 ms on a 10 ms delay link, FANcY uses ≈0.014% of a 100 Gbps link's capacity. For hash-based trees, FANcY also exchanges five control packets, including the hash-tree counter that carries 5320 B in the pipelined version of the zooming algorithm. The resulting traffic overhead is ≈0.00017% on 100 Gbps links for a zooming speed of 200 ms.

To tag packets, FANcY needs 2 bytes to specify the counter ID on each packet matched by a dedicated counter. The same amount of bytes are added to packets counted in the hash-based tree, where one byte encodes the hash path of the tree's node, and the other identifies the counter within the node. The tagging overhead is therefore 0.13% on a 1,500 B packet. Note that tags can also be encoded in unused header fields, which would lead to zero overhead.

4.6 TOFINO MADE FANCY

In this section, we first introduce FANcY's hardware implementation. Our implementation of FANcY is composed of ≈3200 lines of P4 code running on a first-generation Intel Tofino switch [18] with 32 ports. Second, we give an overview of the hardware resource utilization and detail the amount of memory consumed by each key component of our implementation. Finally, throughout a case study we show how FANcY can detect failures and react to them only using data plane logic.

4.6.1 *Hardware implementation*

We first describe our implementation of the state machines, and then we focus on how we support hash-based trees.

State machines. While implementing each state is relatively simple (i.e., storing a state ID and possible counters in registers), supporting state transitions is not. In programmable switches such as Intel Tofino, state is maintained using registers. In Intel Tofino switches, register memory is local to a pipeline stage. Therefore, it can only be accessed once as it crosses the pipeline [147]. Furthermore, read and write operations on register memory are performed in one operation, with a very limited update logic (i.e., you cannot write complex update operations that depend on the read value). This limitation makes the implementation of state machines quite challenging as it requires us to read state, do relatively complex operations and then update the state in a single step. To address this limitation, we implemented a novel two-step approach.

The first step only triggers the state transition and is based on a match-action table, called *next_state* table. This table defines all the possible state transitions. When a FANcY switch receives a packet, it reads the current state from a register and matches the packet against the *next_state* table, if a transition needs to be made, The switch logic will (i) write in the *state_lock* register, in order to avoid additional transitions while the state is being updated, (ii) store all the information needed to update the state in the current packet's metadata, and (iii) force the packet to cross the pipeline again to perform the state update.[3]

The second step actually performs the transition. The recirculated packet updates the stored state ID, resets the state counters (e.g., timers), and releases the *state_lock*. Based on prior and next states information in its metadata, the packet also triggers a transition-specific action: either drops the packet, performs a computation, or transforms it into a control message (*ACK*, *STOP*, etc.) to send out. A final note concerns time-based transitions (e.g., timers). Since time-based events are not supported by current switches,

3 For technical reason, we *resubmit* packets in ingress FSMs and *clone* packets in egress FSMs (making sure we remove unneeded clones)

we approximate them using traffic and packet counts. In the absence of traffic, the internal traffic generator can be used.

Hash-based tree and zooming algorithm. We implement the hash-based tree (depth 3 and split 1) by using four register arrays. One register array, which we call *node register*, stores actual nodes of all the trees kept by the switch (i.e., one per port). The other three register arrays store metadata to support the zooming algorithm: for each tree, the *zooming stage* register array keeps information on the depth we are currently zooming in, the *max0* register indicates the counter at layer zero we are zooming in, and the *max1* register stores the same information but for the counter at layer one. The procedure to update counters in any tree T of width w is implemented as follows. Each incoming packet is hashed according to one hash function per tree's level. We then decide if the *node register* has to be updated by checking whether the *zooming stage* register for T is 0 (i.e., we always update counters when not zooming), or comparing the result of the packet's H_0 with *max0* (if the *zooming stage* register for T is 1) and packet's H_0, H_1 with *max0* and *max1* (if the *zooming stage* register for T is 2). If the *node register* has to be updated, we increase the counter at the address $(H_i \bmod w + o)$, where i is the value stored in the *zooming stage* register for T and o is the port offset that identifies T within the *node register*.

In addition to increasing packet counters, we also support two other operations. First, the downstream switch sends all T's counters in the *node register* to the upstream at the end of each counting session. Since register arrays can be accessed only once per packet, we recirculate packets w times to read all such counters from the *node register*.

Second, the upstream switch compares local counters in T with those reported from downstream switches. Again, since only one register can be read for each packet, we recirculate packets w times to compare the counters one by one. If the *zooming stage* register for T is 2, we simply report (to our reroute app or externally) all the counters with mismatching values. Otherwise, if the *zooming stage* register for T is 0 or 1, we need to compute the counter in T's *node register* with the biggest difference of values between upstream and downstream. We do so by storing the current maximum difference and counter index in a custom header of the packet that we recirculate. After all the counters in T's *node register* are compared, we finally copy the counter index in the recirculated packet's metadata to either *max0* or *max1* (depending on the current value in the *zooming stage* register) and increase the *zooming stage* register by one modulo three.

Resource	Dedicated Counters	Full FANcY	FANcY + Rerouting	switch.p4
SRAM	4.80%	6.65%	8.1%	29.58%
Statefu ALU	16.66%	27.08%	33.33%	14.58%
VLIW Actions	9.4%	14.1%	15.6%	36.72%
TCAM	1.4%	2.1%	2.1%	32.29%
Hash bits	5.8%	11.8%	13.1%	34.74%
Ternary Xbar	1.8%	3.10%	3.10%	43.18%
Exact Xbar	5.1%	10.8%	12.3%	29.36%

TABLE 4.5: Hardware resource usage of FANcY compared to the baseline switch.p4 on a 32-port Intel Tofino switch.

4.6.2 *Hardware resources and memory usage*

In the following section, we give an overview of the hardware resources used by FANcY on an Intel Tofino switch. Further, we detail how much SRAM is used by each component of our design.

Table 4.5 summarizes the resource usage of FANcY, using switch.p4 as a baseline. Overall, FANcY uses a modest amount of hardware resources, including only 6.65% of SRAM (8.1% with rerouting). Stateful ALUs are the only resource that FANcY uses more than switch.p4: this is because FANcY performs several stateful operations to support counters and the counter exchange protocol. For more details about SALUs usage read below. Note that SRAM is the only resource that increases when FANcY is given a higher memory budget and then uses more dedicated counters or larger trees.

FANcY scales and fits very well in current hardware switches. We now detail the resources needed for each component in a 32-port Tofino switch. Note that the software and hardware implementations use the same data structures, however the hardware implementation runs non-pipelined hash-based trees, which heavily reduces the memory consumption. For more details on memory utilization for any type of hash-based tree, see Sec 4.3.3.3.

State machines. Each state machine uses three registers (at ingress and egress): State counter (or timer), current state, and state lock, 32, 8, and 8 bits, respectively. We need one array cell in each of those registers for each sub-state machine used by either dedicated counters or a hash-tree. For each state machine pair, FANcY needs $(32 + 8 + 8) \cdot 2 = 96$ bits. If we want to have 512 state machines per port in a 32-port switch, we need $96 \cdot 512 \cdot 32 = 192$ KB.

Dedicated counters. Each entry covered by dedicated counters requires one pair of 32-bit registers to count packets in each direction, $32 \cdot 2 = 64$ bits per entry per switch. Our implementation of FANcY includes 512 dedicated counters per port. The memory consumption of those counters in a 32-port switch is therefore $64 \cdot 512 \cdot 32 = 128$ KB.

Hash-based tree and zooming algorithm. Supporting any hash-based tree requires five registers in total. First, we need the two 32-bit registers where we will store tree's nodes. Then, at the egress pipe, we have three registers used by the zooming algorithm; zooming stage, max0 and max1, 8, 16, and 16 bits, respectively. Since we implement a hash-tree zooming algorithm without split and pipelining, we can reuse the same memory cells for each tree layer, considerably reducing the memory needed. The hash-based trees in our implementation have width $w = 190$. Each of them therefore needs $32 \cdot 2 \cdot 190 = 12160$ bits per port for the counters, and $8 + 16 + 16 = 40$ bits to keep zooming state. In total, for a 32-port switch we need $(12160 + 40) \cdot 32 = 47.6$ KB.

Rerouting. Supporting the rerouting logic also needs some switch memory. We use 3 registers (all at the ingress) for that: one for dedicated counter entries and one for failures detected with the hash-based tree. For dedicated counter entries, we use a 1-bit wide array, thus we need 1 bit per entry and port. For 512 entries and 32 ports, we need 2 KB. For failures detected via the hash-based tree, we additionally need to use a Bloom filter implemented as two 1-bit registers of 100K cells. The memory used for the rerouting is 26.4 KB.

Total memory. For a 32 port switch, with 512 dedicated counter entries, one hash-based tree of width 190 per port and depth 3 is 367.6 KB (394 KB with rerouting).

4.6.3 *Case study: fine-grained fast rerouting*

As a case study, we build an application on top of FANcY that reroutes packets as soon as the corresponding counters are flagged as mismatching

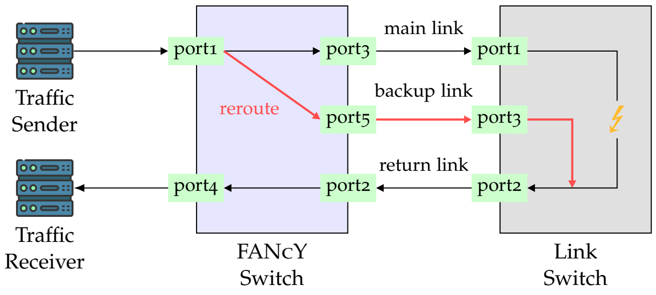


Figure 4.13: Case study topology with two Tofino switches, one running FANcY sender and receiver pipelines, and the other acting as a middle switch to simulate gray failures.

by FANcY (see Section 4.3.4). Note that simply rerouting might not be enough to fix some of the problems shown in Table 4.1. However, it might be enough for inter-switch gray failures caused by faulty links. We now detail our experiments with this application.

Setup. Figure 4.13 illustrates the case study setup. We use two servers, a sender and a receiver, and two Wedge 100BF-32X [18] Intel Tofino switches. The servers are equipped with Intel Xeon E5-2670 v3 2.30GHz CPUs, 256 GB of RAM, and a Mellanox ConnectX-5 100 Gbps NIC.

We connect each server to the Tofino switch that runs FANcY. The sender server generates TCP flows for a total of 50 Gbps of traffic, and 50 Mbps of UDP traffic. For each port, the FANcY switch maintains 500 dedicated counters, and implements a hash-based tree of depth 3, split 1, and width 190. Dedicated counters are exchanged every 200 ms, and the zooming speed for the tree is set to ≈200 ms. We run separate experiments for prefixes mapped to dedicated counters, and for those covered by the hash-based tree.

We use the second Tofino switch as a *link switch* connecting two ports of the FANcY switch. After 2 seconds from the start of each experiment, we instruct the link switch to drop 1%, 10%, or 100% of the packets (in different experiments). We also deploy a third link between the FANcY switch and the link switch, to provide the former with a backup next-hop.

Experimental results. Figure 4.14 shows the traffic throughput as measured in our experiments. In each experiment, the FANcY switch always detects the failure event less than one second after it is introduced, even when the

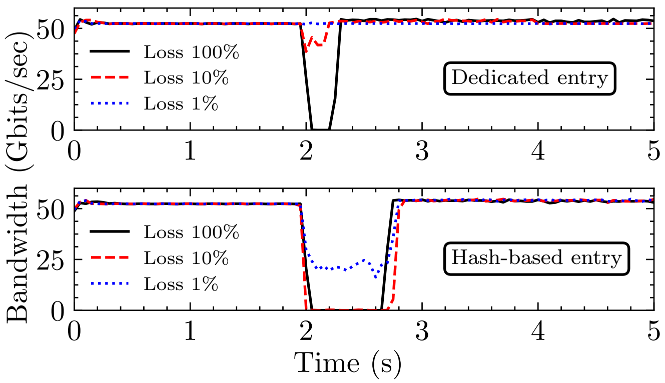


FIGURE 4.14: Case study using our FANcY implementation on a Tofino switch: FANcY detects gray failures even when affecting only 1% of the packets per entry, and reroutes the traffic only for the affected entries in less than one second.

drop rate is only 1%, and the affected traffic is monitored by the hash-based tree. As expected, the detection time is proportional to three times the zooming speed (here, 3 × 200 ms) when failures affect entries covered by the hash-based tree. On the other hand, dedicated counters ensure a predictable detection time, which depends only on the counting session duration (250 ms). Note that we have used a relatively higher counting session duration than the one used during the evaluation (50 ms) so that the impact of the failure is noticeable in the plot.

4.7 CONCLUSION

In this chapter, we introduced FANcY, a data-plane system designed to detect intra-domain gray failures within Internet Service Providers (ISPs). FANcY enables programmable switches to synchronize counters in a way that is both reliable and scalable, all without the need for direct control plane intervention. FANcY's hybrid counter-based approach complements pre-existing failure detection systems, which are effective in data center networks, but present limitations with the high-traffic volumes and link delays typical of ISPs.

Although FANcY focuses on detecting and reporting (but not directly fixing) failures, its interface enables future applications such as selective fast rerouting or root cause analyses. As a feasibility proof, we implemented and made an open prototype of FANcY in a commercial Tofino 1 switch, and demonstrated how our implementation enables sub-second fast rerouting around gray failures.

Our evaluation shows that FANcY can detect and localize gray failures quickly and accurately in ISP settings, except those that induce few, sporadic packet losses per entry – as expected, since FANcY is a data-driven system.

It's important to note that FANcY's performance actually *improves* with increasing traffic volumes, thereby affirming its future-proof design.

5 HARDWARE-ACCELERATED NETWORK CONTROL PLANES

In the previous chapters of this dissertation, we conducted a comprehensive study of network failures, focusing on gray failures within ISP networks. Our research highlighted gray failures as a significant issue affecting Internet users, for which network operators lack efficient solutions. We introduced a novel technique utilizing advances in programmable data planes, demonstrating how in-network programmability can facilitate effective and efficient detection algorithms that operate on all traffic at line rate.

In this chapter, we go one step further and explore the potential and benefits of accelerating the network control plane by offloading some of its tasks directly to network hardware. With FANcY, we have demonstrated how programmable data planes are an excellent technology that can help improve network monitoring. In this chapter, we show that programmable data planes are also powerful enough to run key control plane tasks, including notification, connectivity retrieval, and even policy-based routing protocols. We implement a prototype of such "hardware-accelerated" control plane functions in P4 and illustrate its benefits through a case study.

Despite such benefits, we acknowledge that offloading tasks to hardware is not a silver bullet. We discuss its trade-offs and limitations, and outline future research directions towards hardware-software codesign of network control planes.

As the "brain" of the network, the control plane is one of its most important components. Among other things, a traditional control plane is responsible for *sensing* the status of the network (e.g., which links are up or which links are overloaded), *computing* the best paths along which to guide traffic, and *updating* the underlying data plane accordingly. To do so, the control plane comprises many dynamic and interacting processes (e.g., routing, management and accounting protocols) whose operation must scale to large networks. In contrast, the traditional data plane is "only" responsible for forwarding traffic according to the control plane decisions, albeit as fast as possible.

These fundamental differences lead to very different design philosophies. Given the relative simplicity of the data plane and the "need for speed," it is typically entirely implemented in hardware. That said, software-based implementations of data planes are also commonly found (e.g., OpenVSwitch [51]) together with hybrid software-hardware ones (e.g., CacheFlow [148]). In short, data plane implementations cover the entire implementation spectrum, from pure software to pure hardware. In contrast, there is *much* less diversity in control plane implementations. The sheer complexity of the control plane tasks (e.g., performing routing computations) together with the need to update them relatively frequently (e.g., to support new protocols and features) indeed calls for software-based implementations, with only a few key tasks (e.g., detecting physical failures, activating backup forwarding state) being (sometimes) offloaded to hardware [149, 150].

We argue, however, that a number of recent developments are creating both the *need* and *opportunity* for rethinking basic design and implementation choices of network control planes.

Need. There is a growing need for faster, more scalable, and yet more powerful control planes. Nowadays, even beefed-up and highly optimized software control planes can only process thousands of (BGP) control plane messages per second [151] and can take *minutes* to converge upon large failures [152, 153]. Parallelizing only marginally helps: for instance, the BGP specification [154] mandates to lock all Adj-RIBs-In before proceeding with the best-path calculation, essentially preventing the parallel execution of best path computations. A concrete risk is that convergence time will keep increasing with the network size and the number of Internet destinations. At the same time, recent research has repeatedly shown the performance benefits of controlling networks with extremely tight control loops, among others to handle congestion (e.g., [38, 39, 155]).

Opportunity. Modern programmable switches (e.g., [156]) can perform complex stateful computations on billions of packets per second [19]. Running (pieces of) the control plane at such speeds would lead to almost "instantaneous" convergence, leaving the propagation time of the messages as the primary bottleneck. Besides speed, offloading control plane tasks to hardware would also help by making them traffic-aware. For instance, it enables updating forwarding entries consistently with real-time traffic volumes rather than in a random order. Furthermore, as shown in the previous chapter, tasks like monitoring, traditionally implemented in the control plane, considerably benefit from running directly within the data plane.

Research questions. Given the opportunity and the need, we argue that it is time to revisit the control plane's design and implementation by considering the problem of offloading parts of it to hardware. This redesign opens the door to multiple research questions including: *Which pieces of the control plane should be offloaded? What are the benefits?* and *How can we overcome the fundamental hardware limitations?* These fundamental limitations come mainly from the very limited instruction set (e.g., no floating point) and the memory available (i.e., around tens of megabytes [19]) of programmable network hardware. We start to answer these questions in this paper and make two contributions.

First, we illustrate that programmable switches are powerful enough to run control plane tasks beyond failure detection directly in hardware. Specifically, we implement a working prototype of a hardware-accelerated control plane in P4 [157]. Our approach enables P4-enabled switches' hardware to perform the following tasks, *autonomously* and *at line rate*: *(i)* detect hard, gray and remote failures; *(ii)* run distributed path-vector computations that support both shortest-path and BGP-like policies; and *(iii)* directly update the forwarding state.

Our implementation compensates for the computation and memory limitations with additional packet exchanges. For example, during path computations, each switch only stores the best path, forgetting its alternatives. This implies that more packets have to be exchanged upon configuration or topological changes. Yet, this only induces a marginal cost for hardware implementations, as packet processing takes nanoseconds [158].

Second, we discuss the pros and cons of offloading control plane tasks to hardware. Based on this analysis, we sketch a research agenda centered around the investigation of a software-hardware codesign approach to network control planes, aimed at systematically exploring the trade-offs of running tasks in software, hardware, or a combination of the two.

Our observations complement recent proposals on hardware offloading for network monitoring tasks [159–161], congestion control [162], coordination services [163], consensus algorithms [164, 165], and application-level caching [19, 158]. A few proposals, like DDC [166], have also shown how to offload specific functions to the data plane, such as maintaining connectivity. We expand on this intuition, considering any control plane task as a candidate for hardware offloading.

Overall, we think that offloading control plane tasks to hardware has the potential to radically change the way networks are designed in the future.

5.1 HARDWARE-BASED CONTROL PLANE

Networks are organized around two planes: the control plane and the data plane. The Control Plane (CP) is the "brain" of the network and is responsible for computing forwarding paths. It can be either logically-centralized, as in SDN networks, or distributed, as in "traditional networks" running distributed protocols (IGP, BGP, etc.). The role of the Data Plane (DP) is simply to forward traffic (as fast as possible) according to the CP decisions. While the DP can be implemented in either hardware or software, the CP is typically implemented in software and involves three main processes:

1. *Sensing:* The CP monitors the network topology and configuration, in order to detect changes (e.g., link failures) that may require adapting the forwarding state.
2. *Notification:* When detecting a change, the CP notifies the path computation component. If the CP is logically centralized [49], the central controller is notified. If the CP is distributed, all the network nodes must be notified about the change.
3. *Computation:* When becoming aware of a topological change, the routing component of the CP recomputes the forwarding paths. Once new paths are computed, the CP updates the data-plane.

In this section, we show that each step can run directly in hardware, paving the way for hardware-based CPs.

We use the 4-switches network of Figure 5.1 as visual support. The figure illustrates how a hardware-based CP senses and retrieves connectivity upon a partial link failure occurring between switches B and C.

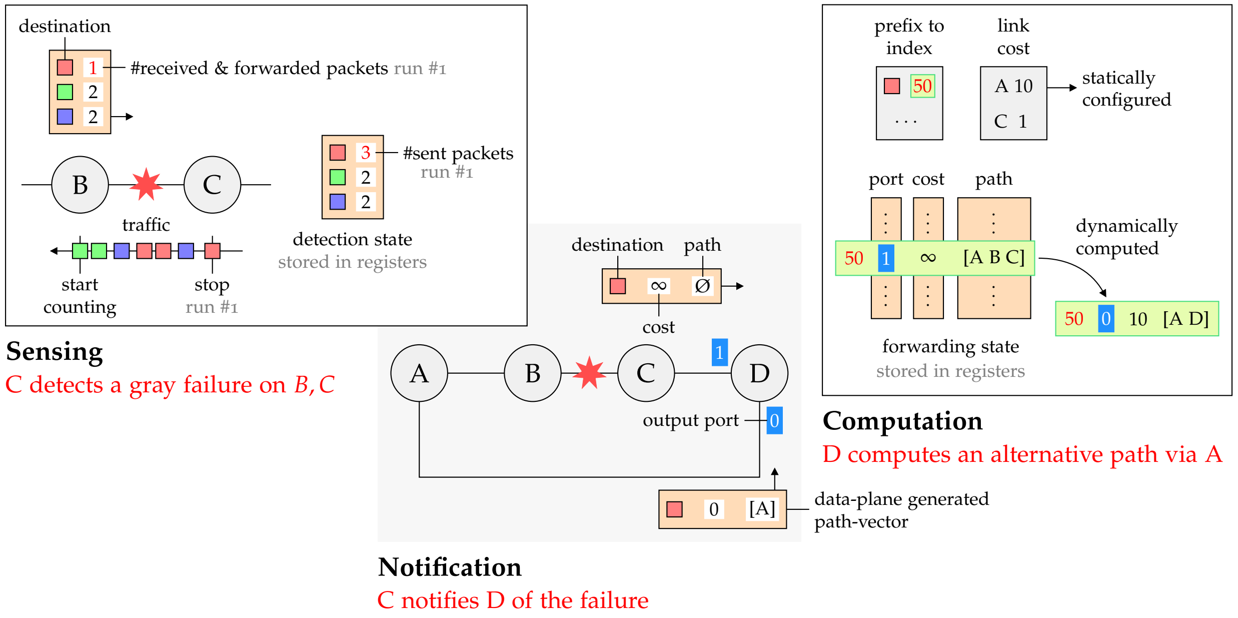


FIGURE 5.1: Despite being limited in terms of computation logic and memory, programmable data planes are powerful enough to run key control plane tasks enabling them to compute forwarding state entirely on their own.

5.1.1 *Hardware-based sensing*

As shown in Section 3.2.1, some forms of hardware-based sensing are already available today. Existing approaches rely on either monitoring properties of the physical medium (e.g., loss of light in an optical fiber) or running the Bidirectional Forwarding Detection (BFD) protocol [149]. BFD sends small echo packets every x ms (50 ms by default [167]) and generates an alert if more than k packets have not been received.

Challenges. Existing hardware sensing schemes can only detect local hard failures, such as a link or a node failing within a network domain. However, these sensing systems fail at detecting gray and remote failures. Gray failures, which are partial and affect only a subset of the traffic (e.g., packets matching a specific forwarding entry [11]), present a unique challenge. Detecting gray failures requires them to be exerted by actual traffic, preventing simple hardware hello-based mechanisms from working.

Remote failures, which occur beyond our immediate network domain, pose a substantial challenge. The slow convergence time associated with these failures, often exceeding 30 seconds [153], is a consequence of the control-plane-driven process, which requires the propagation of BGP updates on a per-router and per-prefix basis. Although previous work, such as Swift [153] has sought to reduce this convergence time by predicting the extent of a remote failure from only a few BGP updates, the propagation of the first BGP update post-failure can still take up to minutes [3]. This control-plane-induced delay highlights the critical need for an innovative solution capable of detecting such failures directly from data-plane signals.

Our approach. A simple solution would be to generalize the concept of BFD to detect both hard and gray failures, in hardware. That could be achieved by programming adjacent switches to "acknowledge" the *data-plane traffic* that they exchange, rather than BFD hello packets. While this may seem excessive, acknowledgments only need to contain enough header information to identify the rule being touched by the packet (e.g., the 32-bit destination prefix). Assuming 32-bit acknowledgments,[1] the overhead would be 267 Mbps for 100 Gbps of traffic with 1500 bytes packet. With minimal-size packets (64 bytes), the same volume of traffic would require 6.25 Gbps of acknowledgments.

1 Here, we consider the use of protocol-independent switches which do not mandate the use of an Ethernet header.

To avoid acknowledging every single packet, we propose a scheme in which switches synchronously exchange packet counts processed by any given forwarding rule. Specifically, an upstream switch instructs a downstream switch to start and stop counting packets matching a given forwarding rule. When receiving the stop signal, the downstream switch sends the counter back to its upstream which compares it with its own packet count. This process is illustrated in Figure 5.1 (left), where C, the upstream switch, sends packets to B. The different counter values for the red destination indicate a gray failure which is reported by C network-wide.

Although effective, this relatively simple packet-counting approach can only be used when the number of forwarding entries to monitor is relatively small, due to hardware resource constraints. To address this scalability issue for local failure detection, we utilize FANcY, as introduced in Chapter 4.3. FANcY offers a scalable hybrid solution by incorporating dedicated counters for high-priority monitoring needs while aggregating other entries in a hash-based tree data structure. This approach allows for efficient monitoring without the limitations imposed by current hardware, adapting dynamically to the network-wide range of monitoring needs.

We integrate Blink [3][2] into our data-plane sensing design to detect remote failures. Blink is a data-driven system that leverages data-plane signals, specifically TCP flows, for remote failure detection, as opposed to slower control-plane ones. Blink is based on the observation that, following a failure, TCP clients retransmit packets with the same SEQ number repeatedly following an exponential backoff. When aggregated across multiple flows, such a pattern provides a strong signal indicative of a failure. By deploying Blink at the network's edge, our hardware-based sensing framework can detect and recover from remote failures before receiving the regular control-plane signals (i.e., BGP withdrawals), thereby reducing the recovery time from potentially minutes to sub-second intervals.

5.1.2 *Hardware-based notification*

We take inspiration from the simplest, least memory-consuming routing protocols, and implement a broadcasting notification mechanism in hardware. As shown in Figure 5.1, notifications correspond to the generation of path-vector messages. Those messages are also used during the path

2 The Blink project is part of this thesis. Although the author made a substantial contribution and is credited as a co-author, he was not the lead researcher

computation, and carry information on: *(i)* affected destinations; *(ii)* the most updated path (i.e., empty for the link failure in the figure); and *(iii)* its cost (i.e., infinity for failures).

Challenges. Broadcasting in hardware poses two main challenges. First, notifications must be exchanged reliably to guarantee correctness. Implementing reliable communication in hardware is challenging as it requires maintaining state, tracking timers, and dealing with the inevitable retransmissions. Second, broadcasting notifications requires extra care to avoid broadcast storms in the presence of physical cycles.

Our approach. We deal with packet loss in two ways. First, we classify control packets in high-priority queues, reducing the likelihood of packet loss. Second, we leverage that the cost of processing a packet is almost negligible in hardware, and either duplicate messages k times, for notifications, or repeat them regularly, e.g., every few ms, for regular state exchange. While continuously repeating state exchange guarantees its eventual consistency network-wide, there is still a small probability that some switches will not receive any of the k retransmitted notifications, leading to a partially converged network. In future work, we intend to develop a lightweight form of reliable message exchange.

To avoid broadcast storms, the originator switch attaches its identifier and a sequence number to the broadcasted packet. Each switch maintains a register with the last sequence number observed for every other switch. Whenever a switch receives a broadcast message, it checks whether the sequence number is smaller than the one stored for the message originator, and drops the packet if it is the case. The sequence number is increased by one during the next broadcasting.

5.1.3 *Hardware-based computation*

We implement a distributed path-vector routing algorithm in hardware in which switches exchange vectors and locally select the vector with the best attributes, e.g., the one with the lowest cost or the one with the highest preference. By doing so, our hardware computation supports policy-based (i.e., BGP-like) routing logic.

Challenges. A key challenge is that the computation logic available is limited and geared towards forwarding pipelines, not distributed algorithms. For instance, P4 match-action primitives do not support basic constructs like loops. On top of that, resources are heavily limited (in terms of size and data structure types), which clashes with the typical choice of routing protocols to maintain *a lot* of state, such as all the routes received or the entire network map. Finally, supporting routing policies adds an extra level of complexity as the presence of policies render many routing problems computationally-hard [168].

Our approach. To manage complexity and reduce the amount of state maintained by each switch, we only make them store the best path and its attributes. This simplifies the computation as it removes the need to iterate: a switch only needs to compare the received attributes with the currently best known path, and possibly adapt the latter accordingly. Of course, it also reduces the amount of state maintained by each switch to the bare minimum.

Observe that this strategy is sufficient to compute a new best path if some input changes, provided each switch re-advertises its best-known path upon a change. To ensure this, the failure notifications are flooded and necessarily trigger a re-advertisement. While doing so leads to more messages than software CPs storing alternative paths, we stress that this is not a problem since hardware-based computation can process *billions* of such packets per second [19].

Finally, we leverage the seminal results from Sobrinho [169] to compute the outcome of policy-based protocols such as BGP in hardware. Those results show that *generic* path-vector protocols can emulate the semantics of policy-based protocols, if the right set of costs is chosen. This observation enables to move the complexity of dealing with policies from the protocol to the path costs. We show how our hardware-based CP encodes typical BGP policies (prefer customer over peer over provider routes) in Section 5.2.

5.2 PRELIMINARY IMPLEMENTATION

We now describe a preliminary P4 implementation of our hardware-accelerated control plane and illustrate its usefulness through a case study in which switches converge entirely independently, for both intra- and inter-domain destinations.

5.2.1 *Implementation*

We implement our algorithms in $P4_{16}$ [60] and use the bmv2 [72] behavioral model to test them. We also implement a software-based control plane logic which is in charge of populating the switches' initial state. The remaining part of the logic is implemented solely within the switches following the approaches described in Section 5.1. Overall, our implementation consists of 1800 lines of P4 and 3000 lines of Python code.

We performed our experiments on a server equipped with a 2×12 Xeon E5-2670 2.30GHz, 128GB RAM and running Ubuntu 16.04. In the future, we intend to adapt and run our algorithms on Tofino switches [156] with a dynamic control plane.

Challenges. We enumerate some of the implementation-related challenges we encountered and how we solved them. These challenges mainly arise from either limitations in the data plane programming language (P4) itself, or from restrictions imposed by the programmable switch architecture.

- *Modifying the forwarding state at line rate:* In P4, the content of the forwarding tables is provisioned by the control plane through dedicated APIs. Unfortunately, the content of these match-action tables cannot be modified at line rate unless the hardware architecture supports it. We solved this challenge by making the LPM match-action tables (TCAM) point to stateful objects (i.e., registers implemented using SRAM), which can then be modified at line rate.

- *Loops:* P4 does not allow loop constructs. We address this by unrolling loops and performing each iteration step in parallel. If the loop is longer than the maximum number of parallel steps supported by the switch, we recirculate the packet.

- *Generating packets:* P4 does not enable to instruct the switch to generate packets[3]. To address this limitation, we use the actual traffic as a carrier for our protocols. If not enough traffic is present, we periodically send empty packets from the network edge. Alternatively, if available, one can use the switch's internal traffic generator.

- *Parsing limits:* Current hardware switches can parse up to 300 bytes per packet to maintain line rate [170], hence limiting the amount

3 Specific targets are equipped with an internal traffic generator that can be programmed with APIs

of data switches can exchange in a packet. We use this limit as a constraint in our design, mandating the switches to generate smaller packets.

5.2.2 *Intra/inter-domain routing . . . in hardware!*

We now describe the key insights behind our path-vector implementation and how it manages to compute intra-domain and inter-domain paths.

Computing intra-domain routes Each switch keeps the best cost, path and output port towards every other switch in stateful registers (see Figure 5.1). Switches periodically advertise a vector $[(ID_i, cost_i, path)_i, ...]$. To generate it, the switch reads a fixed amount of register entries and pushes them into a new header. If the number of switches is bigger than the maximum number of times we can read a register, we recirculate the packet. Once the entire vector is placed into the packet, the switch sends it to all its neighbors.

Upon receiving a vector, a switch parses a fixed amount of fields and runs the shortest path computation in parallel for all of them. Specifically, the switch checks if the cost stored plus the cost to reach the advertising neighbor is smaller than the advertised cost. To avoid count-to-infinity, the switch also verifies that it is not present in the path. If both hold, the switch updates its register with the new cost, path and output port. This process is repeated until the entire vector is processed. If any cost is changed during the updating phase, the switch generates an advertisement.

Our prototype assumes that the switch receives a failure notification from the notification system. Upon receiving it, the switch iterates through its distance vector register and forgets all the routes using that link. Finally it generates an advertisement.

Computing inter-domain routes. To compute new egresses for inter-domain routes, switches keep both: *(i)* a register that maps a prefix to the best exit point known in the network; and *(ii)* a register with prefixes that the switch can reach from its external peers. To support the normal BGP decision process, switches also keep the AS path length for each route along with the type of peering relationship (e.g., customer/peer/provider) for all the egress points.

The computation process is triggered once a switch receives a prefix withdrawal from one of its peers or from the notification system running Blink. It then proceeds in two steps. First, it broadcasts a special packet indicating that the prefix cannot be reached through that egress. Second, the switch removes the corresponding route (if it exists).

Upon receiving a broadcasted prefix withdrawal, a switch looks at its registers to check whether it affects its egress. If so, it removes the route. If it uses another egress or if it knows how to reach the prefix via one of its direct peers, the switch broadcasts a message announcing the backup egress.

Each switch runs a BGP-like route selection algorithm upon receiving a backup announcement and compares the best route they currently know with the advertised one. Route selection is done as follows: if the local preference is higher, the egress is accepted as a backup; if the local preference is equal, the egress with the shortest AS path length is selected; if the path lengths are equal, the shortest distance to the egress is used; otherwise, the route is rejected. Besides computing the best egress point, upon an egress update, switches also immediately block all the traffic that violates export policies (e.g., traffic from a peer to a peer).

5.2.3 *Case study*

We now show that our implementation enables programmable switches to converge on their own upon different failures.

Methodology. We use a small topology consisting of 5 internal switches running our hardware-based control plane algorithms (Figure 5.2). Each switch is externally connected to either one customer or one peer.

We generate two TCP flows, one from AS1 and one from AS2, both flows have network X (in AS7) as a destination. To show that switches can react autonomously to internal and external failures, we introduce two events at different times. First, we fail the internal link S2-S3, which will trigger the intra-domain computation. Then, after some seconds, we send a withdrawal for prefix X to S1 from AS3, henceforth triggering the inter-domain computation and enabling the switches to find the second best egress for destination X.

We start the experiment with a converged network in which the control plane has populated the forwarding register that maps external prefixes to

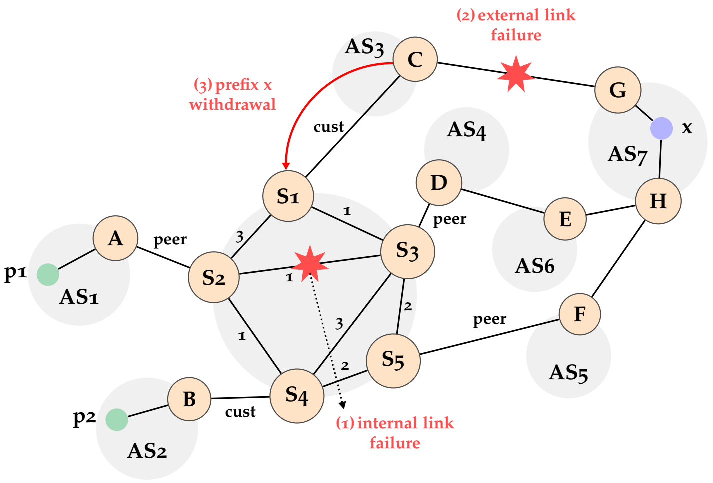


FIGURE 5.2: Case study topology illustrating a network with programmable switches and various scenarios. The topology shows 5 internal switches (S1-S5) running our hardware-based control plane, and connected to external Autonomous Systems (AS1-AS7). Customer (cust) and peer relationships are indicated. Three key events are highlighted: (1) internal link failure between S2 and S3, (2) external link failure between C and G, and (3) a withdrawal notification from AS3 to S1. Traffic originates from AS1 (p1) and AS2 (p2), both destined for network X in AS7.

the best egress IDs using BGP. Each switch also stores in memory which external prefixes can be reached via itself. To avoid being CPU bounded during the study, we set the bandwidth of every link to 10Mbps.

Results. We study how and for how long failures affect traffic that crosses our hardware-based control plane network. Figure 5.3 depicts the throughput observed over the link S1-AS3 and S5-AS5. Initially, we see that both flows are using S1-AS3 to leave the network (i.e., using the customer link) and, as such, get on average a throughput of 5Mbps.

We first fail the link S2-S3 and send a notification to the affected switches 200 ms after the failure, which triggers the intra-domain routing algorithm. As we can see in Figure 5.3 (left), the failure affects both flows for a short period of time, mainly due to the detection delay.

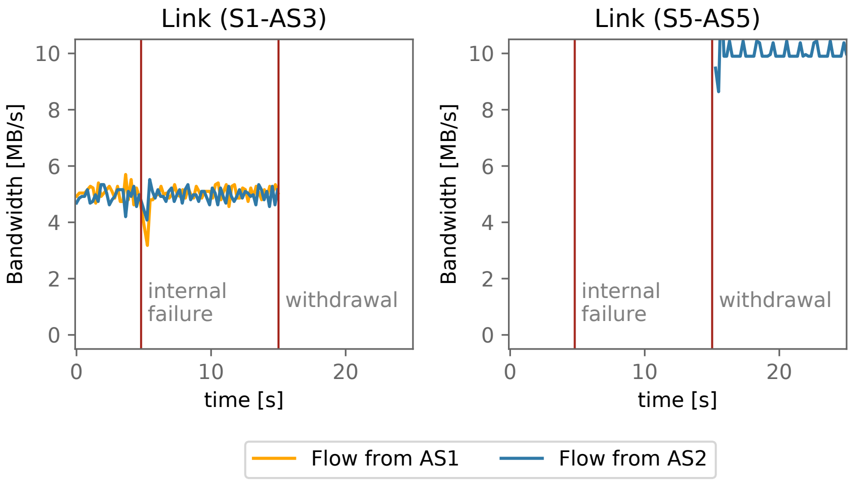


FIGURE 5.3: Per flow bandwidth at two egress points towards prefix X. Red vertical lines indicate network events

We then fail the link S1-AS3 by sending a withdrawal to S1. S1 immediately removes its route and starts dropping packets.[4] S1 then broadcasts that network X cannot be reached, making S3 and S5 broadcast back their alternative egress point. This in turn triggers the inter-domain route selection algorithm on all switches. Since S3 and S5 have the same local preference, the tie is broken using the AS path length making S5 the preferred egress. As S5-AS5 is a peer link, only the customer flow from AS2 is allowed (due to BGP export policy violations). Accordingly, we can see in Figure 5.3 (right) that S1 stops forwarding traffic and that the flow coming from AS2 starts egressing at S5-AS5 at 10Mbps.

Overall, we see that our data-plane implementation is able to automatically converge while respecting the BGP policies.

4 We leave for future work the implementation of a mechanism to maintain connectivity while learning the backup egress.

5.3 hardware is not "all roses"

In this section we discuss the pros and cons of offloading control plane tasks to hardware.

The pros. A key motivation to offload control plane tasks to programmable hardware is that most control plane operations are compatible with programmable hardware's capabilities. In our approach, for example, sensing, notification and computation are implemented by exchanging packets of a given format, processing them in a predefined way, updating the hardware state, and generating packets of potentially a different format as a result. Receiving, elaborating, and generating packets is exactly what the hardware is powerful at.

In addition, it is very natural for the hardware implementation of control plane tasks to be driven by data-plane traffic, so that the forwarding state is computed and updated according to the actual data traffic. In our prototype, forwarding entries tend to be updated in an order consistent with per-destination traffic volumes: since packets trigger actions from the hardware-based control plane, traffic for destinations carrying more traffic is probabilistically rerouted first. This produces fewer packet losses than updating forwarding entries in a random order, as software control planes often do.

Even better, running the control plane in hardware unlocks capabilities that *cannot* be easily implemented otherwise, such as the cheap and prompt detection of gray failures (Section 5.1). In fact, state-of-the-art approaches to detect gray failures either generate and post-process a huge amount of data-plane traffic, like [11], or do use programmable hardware to track packets as they cross different devices [42].

Maintaining connectivity *during* failures is another case where hardware offloading is strictly needed as waiting for the control plane to react would necessarily lead to packet losses. This is the reason why existing fast-reroute frameworks, like [150], pre-load backup paths in the switches, so as to activate them, in hardware, as soon as the failure is detected. Of course, pre-loading backup states consumes a lot of memory and is generally not scalable with respect to the exponential number of possible failure cases. Recent works, like DDC [166], show that performing control-plane computations in hardware enables to break this otherwise-fundamental trade-off between switch memory and reaction time.

Finally, being able to make forwarding decisions entirely in the data plane, without any control plane or controller, can be critical in environments where microseconds matter. For example, in data center networks where traffic loads change rapidly, decisions have to be taken almost instantaneously. Having a control-loop that goes though a software control plane leads to outdated decisions. Recent research, has shown that being able to load-balance traffic entirely in the data plane is not only possible, but surprisingly simple and effective (e.g., [39, 155, 171]).

In general, the investigation of additional use cases opened by the hardware implementation of control plane capabilities is an interesting direction for future research.

The cons. Hardware offloading is not infinitely expressive: some tasks *cannot* be delegated to hardware. For example, hardware sensing cannot be used for detecting software failures, hence detection and reaction mechanisms to these types of failures must remain in software.

Also, even when technically possible, offloading tasks to hardware might not be desirable. For example, it makes little sense to implement protocols like BGP and the underlying TCP in hardware. First, a hardware implementation would consume many hardware resources for little or no gain—especially if we consider that BGP performance is often limited by the TCP's internal algorithms [172]. Second, performance and capabilities cannot be radically changed without revisiting the implementation of the protocol on multiple administrative authorities.

For the remaining control plane tasks for which offloading to hardware can come with benefits, a major limitation is represented by the scalability of hardware implementations, a characteristic for which a software component of the control plane is likely to be needed in many realistic settings. In particular, hardware offloading is likely to scale poorly with the number of control plane tasks. On the one hand, hardware resources, like ASICS registers or memory, are typically scarce, and hard (and expensive) to scale. On the other hand, offloading control plane tasks are likely to consume a lot of hardware resources, e.g., because of the need to store messages, data, and computation parameters in hardware. Combined together, these two factors create the need for limiting the number of tasks offloaded to hardware, and hence to accurately select which functions to offload to hardware.

Carefully, and perhaps dynamically, allocating resources to different hardware computations is an interesting challenge to address in future research.

5.4 HARDWARE-SOFTWARE CODESIGN MEETS CONTROL PLANES

So far, we have shown the benefits but also the limitations of offloading tasks to hardware. This duality indicates that *accelerating* the control plane by offloading *some* tasks to hardware and keeping others in software can lead to control plane design points of great practical interest.

Our vision is that the search for an optimal design point can be formalized as a hardware-software codesign problem and solved using the classical 4-phases methodology [173]: specification, analysis or optimization, synthesis and validation. Instantiating this methodology to our context is a challenging problem that calls for interesting future research contributions.

Figure 5.4 illustrates our vision of the first 3 phases of the hardware-software codesign problem in the context of networking. More specifically, the specification phase requires precise models of the current control plane functions (e.g., failure detection, routing, and updates). These models should allow for the efficient evaluation of the performance and cost of performing each function in software, hardware, or a mix of both. Interestingly, realistic models and cost functions must take into account the dynamic interaction between distinct control plane components, which potentially makes the cost of each specific design higher than the cost of running each component separately. Furthermore, these models should also account for the cost of "hybridizing" control plane tasks by allocating some parts in software and others in hardware (e.g., accounting for the cost of synchronizing both entities).

Likewise, the analysis and synthesis phases call for the design of efficient search heuristics that leverage domain-specific knowledge to navigate the exponential space of possible hardware-software codesigns (a problem known to be NP-hard [174]). In particular, we plan to explore if it is possible to learn probabilistic models of the likelihood that a particular design is better than another.

Finally, the validation of (partially) offloaded control planes opens up interesting verification questions such as how to ensure that a specific design will perform accordingly both feature- *and* performance-wise.

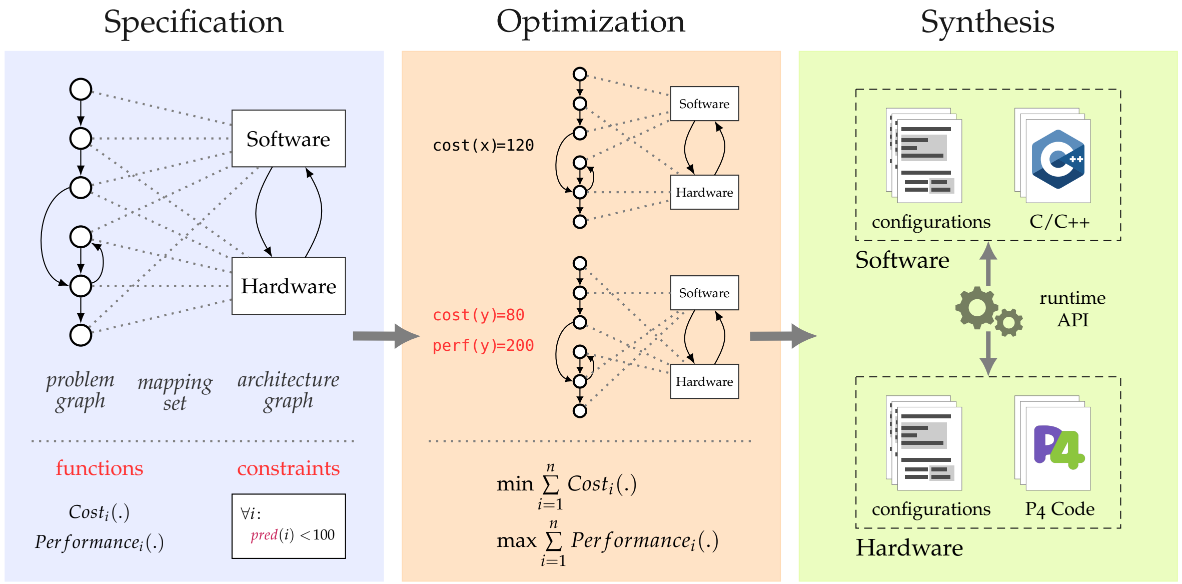


FIGURE 5.4: Example of our vision for the codesign pipeline for control plane optimization, detailing: (i) Specification phase, where control plane functions and possible interactions are modeled; (ii) Optimization phase, which employs these models to determine the best trade-offs between performance and cost analytically; and (iii) Synthesis phase, which generates the optimal software and hardware configurations, code, and runtime interfaces for efficient control plane acceleration.

6 CONCLUSION AND OUTLOOK

In this dissertation, we explored the significant yet overlooked impact of gray failures in ISP networks, a problem previously only explored in the context of data center networks. We identified that recent advancements in programmable network data planes provide a unique opportunity to implement scalable failure detection systems directly within the network infrastructure. We developed a system capable of detecting and localizing gray failures in high-speed and high-delay environments such as ISP networks. Furthermore, we demonstrated that programmable data planes are not only useful for failure detection, but can also accelerate network recovery by automatically rerouting traffic in the data plane.

In Chapter 3, we provided an overview of network failures, specifically putting an emphasis on gray failures and the different ways they manifest. This chapter established a foundation for understanding the complex nature and significant impact of such network disruptions, and the necessity for better detection systems. Through an analysis of bug reports from leading ISP router vendors, we illustrated that even the most advanced networking equipment is susceptible to gray failures. A survey conducted among ISP operators confirmed that gray failures are a common and frequent issue in their ISP networks, emphasizing the need for detection systems designed with the challenges of these specific networks. Finally, we highlighted the limitations of existing detection techniques, including those considered state-of-the-art for data center networks, when applied to high-traffic and high-delay networks such as ISPs.

In Chapter 4, we presented FANcY, a novel data plane system designed to detect and localize gray failures in ISP networks efficiently. FANcY leverages programmable switches to implement a reliable inter-switch synchronization protocol, enabling switches to exchange and compare counters to detect packet losses precisely. With each counter, FANcY tracks individual traffic entries, such as IP prefixes. However, direct monitoring of all prefixes is unfeasible due to the limited memory capacity of switches. To address this limitation, we introduced a hybrid solution with two modes of operation: (i) dedicated counters for high-priority traffic, ensuring fast and accurate monitoring, but requiring memory for each monitored entry, and (ii) hash-based trees for best-effort entries, optimizing for memory with constant use, but potentially slower and reduced accuracy. We showed that FANcY's reliable synchronization protocol and hybrid counting approach make it not only ideal for ISP networks, but also a future-proof design for the constantly increasing traffic volumes. Our extensive evaluations, conducted via simulations and with a prototype running on an Intel Tofino switch, demonstrated that FANcY enables sub-second detection and reaction (i.e., rerouting) upon gray failures, except in instances where the failure causes minimal packet losses.

In Chapter 5, we went one step further and explored the feasibility of offloading control plane tasks traditionally implemented in software into hardware, utilizing programmable data planes. We showed that programmable data planes are not only an excellent technology that can help improve detection systems, as shown with FANcY, but can also implement a broader range of control plane tasks. With a working prototype, we demonstrated that programmable data planes can efficiently handle diverse tasks: (i) detect regular, gray (FANcY) and remote failures (Blink); (ii) notify other devices; (iii) run simple distributed path-vector computations that support both shortest-path and BGP-like policies; and (iv) update its forwarding state, enabling switches to restore connectivity after a failure. Finally, we concluded with an analysis of the advantages and disadvantages associated with offloading functions to programmable data planes. We found that despite their potential, certain tasks might not benefit from or are currently not suited for offloading due to current limitations in expressiveness and scalability. This realization makes task selection an open challenge worth investigating in future research.

6.1 OPEN RESEARCH PROBLEMS

In this section, we explore open research problems and propose future enhancements for the systems detailed in this dissertation and any future systems that leverage data plane offloading. Initially, we propose three research directions to improve FANcY and failure detection in general. Subsequently, we discuss two emerging challenges introduced by the decentralization of state and computational resources across different network planes.

6.1.1 *Integrating the control plane with* FANcY

FANcY is a data-plane-only solution with minimal interaction with the control plane. We identify opportunities to integrate the control plane into the detection loop. One such opportunity is for the control plane to initiate periodic active measurements for selected traffic entries, enabling faster or preemptive (even without traffic) failure detection. This mechanism can use the built-in traffic generator and FANcY's dedicated counters. Another potential research direction could be implementing root cause analysis and tools for guided or automatic in-depth debugging. Currently, upon detecting a failure, FANcY only informs the control plane and flags the affected data plane entry to activate a backup route. Future research could expand on this by leveraging the control plane to analyze reported failure events over time and pinpoint the underlying cause of the problem, or if needed, start an automatic debugging process to gather further relevant information. Finally, with that information, the control plane could either attempt self-repair or generate a comprehensive report that operators could use for further manual debugging or repair.

6.1.2 *Enhancing "weak" traffic signals with adaptive traffic generation*

FANcY's detection accuracy highly depends on the traffic volume it monitors. In scenarios of gray failures, where only a subset of packets may be dropped, a sufficient traffic volume is crucial for generating a "strong" failure signal. Furthermore, the volume of traffic, and consequently its loss signal, tends to decrease during failures. This reduction is due to the nature of Internet traffic, dominated by TCP flows that reduce their sending rate following packet losses, diminishing the failure signal and making failures harder to detect. FANcY's evaluation reveals that while this is not a predominant issue for entries monitored with dedicated counters, it poses a significant challenge for hash-based trees, which rely on detecting packet drops across multiple consecutive intervals. We believe that we could mitigate this by enhancing passive detection systems, such as FANcY, with active, adaptively triggered (i.e., only when needed) traffic generation.

While several methods for generating traffic exist, we believe in-network generation, particularly from programmable switches, offers a unique advantage due to its inherent in-network visibility and potential direct interaction with the monitoring system already running in the switch. That would allow the generation of traffic in reaction to identified weak signals or as requested by the monitoring system. For instance, FANcY could request a signal enhancement when the zooming process does not successfully finish. Our preliminary work [175] demonstrates that today's programmable data planes are capable of fulfilling these requirements. They can generate adaptive stateless and stateful traffic (emulating the TCP state machine), potentially benefiting FANcY and other passive in-network applications reliant on robust data plane signals. Future work could dive into better combining passive in-network systems with adaptive in-network traffic generation to boost their performance.

6.1.3 *A mixture of detectors*

While the state-of-the-art data center gray failure detectors such as LossRadar [42] and NetSeer [17] excel at detecting packet losses, their applicability within ISPs faces limitations due to the significant memory requirements in high-bandwidth and high-delay networks (see details in Section 4.1.3). FANcY, in comparison, offers the flexibility of detecting packet losses in a more diverse set of environments but requires a minimum amount of traffic to operate properly. Inspired by the *mixture of experts* concept from machine learning [176], where distinct models are tailored to handle specific subsets of the input data, we propose a hybrid approach to failure detection and envision the concept of *mixture of detectors*. For example, with this approach one could utilize FANcY for initial traffic monitoring through its hash-based tree, and upon partial detection issues (e.g., incomplete zooming), redirect the relevant traffic entries for further scrutiny by systems such as LossRadar or NetSeer, which thanks to the reduced traffic input would not face limitations. Future research could focus on finding the optimal way of integrating different detection systems to leverage their individual strengths while avoiding their weaknesses and creating a robust and versatile failure detection system.

6.1.4 *Towards seamless smart network planes integration*

Shifting to network designs where both control and data planes independently compute and update their states may temporarily introduce state inconsistencies, commonly referred to as "split-brain" in distributed systems. In our work on hardware-accelerated control planes, we show that while the data plane can autonomously compute forwarding states and recover from failures, it might result in selecting suboptimal paths due to limitations in computational complexity and available memory. The control plane, typically slower but not subject to such limitations, has to ensure that the data plane runs in the optimal state whenever possible. However, improper coordination between control and data plane can lead to inconsistent data plane states, potentially causing network disruptions such as temporary blackholes, routing loops, or violations of network policies. An interesting area for future research could involve a comprehensive analysis of the interface requirements between "smart" network planes across a wide range of network applications, including identifying existing limitations and requirements, and developing a set of primitives to facilitate the seamless integration between network planes.

6.1.5 *Optimizing the network plane's slow path*

Another significant challenge that arises from distributing intelligence across different layers of the networking stack is the resultant increase in required communication between the data and control plane. As detailed in our thesis (see Section 3.2.2.2), gray failure detection systems that rely on high-frequency polling of data plane data structures, are already significantly impacted by the high-delay and low-bandwidth between the control and data planes. This situation is further aggravated by the constant increase in the data plane's processing capabilities, bandwidth, and topology complexity, which directly translates into an increase in the required data exchange between the control and data planes. Therefore, the slow path between the control and data planes, which traditionally has not been considered critical and has received little attention, is becoming one of the biggest bottlenecks. This issue underscores the pressing need for dedicated research toward improving this part of the communication stack. Future work should focus on identifying standard primitives for infrastructure control protocols and designing an architecture that balances flexibility with performance, ensuring predictable response times, efficient handling of common data structures, and fast data plane table updates.

A APPENDIX

A.1 CISCO AND JUNIPER BUG LIST.

This appendix presents a comprehensive list of bugs associated with gray failures in Cisco and Juniper network devices. Each entry includes the following information: vendor, bug code, root cause, and a brief description of the bug's impact on network traffic.

Table A.1: List of reported bugs on Cisco and Juniper bug portals.

ID	Vendor	Bug Code	Root Cause	Short Description
1	Cisco	CSCsz10107	Software bug	Some connections get dropped when the Cat6k with ACE module is reloaded.
2	Juniper	PR1423310	Configuration issue	IPv6 multicast traffic might get dropped if ig and eg are on different VC members.
3	Juniper	PR1308438	Others	With large scale routes(e.g., 650k) GRE tunneled traffic might get dropped.
4	Juniper	PR1161485	Configuration issue	L3 multicast traffic experiences continuous drops after a down/up of an interface.
5	Cisco	CSCtb21313	Configuration issue	Connections get dropped after rebalanced to a different L7 policy.
6	Juniper	PR1459692	Others	In a MC-LAG scenario, VRRP-virtual MAC traffic gets dropped by PFE.
7	Cisco	CSCta03825	Configuration issue	When UDP booster is enabled, every first connection packet is dropped.
8	Juniper	PR1425927	Configuration issue	Drops in encapsulation flexible Ethernet services on specific interfaces.
9	Cisco	CSCsl82712	Configuration issue	HTTP method request exceeds the configured maximum HTTP header length.

Continued on next page

Table A.1 – continued from previous page

ID	Vendor	Bug Code	Root Cause	Short Description
10	Cisco	CSCve98991	Software update	Traffic outage seen after the image on the line card module is upgraded.
11	Juniper	PR1463092	Software bug	When deleting the IRB not removed from PFE leading to traffic blackhole.
12	Cisco	Nan	Hardware malfunction	Specific Fabric module causes packet drops during a power cycle.
13	Juniper	PR1402626	Software bug	VLAN tagged traffic on VPLS interface dropped due to PFE programming failure.
14	Cisco	CSCsm34992	Configuration issue	Connections dropped post policy map action mod from SIP/RTSP/Skinny to HTTP.
15	Juniper	PR1472083	Software bug	Traffic loss on MX with EQ MPC under fusion scenario with 'rate-limit' CoS policy.
16	Cisco	CSCuz95179	Hardware malfunction	UDP packets with specific port ranges or fragmented frames get dropped.
17	Cisco	CSCvr35120	Others	Cisco ACI fabric drops incoming non-VxLAN traffic with specific UDP port.
18	Juniper	PR1450545	Configuration issue	Traffic loss might occur when there are around 800,000 routes in FIB.
19	Juniper	PR1436119	Software bug	Traffic loss on MX/PTX post rapid LDP session flaps and ecmp-fast-reroute enabled.
20	Juniper	PR1456905	Configuration issue	Traffic loss in seamless MPLS with pseudo-wire and MVPN processing and GRE.
21	Cisco	CSCvq60859	Hardware malfunction	Traffic loss line card post-reload, due to HW programming delay in selective VRF.
22	Juniper	PR1327062	Software bug	Drops during initial ARP refresh in EVPN-VXLAN multi-homed CE.
23	Juniper	PR1313977	Hardware malfunction	Traffic drops due to CRC error.
24	Cisco	CSCvk01435	Configuration issue	Traffic for specific multicast group dropped when the PTP feature is disabled.
25	Juniper	PR1439251	Software bug	Traffic blackholing due to delayed PathTear message from Juniper LER.
26	Juniper	PR1376057	Software bug	Pass-through traffic dropped due to routers using indirect next hop and LB.
27	Juniper	PR1403727	Others	TCP traffic experience drops and increased latency.
28	Juniper	PR1444186	Software bug	GRE packets larger than MTU are dropped when sampling is enabled.
29	Juniper	PR1446132	Software bug	Dynamic tunnels with ECMP send traffic with incorrect VLAN IDs, leading drops.
30	Juniper	PR1430685	Software bug	Enabling TCP proxy-based and rst-invalidate-session features lead to TCP reset drops.
31	Juniper	PR1387895	Configuration issue	Changing MTU configuration leads to packet drops of the SUN-RPC traffic.
32	Juniper	PR1364657	Hardware malfunction	Improper device state leading to malfunctioning PE blackholing specific IPs.
33	Juniper	PR1379734	Software bug	Configuring sampling or pkt capture leads to packet drops.
34	Juniper	PR1420103	Hardware malfunction	Post-RE switchover with GRES and NSR, LDP label corruption leads to BGP session drops.

Continued on next page

Table A.1 – continued from previous page

ID	Vendor	Bug Code	Root Cause	Short Description
35	Juniper	PR1458499	Software bug	Timing issue in updating firewall filter leads to traffic blackholes or MPC crashes.
36	Juniper	PR1428935	Software bug	Post-GRES delay in BPDU transmission causes traffic loss.
37	Cisco	CSCvu02712	Hardware malfunction	Intermittent CRC errors leading to packet drops.
38	Juniper	PR1440847	Configuration issue	After device reboot DDOS limits get to default values. Can lead to drops.
39	Juniper	PR1450928	Configuration issue	ARP packet drops by PFE after chassis restart.
40	Juniper	PR1348029	Software bug	ARP update failure leading to packet drops at routing engine.
41	Cisco	CSCvn53560	Configuration issue	Packets drops on ToR switch when returned from service device on 2nd path.
42	Juniper	PR1475031	Software bug	After payload changed by SIP, fragmented packets might get dropped.
43	Juniper	PR1459698	Others	Silent dropping of traffic upon interface flapping after DRD auto-recovery.
44	Cisco	CSCsy84895	Configuration issue	ACE drops server packets larger than advertised MSS.
45	Juniper	PR1470619	Software bug	RED drop could be seen on an interface even when there is no congestion.
46	Cisco	CSCvd43653	Hardware malfunction	Random frame drops on MDS 9700 platforms due to timeout in port buffers.
47	Juniper	PR1455388	Hardware malfunction	QSFP-100G-SR4 transceivers on QFX5110 cause CRC errors and packet loss.
48	Cisco	CSCea91692	Software update	PSA corrupted entry. Affects IP traffic for specific line card.
49	Cisco	CSCvm59661	Hardware malfunction	Incorrect L2 entry leading to intermittent reachability issues.
50	Juniper	PR1422877	Software bug	PDP context response messages dropped due to packet size issues.
51	Juniper	PR1407424	Software bug	Packet drops after setting packet filter and RPD restart.
52	Juniper	PR1394085	Configuration issue	Packet loss might occur on unrelated traffic when AppQos rate limiter applied.
53	Juniper	PR1429899	Hardware malfunction	FPGA back pressure on with SPC3 cards leads to minor packet loss.
54	Cisco	NaN	Others	Packet drops can occur in CSR if any of the CPUs reaches 100% utilization.
55	Juniper	PR1474674	Configuration issue	Packet drops when add/del interfaces from MACsec with specific settings.
56	Juniper	PR1421857	Configuration issue	Wrong config speed leads can lead to traffic drops for that interface.
57	Juniper	PR1423989	Software bug	Adding an unconnected port to an existing LAG might cause packet drops.
58	Juniper	PR1387746	Software bug	After link flap, router does not install BGP LU label causing traffic to be dropped.
59	Juniper	PR1443345	Others	With source NAT and under high traffic load, TCP-SYN packets might be dropped.

Continued on next page

Table A.1 – continued from previous page

ID	Vendor	Bug Code	Root Cause	Short Description
60	Juniper	PR1398407	Others	BGP packets might be dropped under high CPU usage.
61	Juniper	PR1462825	Software bug	Wrongly calculated MTU leads to packets being dropped.
62	Juniper	PR1475395	Software bug	Traffic blackhole in L3 VPN when destination resolved with two LPSs.
63	Juniper	PR1338444	Configuration issue	Enabling/disabling ICCP/ICL link leading to ARP learning problems.
64	Juniper	PR1441047	Hardware malfunction	Specific UDP port packets dropped due to erroneous VXLAN filter hit.
65	Juniper	PR1309613	Hardware malfunction	Traffic loss might be seen while CRC errors of the same interface keep increasing.
66	Juniper	PR1177499	Hardware malfunction	Packet loss and framing errors with QSFP+40GE-LX4 transceiver.
67	Juniper	PR1429543	Software bug	Specific types of genuine IPv4/6 traffic are improperly filtered and discarded.
68	Juniper	PR1169700	Memory corruption	Parity error on interfaces affecting MMU unit memories and corrupting counters.
69	Juniper	PR1433300	Hardware malfunction	Traffic loss on LC1105 line card when MACsec is configured and handling PF.
70	Juniper	PR1289546	Configuration issue	Temporary drops after deleting and adding 1K LAG interfaces.
71	Juniper	PR1409631	Software bug	Intra-VLAN traffic loss when restarting FPC with MC-LAG enhanced-convergence.
72	Juniper	PR1348659	Configuration issue	Traffic blackhole with EVPN-VXLAN and VRRP when deleting IRB intf.
73	Juniper	PR1296089	Memory corruption	Upon config changes, fd memory corruption leading to traffic loss on certain ports.
74	Juniper	PR1243724	Memory corruption	Memory corruption during route-next hop or EDF job, leading to packet drops.
75	Juniper	PR1460406	Hardware malfunction	Transient voltage fluctuations trigger fabric healing process leading to drops.
76	Juniper	PR1433648	Software bug	Firewall config changes cause transit packet drops in PFE due to timing issues.
77	Juniper	PR1447170	Software bug	Resource exhaustion due to unfreed data structures leading to packet loss.
78	Juniper	PR1401802	Software bug	Unexpected multicast packet drops when active RPF path is disabled.
79	Juniper	PR1388082	Hardware malfunction	Intermittent loss when flapping RTG primary interface.
80	Juniper	PR1466659	Software bug	IPv6 traffic loss in L3VPN networks on specific configuration.
81	Juniper	PR1409773	Software bug	Transient traffic loss with MC-LAG during routing daemon restart with new config.
82	Juniper	PR1443466	Configuration issue	RED traffic drops following a link flap or CoS configuration change.
83	Juniper	PR1410233	Configuration issue	Packet drops due to reroute failures in ECMP routing.
84	Juniper	PR1459446	Software bug	Traffic blackhole during link recovery in open Ethernet access ring.

Continued on next page

Table A.1 – continued from previous page

ID	Vendor	Bug Code	Root Cause	Short Description
85	Juniper	PR1389120	Software bug	Unexpected multicast packet drops during failure recovery.
86	Juniper	PR1427842	Software bug	Packet drops at the time of routing engine switchover if system up for long.
87	Juniper	PR1329141	Configuration issue	CoS incorrectly applied on PFE leading to egress packet drops on some intfs.
88	Juniper	PR1426734	Hardware malfunction	Resource leakage on ARP table leading to 50% of entries experiencing drops.
89	Juniper	PR1436494	Configuration issue	Traffic drop might be seen after deactivate/activate “class-of-service".
90	Juniper	PR1295774	Configuration issue	TCP connection drops during large file transfers at high speeds through MP.
91	Cisco	CSCvg17452	Software bug	Router incorrectly programs egress LIF for VLAX, leading to traffic drops.
92	Cisco	CSCvt25313	Configuration issue	Inter-pod traffic drop on spine switches, due to tunnel nh set to global bounce.
93	Cisco	CSCvt56182	Software bug	Transient drops during ND ISSU particularly when BFD enabled.
94	Cisco	CSCtx61116	Software bug	After upgrade, NAT unreasonably drops all traffic for random source ports.
95	Cisco	CSCvk38405	Software bug	Fragmented PIM BSR packets punted to CPU and dropped.
96	Cisco	CSCvr30525	Software bug	Mcast Traffic Loss To All Receivers After One Receiver Sends Multiple Leafs.
97	Cisco	CSCvs50407	Software update	After OS upgrade. Multicast traffic drop on ISSU.
98	Cisco	CSCvs06516	Hardware malfunction	Multicast group not programmed in hardware leading to traffic drop.
99	Cisco	CSCvg34717	Hardware malfunction	Multicast CP packets are dropped by F2/F3 module.
100	Cisco	CSCvd44475	Software bug	Multicast traffic loss during switch ID change.
101	Cisco	CSCvq04585	Software bug	Multicast traffic loss with module reload and other triggers.
102	Cisco	CSCvh87462	Hardware malfunction	MIPv6 packet of 320B size dropped by M3 module as invalid.
103	Cisco	CSCvf86400	Software bug	killing lisp manually on a scaled ITR/ETR configuration causes traffic loss.
104	Cisco	CSCuv31196	Software bug	Unicast IP packets with specific IP ID get dropped.
105	Juniper	PR1427866	Configuration issue	IPv6 traffic might be dropped when static /64 Ipv6 routes are configured.
106	Juniper	PR1434757	Software bug	Intermittent packet drop might be observed if IPsec is configured.
107	Juniper	PR1469596	Software bug	Ingress traffic dropped in EVPN-VXLAN when interface flaps causing blackhole.
108	Juniper	PR1455973	Configuration issue	Traffic loss observed during seamless migration from VPLS to EVPN.
109	Juniper	PR1392261	Software bug	DHCP packets get drooped with specific DHCP configuration.

Continued on next page

Table A.1 – continued from previous page

ID	Vendor	Bug Code	Root Cause	Short Description
110	Juniper	PR1461983	Software bug	Extended traffic loss during NSSU.
111	Juniper	PR1350103	Hardware malfunction	Traffic drop when MIC is physically removed and reinserted in an MPC.
112	Juniper	PR1416487	Software bug	Traffic silently dropped due to long LSP switchover during RSVP-signaled LSP.
113	Juniper	PR1444845	Hardware malfunction	CRC errors after VC connections are disrupted, due to improper VCP init.
114	Juniper	PR1447187	Software bug	Multicast traffic loss in PIM with BGP PIC when a link flap occurs.
115	Juniper	PR1452866	Software bug	Traffic blackhole after LACP timeout in LACP with Unilist next-hop scenario.
116	Juniper	PR1462583	Configuration issue	Traffic loss in l2circuit with local-switching due to MTU mismatch.
117	Juniper	PR1417139	Configuration issue	Traffic blackhole in JunosFusion with dual-AD due to ICL link not coming up.
118	Juniper	PR1434567	Hardware malfunction	IPv6 neighbor solicitation packets getting dropped due to hardware party errors.
119	Juniper	PR1311773	Software bug	Traffic loss from IP Fabric to EVPN in collapsed L2/L3 multi-homed GW topology.
120	Juniper	NaN	Configuration issue	Traffic drop when configuring GRE with ECMP NH instead of unicast NH.
121	Juniper	PR1441402	Configuration issue	Traffic drop after QinQ interface flap or vlan-id-list change.
122	Juniper	PR1395186	Configuration issue	PTPoETH traffic dropped when IGMP and PTP are configured on the same VLAN.
123	Juniper	PR1355878	Software bug	MPLS routes become dead after quick enable/disable or label number change.
124	Cisco	CSCsu42225	Software bug	UDP packets with 32000 bytes of payload dropped when load balanced.
125	Juniper	PR1454907	Software bug	Temporary traffic drop when modifying large number of configured policies.
126	Cisco	CSCvr11055	Software bug	GRE traffic with incorrectly formed IP header payload dropped.
127	Juniper	PR1468570	Configuration issue	FTP data connection timeouts when FTP traffic is routed through the dialer intf.
128	Cisco	CSCtg98720	Software bug	Fabric plane connectivity issues and packet drops due to a system time change.
129	Juniper	PR1429714	Hardware malfunction	Traffic loss due to random fabric drops when packet traverse different FPCs.
130	Juniper	PR1231402	Software bug	Incorrect PE router attached to ESI in EVPN/VXLAN causing partial blackholes.
131	Juniper	PR1322288	Software bug	Permanent loss for some hosts due to unsynchronized ARP entries after expiry.
132	Juniper	PR1441816	Hardware malfunction	Egress stream flush and traffic blackholes during repeated link flaps.
133	Cisco	CSCvs19509	Software bug	CP traffic loss when dst IP set via a static route through BGP-learnt EVPN route.
134	Juniper	PR1424705	Software bug	Traffic blackholing when an interface on the primary node is disabled.

Continued on next page

Table A.1 – continued from previous page

ID	Vendor	Bug Code	Root Cause	Short Description
135	Juniper	PR1359841	Software bug	Disabling a LAG member from a L3 IRB might cause traffic loss.
136	Cisco	CSCuy06749	Configuration issue	Traffic drop between two isolated EPGs affecting L2 and specific L3 packets.
137	Cisco	CSCvi58895	Hardware malfunction	CRC errors increment randomly on 10G interfaces leading to drops.
138	Juniper	PR1449406	Hardware malfunction	Unexpected CRC errors seen on the VCP, leading to system performance issues.
139	Cisco	CSCvq45166	Configuration issue	CP traffic affected by high rate of NetFlow record packets on in-band.
140	Cisco	CSCuy29638	Software bug	Whenever the AN sends a packet to a subnet without label it gets malformed.
141	Juniper	PR1282349	Software bug	ARP reply drop when recovering from local and peer MC-AE being down.
142	Juniper	PR1414509	Software bug	Traffic originated from the device itself might be dropped in IPsec tunnel.
143	Cisco	CSCsv49518	Configuration issue	ICMP packet loss on servers tagged for load-balancing VLAN.
144	Cisco	CSCsm52480	Software bug	IPv4/6 multicast traffic not bridged by ACE module in VLANs with IGMP snooping.
145	Cisco	CSCvg19938	Configuration issue	High CPU leads to drops when shutting down an interface in large scale IPv6.
146	Cisco	CSCvp01676	Software bug	Packet loss for traffic destined to prefixes learned via routing protocols.
147	Cisco	CSCti14290	Memory corruption	Drops after a router reload, upgrade or crash due to corrupted hardware-forwarding.
148	Cisco	CSCvq65959	Configuration issue	80% packets loss in route leaking environment after changing SVI IP address.
149	Cisco	CSCtc33158	Configuration issue	Drops for specific packet sizes when L2TPv3 cookies are enabled.

OWN PUBLICATIONS

[1] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. “Hardware-Accelerated Network Control Planes”. In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. HotNets ’18. Redmond, WA, USA: Association for Computing Machinery, 2018, 120.

[2] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. “FAst In-Network GraY Failure Detection for ISPs”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. SIGCOMM ’22. Amsterdam, Netherlands: Association for Computing Machinery, 2022, 677.

[3] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. “Blink: Fast Connectivity Recovery Entirely in the Data Plane”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, 161.

[4] Daniele De Sensi, Edgar Costa Molero, Salvatore Di Girolamo, Laurent Vanbever, and Torsten Hoefler. “Canary: Congestion-aware in-network allreduce using dynamic trees”. In: *Future Generation Computer Systems* 152 (2024), 70.

[5] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. “A Brief History of the Internet”. In: *SIGCOMM Comput. Commun. Rev.* 39.5 (2009), 22.

[6] Vinton G. Cerf and Robert E. Icahn. “A Protocol for Packet Network Intercommunication”. In: *SIGCOMM Comput. Commun. Rev.* 35.2 (2005), 71.

[7] U Cisco. “Cisco annual internet report (2018–2023) white paper”. In: *Cisco: San Jose, CA, USA* 10.1 (2020), 1.

[8] Holly Honderich. *Rogers outage: Why a network upgrade pushed millions in Canada offline*. `https://www.bbc.com/news/world-us-canada-62174477`. Accessed: [2023]. 2022.

[9] Celso Martinho and Tom Strickx. *Understanding how Facebook disappeared from the Internet*. `https://blog.cloudflare.com/october-2021-facebook-outage`. Accessed: [2023]. 2021.

[10] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. “Gray Failure: The Achilles’ Heel of Cloud-Scale Systems”. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS ’17. Whistler, BC, Canada: Association for Computing Machinery, 2017, 150.

[11] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. “Pingmesh: A large-scale system for data center network latency measurement and analysis”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. ACM. 2015, 139.

[12] J. Case, R. Mundy, D. Partain, and B. Stewart. *Introduction and Applicability Statements for Internet-Standard Management Framework*. RFC 3410. `http://www.rfc-editor.org/rfc/rfc3410.txt`. RFC Editor, 2002.

[13] Peter Phaal, Sonia Panchen, and Neil McKee. *InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. RFC 3176 (Informational). `http://www.ietf.org/rfc/rfc3176.txt`. Internet Engineering Task Force, 2001.

[14] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. "Packet-Level Telemetry in Large Datacenter Networks". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: Association for Computing Machinery, 2015, 479.

[15] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. "Stroboscope: Declarative Network Monitoring on a Budget". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018.

[16] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Forster, Arvind Krishnamurthy, and Thomas Anderson. "Understanding and Mitigating Packet Corruption in Data Center Networks". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, 362.

[17] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. "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*. 2020, 76.

[18] Barefoot. *Barefoot Tofino, World's fastest P4-programmable Ethernet switch ASICs*. `https://barefootnetworks.com/products/brief-tofino/`.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "Netcache: Balancing key-value stores with fast in-network caching". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, 121.

[20] R. Bellman. "On a Routing Problem". In: *Quarterly of Applied Mathematics* (1958).

[21] Gary S. Malkin. *RIP Version 2*. RFC 2453. `https://www.rfc-editor.org/info/rfc2453`. 1998.

[22] Cisco. *Interior Gateway Protocol (IGRP)*. https://www.cisco.com/c/en/us/support/docs/ip/interior-gateway-routing-protocol-igrp/26825-5.html. 2005.

[23] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* (1959).

[24] John Moy. *OSPF Version 2*. RFC 2328. https://www.rfc-editor.org/info/rfc2328. 1998.

[25] Juniper Networks. *IS-IS Routing Protocol*. https://www.juniper.net/documentation/us/en/software/junos/is-is/topics/concept/is-is-routing-overview.html. 2023.

[26] Janne Lindqvist. "Counting to infinity". In: *Seminar on Internetworking, Helsinki University of Technology Telecommunications, Software and Multimedia Laboratory*. Citeseer. 2004.

[27] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271 (Draft Standard). 2006.

[28] Benoit Claise. *Cisco Systems NetFlow Services Export Version 9*. RFC 3954 (Informational). http://www.ietf.org/rfc/rfc3954.txt. Internet Engineering Task Force, 2004.

[29] Bobbi Sandberg. "Networking The Complete Reference". In: 3rd. McGraw-Hill Education Group, 2015. Chap. Chapter 2.

[30] Jon Postel. *Transmission Control Protocol*. STD 7. http://www.rfc-editor.org/rfc/rfc793.txt. RFC Editor, 1981.

[31] W. Eddy. *Transmission Control Protocol (TCP)*. STD 7. RFC Editor, 2022.

[32] J. Postel. *User Datagram Protocol*. STD 6. http://www.rfc-editor.org/rfc/rfc768.txt. RFC Editor, 1980.

[33] P. Mockapetris. *Domain names - concepts and facilities*. STD 13. http://www.rfc-editor.org/rfc/rfc1034.txt. RFC Editor, 1987.

[34] P. Mockapetris. *Domain names - implementation and specification*. STD 13. http://www.rfc-editor.org/rfc/rfc1035.txt. RFC Editor, 1987.

[35] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. "PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric". In: ().

[36] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. "VL2: A Scalable and Flexible Data Center Network". In: *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*. SIGCOMM '09. Barcelona, Spain: Association for Computing Machinery, 2009, 51.

[37] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *SIGCOMM Comput. Commun. Rev.* 38.4 (2008), 63.

[38] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. "Fastpass: A centralized zero-queue datacenter network". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), 307.

[39] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. "Hula: Scalable load balancing using programmable data planes". In: *Proceedings of the Symposium on SDN Research*. ACM. 2016, 10.

[40] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. "DRILL: Micro Load Balancing for Low-Latency Data Center Networks". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, 225.

[41] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, 503.

[42] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. "LossRadar: Fast Detection of Lost Packets in Data Center Networks". In: *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '16. Irvine, California, USA: ACM, 2016, 481.

[43] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. "007: Democratically finding the cause of packet drops". In: *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 2018, 419.

[44] Nick McKeown. *SDN: Getting the humans out of the way*. `https://www.juniper.net/documentation/us/en/software/junos/network-mgmt/topics/topic-map/switches-interface-oam-lfm.html`. (Accessed: 2023-07-24).

[45] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Larry Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. 2014.

[46] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends". In: *CoRR* abs/2102.00643 (2021).

[47] Richard Chirgwin. *Google routing blunder sent Japan's Internet dark on Friday*. `https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/`. 2017.

[48] Yevgeniy Sverdlik. *"Configuration Issue" Halts Trading on NYSE*. `https://www.datacenterknowledge.com/archives/2015/07/08/technical-issue-halts-trading-on-nyse`. 2016.

[49] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), 69.

[50] The open networking fundation. *OpenFlow Switch Specification Version 1.5.1*. `https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf`. 2015.

[51] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. "The Design and Implementation of Open vSwitch." In: *NSDI*. Vol. 15. 2015, 117.

[52] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. "High-Speed Software Data Plane via Vectorized Packet Processing". In: *Comm. Mag.* 56.12 (2018), 97.

[53] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. "NetBricks: Taking the V out of NFV". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, 203.

[54] Linux Foundation. *Data Plane Development Kit (DPDK)*. Accessed: 2023-08-15. 2023.

[55] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, 2018, 54.

[56] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. "NetFPGA SUME: Toward 100 Gbps as Research Commodity". In: *IEEE Micro* 34.5 (2014), 32.

[57] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. USA: Prentice-Hall, Inc., 1996.

[58] Volnei A. Pedroni. *Circuit Design with VHDL*. Cambridge, MA, USA: MIT Press, 2004.

[59] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: Programming Protocol-Independent Packet Processors". In: *SIGCOMM Comput. Commun. Rev.* 44.3 (2014), 87.

[60] The P4 Language Consortium. *$P4_{16}$ Language Specification (Version 1.2.4)*. `https://p4.org/p4-spec/docs/P4-16-v1.2.4.pdf`. 2023.

[61] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN". In: *Proceedings of the ACM SIGCOMM*

2013 Conference on SIGCOMM. SIGCOMM '13. Hong Kong, China: Association for Computing Machinery, 2013, 99.

[62] Dan Daly Calin Cascaval. *P4 Architectures*. https://opennetworking.org/wp-content/uploads/2020/12/p4-ws-2017-p4-architectures.pdf. 2017.

[63] The P4.org Architecture Working Group. *Portable Switch Architecture (PSA) version 1.2*. Accessed: 2023-08-15. 2022.

[64] The P4.org Architecture Working Group. *Portable NIC Architecture (PNA) version 0.7*. Accessed: 2023-08-15. 2022.

[65] Intel. *Intel Tofino Native Architecture - Public Version*. Accessed: 2023-08-15. 2021.

[66] Intel. *Switching to Intelligence: Intel Tofino Intelligent Fabric Processors*. Accessed: 2023-08-15. 2022.

[67] The P4 Language Consortium. *P4 Language Specification (Version 1.0.5)*. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf. 2018.

[68] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends". In: *CoRR* abs/2102.00643 (2021).

[69] Open Tofino. *Tofino 1 P4 Architecture Description*. 2022.

[70] P4 Language Consortium. *Simple Switch/v1model Architecture Description*. 2021.

[71] P4 Language Consortium. *P4 Language GitHub*. https://github.com/p4lang/. 2018.

[72] P4 Language Consortium. *P4 behavioral model.* https://github.com/p4lang/behavioral-model. 2018.

[73] P4 Language Consortium. *P4c Compiler.* https://github.com/p4lang/p4c. 2018.

[74] Edgar Costa Molero and Jurij Nota. *P4Utils.* https://github.com/nsg-ethz/p4-utils. 2018.

[75] Bob Lantz, Brandon Heller, and Nick McKeown. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: ACM, 2010, 19:1.

[76] Networked Systems Group ETH Zürich. *P4-learning.* https://github.com/nsg-ethz/p4-learning. 2018.

[77] Laurent Vanbever. *Do you care about "gray" failures? Can we (network academics) help? A 10-min survey*. https://mailman.nanog.org/pipermail/nanog/2021-July/214217.html. 2021.

[78] Ralph Droms. *Dynamic Host Configuration Protocol*. RFC 2131. http://www.rfc-editor.org/rfc/rfc2131.txt. RFC Editor, 1997.

[79] Matteo Adriani and Maurizio Naldi. “Whose fault is it? correctly attributing outages in cloud services”. In: *2019 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE. 2019, 433.

[80] Jeffrey D. Case, Mark Fedor, Martin Lee Schoffstall, and James R. Davin. *Simple Network Management Protocol (SNMP)*. STD 15. http://www.rfc-editor.org/rfc/rfc1157.txt. RFC Editor, 1990.

[81] Cisco. *Cisco: Bug Search Tool*. https://bst.cisco.com/bugsearch/. 2022.

[82] Juniper Networks. *Problem Report Search*. https://prsearch.juniper.net/home. 2022.

[83] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. “Lossradar: Fast detection of lost packets in data center networks”. In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM. 2016, 481.

[84] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. “Chronos: Predictable Low Latency for Data Center Applications”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC ’12. San Jose, California: Association for Computing Machinery, 2012.

[85] Ramki Krishnan, Lucy Yong, Anoop Ghanwani, Ning So, and Bhumip Khasnabish. *Mechanisms for Optimizing Link Aggregation Group (LAG) and Equal-Cost Multipath (ECMP) Component Link Utilization in Networks*. RFC 7424. 2015.

[86] Marco Foschiano. *Cisco Systems UniDirectional Link Detection (UDLD) Protocol*. RFC 5171. 2008.

[87] Juniper Networks. *Network Management and Monitoring Guide: OAM Link Fault Management*. https://www.juniper.net/documentation/us/en/software/junos/network-mgmt/topics/topic-map/switches-interface-oam-lfm.html. (Accessed: 2023-07-01).

[88] Donnie Savage, James Ng, Steven Moore, Donald Slice, Peter Paluch, and Russ White. *Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP)*. RFC 7868. 2016.

[89] D. Katz and D. Ward. *Bidirectional Forwarding Detection*. RFC 5880. Internet Engineering Task Force, 2010.

[90] David Schweikertnet. *fping*. `https://github.com/schweikert/fping`. (Accessed: 2023-07-01).

[91] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. "Trumpet: Timely and Precise Triggers in Data Centers". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, 129.

[92] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. "Profiling Network Performance for Multi-Tier Data Center Applications". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, 57.

[93] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. "Passive Realtime Datacenter Fault Detection and Localization". In: *Nsdi* (2017), 25.

[94] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. "Taking the Blame Game out of Data Centers Operations with NetPoirot". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, 440.

[95] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. "007: Democratically Finding the Cause of Packet Drops". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, 419.

[96] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. "SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, 549.

[97] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. "Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, 421.

[98] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. "Netbouncer: Active device and link failure localization in data center networks". In: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, 599.

[99] François Aubry, David Lebrun, Stefano Vissicchio, Minh Thanh Khong, Yves Deville, and Olivier Bonaventure. "SCMon: Leveraging Segment Routing to Improve Network Monitoring". In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. San Francisco, CA, USA: IEEE Press, 2016, 1.

[100] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. "deTector: a Topology-aware Monitoring System for Data Center Networks". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, 55.

[101] David Eppstein, Michael T. Goodrich, Frank Uyeda, and George Varghese. "What's the Difference? Efficient Set Reconciliation without Prior Context". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, 218.

[102] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. "Planck: Millisecond-Scale Monitoring and Control for Commodity Networks". In: *SIGCOMM Comput. Commun. Rev.* 44.4 (2014), 407.

[103] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. "FlowRadar: A Better NetFlow for Data Centers". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, 311.

[104] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. "I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks". In: *11th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014) (2014), 71.

[105] Changhoon Kim and Parag Bhide and Ed Doe and Hugh Holbrook and Anoop Ghanwani and Dan Daly and Mukesh Hira and Bruce Davie. *In-band Network Telemetry (INT)*. P4 specification. 2016.

[106] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. "PINT: Probabilistic In-Band Network Telemetry". 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*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, 662.

[107] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. "One sketch to rule them all: Rethinking network flow monitoring with univmon". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, 101.

[108] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. "Sketch-based change detection: Methods, evaluation, and applications". In: *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. 2003, 234.

[109] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. "Data streaming algorithms for estimating entropy of network traffic". In: *ACM SIGMETRICS Performance Evaluation Review* 34.1 (2006), 145.

[110] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. "Scream: Sketch resource allocation for software-defined measurement". In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 2015, 1.

[111] Minlan Yu, Lavanya Jose, and Rui Miao. "Software Defined Traffic Measurement with OpenSketch". In: *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 2013, 29.

[112] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. "Qpipe: Quantiles sketch fully in the data plane". In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 2019, 285.

[113] Qun Huang, Patrick P. C. Lee, and Yungang Bao. “Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference”. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’18. Budapest, Hungary: Association for Computing Machinery, 2018, 576.

[114] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. “Elastic Sketch: Adaptive and Fast Network-Wide Measurements”. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’18. Budapest, Hungary: Association for Computing Machinery, 2018, 561.

[115] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. “Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM ’19. Beijing, China: Association for Computing Machinery, 2019, 334.

[116] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. “LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets”. In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 2021, 991.

[117] Kaicheng Yang, Yuhan Wu, Ruijie Miao, Tong Yang, Zirui Liu, Zicang Xu, Rui Qiu, Yikai Zhao, Hanglong Lv, Zhigang Ji, and Gaogang Xie. “ChameleMon: Shifting Measurement Attention as Network State Changes”. In: *Proceedings of the ACM SIGCOMM 2023 Conference*. ACM SIGCOMM ’23. New York, NY, USA: Association for Computing Machinery, 2023, 881.

[118] Hun Namkung, Daehyeok Kim, Zaoxing Liu, Vyas SekaR, and Peter Steenkiste. “Telemetry Retrieval Inaccuracy in Programmable Switches: Analysis and Recommendations”. In: *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. New York, NY, USA: Association for Computing Machinery, 2021, 176.

[119] Mimi Qian, Lin Cui, Xiaoquan Zhang, Fung Po Tso, and Yuhui Deng. “dDrops: Detecting silent packet drops on programmable data plane”. In: *Computer Networks* 214 (2022), 109171.

[120] *Trace Statistics for CAIDA Passive OC48 and OC192 Traces*. https://www.caida.org/catalog/datasets/trace_stats/. 2023.

[121] *Intel Tofino 3 Brief*. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html.

[122] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. "Efficient Measurement on Programmable Switches Using Probabilistic Recirculation". In: *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*. IEEE Computer Society, 2018, 313.

[123] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. "Elastic Switch Programming with P4All". In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. HotNets '20. Virtual Event, USA: Association for Computing Machinery, 2020, 168.

[124] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. "D2R: Policy-Compliant Fast Reroute". In: *SOSR*. ACM, 2021, 148.

[125] Stephane Litkowski, Ahmed Bashandy, Clarence Filsfils, Pierre Francois, Bruno Decraene, and Daniel Voyer. *Topology Independent Fast Reroute using Segment Routing*. Internet-Draft draft-ietf-rtgwg-segment-routing-ti-lfa-08. Work in Progress. Internet Engineering Task Force, 2022.

[126] Zhizhen Zhong, Manya Ghobadi, Alaa Khaddaj, Jonathan Leach, Yiting Xia, and Ying Zhang. "ARROW: Restoration-Aware Traffic Engineering". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM '21. 2021, 560.

[127] *Juniper Bug: PR1434567 – IPv6 neighbor solicitation packets getting dropped on PTX. (Open Registration Required)*. https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1434567.

[128] *Juniper Bug: PR1398407 – On SRX4600 and SRX5000 line of devices, BGP packets might be dropped under high CPU usage.. (Open Registration Required)*. https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1398407.

[129] *Cisco Bug: CSCea91692 - PSA has a corrupted cef entry, affecting IP:IP traffic*. https://quickview.cloudapps.cisco.com/quickview/bug/CSCea91692.

[130] *Cisco Bug: CSCti14290 - VPN Aggregate Label dmac corruption in hardware forwarding entry*. https://quickview.cloudapps.cisco.com/quickview/bug/CSCti14290.

[131] *Cisco Bug: CSCtc33158 - 7600-ES+40G3CXL drops random sized L2TPv3 packets with cookies enabled*. https://quickview.cloudapps.cisco.com/quickview/bug/CSCtc33158.

[132] *Cisco Bug: CSCuv31196 - Random MPLS Packet Drops With IP Multicast Over L3 Ring on ASR901*. https://quickview.cloudapps.cisco.com/quickview/bug/CSCuv31196.

[133] *Juniper Bug: PR1296089 – Traffic received from core are not sent to locally attached circuit due to QSN timeout*. https://www.juniper.net/documentation/en_US/junos/information-products/topic-collections/release-notes/18.1/jd0e17997.html.

[134] *Juniper Bug: PR1450545 – Traffic loss might occur when there are around 80,000 routes in FIB (Open Registration Required)*. https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1450545.

[135] *Juniper Bug: PR1313977 – Traffic drop occurs on sending traffic over "et" interfaces due to CRC errors*. https://www.juniper.net/documentation/en_US/junos/information-products/topic-collections/release-notes/17.4/jd0e19328.html.

[136] *Juniper Bug: PR1309613 – Traffic loss may be seen if sending traffic via the 40G interface*. https://www.juniper.net/documentation/en_US/junos/information-products/topic-collections/release-notes/17.4/jd0e19328.html.

[137] *Juniper Bug: PR1459698 – Silent dropping of traffic upon interface flapping after DRD auto-recovery (Open Registration Required)*. https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1459698.

[138] *Juniper Bug: PR1441816 – Egress stream flush failure and traffic blackhole might occur (Open Registration Required)*. https://prsearch.juniper.net/InfoCenter/index?page=prcontent&id=PR1441816.

[139] *Network Simulator 3*. https://www.nsnam.org/. 2018.

[140] Ethernet Alliance. *2023 Ethernet Roadmap*. https://ethernetalliance.org/technology/ethernet-roadmap/. Accessed: 2023-11-07. 2023.

[141] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. "Leveraging Zipf's law for traffic offloading". In: *ACM SIGCOMM Computer Communication Review* 42.1 (2012), 16.

[142] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. "Theory and Practice of Bloom Filters for Distributed Systems". In: *IEEE Communications Surveys & Tutorials* 14.1 (2012), 131.

[143] Andrei Broder and Michael Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In: *Internet Mathematics*. Vol. 1. 2002, 636.

[144] CAIDA. *The CAIDA UCSD Anonymized 2013/2014/2015/2016/2018 Internet Traces*. `http://www.caida.org/data/passive/passive_2013_dataset.xml`.

[145] *Summary of Anonymization Best Practice Techniques*. `hhttps://www.caida.org/projects/predict/anonymization/`.

[146] *Visibility of IPv4 and IPv6 Prefix Lengths in 2019*. `https://labs.ripe.net/Members/stephen_strowes/visibility-of-prefix-lengths-in-ipv4-and-ipv6`.

[147] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. "DRMT: Disaggregated Programmable Switching". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, 1.

[148] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. "Infinite cacheflow in software-defined networks". In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, 175.

[149] D. Katz and D. Ward. *Bidirectional Forwarding Detection*. RFC 5880. Internet Engineering Task Force, 2010.

[150] Clarence Filsfils, Pradosh Mohapatra, John Bettink, Pranav Dharwadkar, Peter De Vriendt, Yuri Tsier, Virginie Van Den Schrieck, Olivier Bonaventure, and Pierre Francois. *BGP Prefix Independent Convergence (PIC) Technical Report*. Tech. rep. `http://www.cisco.com/en/US/prod/collateral/routers/ps5763/bgp_pic_technical_report.pdf`. Cisco, 2011.

[151] Aristidis Lambrianidis and Eric Nguyen Dyu. "Route Server Implementations Performance". Euro-IX Forum, Amsterdam, The Netherlands. 2012.

[152] Allen Taylor, Benedikt Rudolph, Daniel Spierling, and Johannes Moos. "An IXP Route Server Test Framework". Euro-IX Forum, Barcelona, Spain. 2017.

[153] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. "SWIFT: Predictive fast reroute". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM. 2017, 460.

[154] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271 (Draft Standard). 2006.

[155] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. "CONGA: Distributed Congestion-aware Load Balancing for Datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, 503.

[156] *Barefoot Tofino*. `https://barefootnetworks.com/products/product-brief-tofino/`.

[157] *The P4_16 Language Specification - Version 1.0.0*. `https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html`.

[158] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. "In-Network Computation is a Dumb Idea Whose Time Has Come". In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. HotNets-XVI. Palo Alto, CA, USA: ACM, 2017, 150.

[159] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. "Language-directed hardware design for network performance monitoring". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM. 2017, 85.

[160] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. "Heavy-hitter detection entirely in the data plane". In: *Proceedings of the Symposium on SDN Research*. ACM. 2017, 164.

[161] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Mohammad Alizadeh, David Walker, Jennifer Rexford, Vimalkumar Jeyakumar, and Changhoon Kim. “Hardware-software co-design for network performance measurement”. In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM. 2016, 190.

[162] Mina Tahmasbi Arashloo, Monia Ghobadi, Jennifer Rexford, and David Walker. “HotCocoa: Hardware Congestion Control Abstractions”. In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. 2017, 108.

[163] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. “NetChain: Scale-Free Sub-RTT Coordination”. In: *15th USENIX Symposium on Networked Systems Design and Implementation*. 2018.

[164] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. “Paxos made switch-y”. In: *ACM SIGCOMM Computer Communication Review* 46.2 (2016), 18.

[165] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. “Netpaxos: Consensus at network speed”. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 2015, 5.

[166] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. “Ensuring Connectivity via Data Plane Mechanisms”. In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, 113.

[167] *Cisco IOS. IP Routing: BFD Configuration Guide.* `https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bfd/configuration/15-mt/irb-15-mt-book/irb-bi-fwd-det.html`.

[168] Timothy G. Griffin and Gordon Wilfong. “An Analysis of BGP Convergence Properties”. In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’99. Cambridge, Massachusetts, USA: ACM, 1999, 277.

[169] Joao Luis Sobrinho. “Network routing with path vector protocols: Theory and applications”. In: *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM. 2003, 49.

[170] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. “In-Network Computation is a Dumb Idea Whose Time Has Come”. In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. HotNets-XVI. Palo Alto, CA, USA: ACM, 2017, 150.

[171] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. “DRILL: Micro Load Balancing for Low-latency Data Center Networks”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: ACM, 2017, 225.

[172] *TCP Behavior of BGP*. `https://archive.psg.com/121009.nag-bgp-tcp.pdf`. 2012.

[173] Stephen Edwards, Luciano Lavagno, Edward A Lee, and Alberto Sangiovanni-Vincentelli. “Design of embedded systems: Formal models, validation, and synthesis”. In: *Proceedings of the IEEE* 85.3 (1997), 366.

[174] Péter Arató, Zoltán Ádám Mann, and András Orbán. “Algorithmic aspects of hardware/software partitioning”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 10.1 (2005), 136.

[175] Jurij Nota. “Enhancing Data Plane Signals for Network Monitoring Systems”. Master’s thesis. ETH Zürich, 2022.

[176] Saeed Masoudnia and Reza Ebrahimpour. “Mixture of experts: a literature survey”. In: *Artificial Intelligence Review* 42.2 (2014), 275.