FEDERAL UNIVERSITY OF PAMPA

Ariel Góes De Castro

Towards Probe Planning for In-band Network Telemetry

Alegrete
2023

Ariel Góes De Castro

Towards Probe Planning for In-band Network Telemetry

Qualification submitted to the Graduate Program in Software Engineering of Federal University of Pampa in partial fulfillment of the requirements for the Master's degree in Software Engineering.

Supervisor: Prof. Dr. Marcelo Caggiani Luizelli

Alegrete
2023

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

G355t Góes de Castro, Ariel
Towards Probe Planning for In-band Network Telemetry / Ariel Góes de Castro.
69 p.

Tese(Doutorado)-- Universidade Federal do Pampa, MESTRADO EM ENGENHARIA DE SOFTWARE, 2023.
"Orientação: Marcelo Luizelli".

1. In-band Network Telemetry (INT). 2. Software-Defined Network(SDN). 3. Probe. 4. Network Monitoring. 5. Fast Reroute (FRR). I. Título.

ARIEL GÓES DE CASTRO

TOWARDS PROBE PLANNING FOR IN-BAND NETWORK TELEMETRY

Dissertação apresentada ao Programa de Pós-Gradução em Engenharia de Software da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Mestre em Engenharia de Software.

Dissertação defendida e aprovada em: 20 de julho de 2023.

Banca examinadora:

_____

Prof. Dr. Marcelo Caggiani Luizelli

Orientador

UNIPAMPA

_____

Prof. Dr. Fábio Diniz Rossi

IFFAR

_____

Prof. Dr. Roberto Irajá Tavares da Costa Filho

IFSul

Assinado eletronicamente por **MARCELO CAGGIANI LUIZELLI**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 20/07/2023, às 16:13, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Fábio Diniz Rossi**, **Usuário Externo**, em 20/07/2023, às 16:14, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Roberto Irajá Tavares da Costa Filho**, **Usuário Externo**, em 20/07/2023, às 16:15, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1191873** e o código CRC **9FFE9920**.

This work is dedicated to my family, friends, and everyone else who helped me somehow to achieve this moment in my life.

ACKNOWLEDGEMENTS

I first would like to thank my family. They have always supported me and provided all kinds of support to make me happy, without measuring efforts. Certainly, none of this would have been possible without their help. I am forever grateful to Dr. Marcelo Caggiani Luizelli (friend and supervisor) for having offered me all the motivation and tools since the moment I started to conduct research. I would also like to thank Dr. Fábio Diniz Rossi for his sincere friendship and constant support expressed in different ways. Both were very close to me along these recent years and I believe that both are examples of people and professionals to be followed

“I don’t play the odds. I play the man.” (Harvey Specter - Suits)

RESUMO

O Monitoramento de Rede em Banda (INT, do inglês In-band Network Telemetry) tem se destacado como uma abordagem poderosa para monitorar redes programáveis, fornecendo uma visibilidade detalhada dos eventos na rede. No entanto, as abordagens existentes para a orquestração do INT frequentemente negligenciam a tolerância a falhas no plano de dados, deixando os mecanismos de monitoramento comprometidos durante falhas na rede. Para solucionar essa lacuna, propomos o `InPatching`, uma abordagem de tolerância a falhas no plano de dados para o monitoramento baseado em INT. O `InPatching` detecta autonomamente dispositivos com falhas e aplica desvios coordenados nos ciclos de sondagem afetados, garantindo a coleta ininterrupta de dados de telemetria sem depender do plano de controle. Ao transferir a recuperação para o plano de dados, o `InPatching` reduz significativamente o tempo de recuperação em comparação com as estratégias do plano de controle. Para viabilizar desvios eficientes, formalizamos o planejamento de sondagem tolerante a falhas para INT usando um modelo de Programação Linear de Inteiros Mistas (MILP). Esse modelo nos permite determinar de forma eficiente os caminhos ótimos de desvio e minimizar o impacto no desempenho da rede. Nossa extensa avaliação demonstra a eficácia do `InPatching` em comparação com as soluções do plano de controle. Mostramos que o `InPatching` supera as abordagens do plano de controle em um fator de 18X, proporcionando recuperação rápida e confiável para o monitoramento baseado em INT, evitando um impacto substancial no desempenho. A compilação do código em hardware também foi efetuada com sucesso e as métricas obtidas sobre o uso de recursos – i.e., Match-Action UNIT (MAU) e Tagalong collections – indicam um baixo uso de recursos de memória, em média, para alocação do componentes código reescrito na arquitetura Tofino™ Native Architecture (TNA). Além das contribuições técnicas, também disponibilizamos artefatos de software de código aberto que facilitam a adoção e a reprodutibilidade do `InPatching`. Os operadores de rede podem aproveitar essa solução para manter uma visibilidade abrangente da rede, mesmo durante falhas na rede, garantindo uma cobertura contínua e atualizada dos dados de INT coletados. No geral, nosso trabalho contribui para o avanço do monitoramento de rede tolerante a falhas e destaca a importância de considerar a resiliência do plano de dados no projeto das abordagens de orquestração do INT. Ao abordar esse aspecto crítico, o `InPatching` aprimora significativamente a confiabilidade e a eficácia de sistemas de monitoramento baseados em INT em redes programáveis.

Palavras-chave: Telemetria *In-Band*, *Software-Defined Network(SDN)*, *Probes*, Monitoramento de Rede, Fast Reroute (FRR)

ABSTRACT

In-Band Network Telemetry (INT) has emerged as a powerful network monitoring approach in programmable networks, providing fine-grained visibility into network events. However, existing INT orchestration approaches often overlook fault tolerance in the data plane, leaving monitoring mechanisms compromised during network failures. To address this gap, we propose `InPatching`, an in-network fault-tolerant approach for INT-based monitoring. `InPatching` autonomously detects faulty devices and applies coordinated detours in affected probing cycles, ensuring uninterrupted telemetry data collection without relying on the control plane. By offloading recovery to the data plane, `InPatching` significantly reduces the recovery time compared to control plane strategies. To enable efficient detours, we formalize fault-tolerant probing planning for INT using a Mixed-Integer Linear Programming (MILP) model. This model allows us to efficiently determine the optimal detour paths and minimize the impact on network performance. Our extensive evaluation demonstrates the effectiveness of `InPatching` in comparison to control plane solutions. We show that `InPatching` outperforms control plane approaches by a factor of 18X, providing fast and reliable recovery for INT-based monitoring while avoiding substantial overhead. The compilation of the code into hardware has also been successfully performed, and the metrics obtained regarding resource usage – i.e., MAU and Tagalong collections –indicate low memory resource utilization, on average, for allocating the rewritten code components in the TNA architecture. In addition to the technical contributions, we also release open-source software artifacts that facilitate the adoption and reproducibility of `InPatching`. Network operators can leverage this solution to maintain network-wide visibility even during network failures, ensuring continuous coverage and freshness of collected INT data. Overall, our work contributes to the advancement of fault-tolerant network monitoring and highlights the importance of considering data plane resilience in the design of INT orchestration approaches. By addressing this critical aspect, `InPatching` significantly enhances the reliability and effectiveness of INT-based monitoring systems in programmable networks.

Key-words: In-band Network Telemetry (INT), Software-Defined Network(SDN), Probe, Network Monitoring, Fast Reroute (FRR)

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

DCN Data Center Network

DDoS Distributed Denial of Service

DFS Depth-First Search

ETSI European Telecommunications Standards Institute

EWMA Exponentially Weighted Moving Average

FIFO First In First Out

FPGA Field Programmable Gate Array

FRR Fast Rerouting

ILP Inter Linear Programming

INT In-Band Network Telemetry

INT-MD INT eMbed Data

INT-MX INT eMbed instruct(X)ions

INT-XD INT eXport Data

INTO In-Band Network Telemetry Orchestration

INTOPP In-band Network Telemetry Orchestration Plan Problem

MAU Match-Action UNIT

MILP Mixed-Integer Linear Programming

MTU Maximum Transmission Unit

NFV Network Function Virtualization

NPU Neural Processing Unit

P4 Programming Protocol-independent Packet Processors

PCAP Packet Capture

PHV Packet Header Vector

POF Protocol Oblivious Forwarding

Rx Receiver

SALU Static Arithmetic Logical Unit

SDK Software Development Kit

SDN Software-Defined Network

SGT Select Group Table

SLA Service Level Agreement

SmartNIC Smart Network Interface Card

SNMP Simple Network Management Protocol

SRAM Static Random-Access Memory

TNA Tofino™ Native Architecture

URLLC Ultra-Reliable and Low-Latency Communications

VLIW Very Long Instruction Word

VoIP Voice Over Ip

CONTENTS

1 INTRODUCTION

This chapter discusses the problem of orchestrating probes for INT. First, we briefly introduce INT networks, followed by the problem definition and constraints. Then, we formally define the problem and our contributions to this research.

1.1 Context and Motivation

No matter how fast networking research evolves, challenges always seem to be unsolved or, at least, partially solved. One of the culprits to existing limitations is the growing demand for services with increasingly stringent restrictions – e.g., Ultra-Reliable and Low-Latency Communications (URLLC) operation mode in 5G networks – that push existing network architectures to the limit. Besides that, existing mechanisms limit the operator's ability to express their intentions about network behavior and are prone to human error (e.g., device misconfiguration). With that in mind, recent efforts from both the academia (FEAMSTER; REXFORD, 2017) and industry (Juniper Networks, 2017; Huawei, 2019) have tried to provide both automatic (i.e., independent of human instructions) and autonomous (i.e., capable of making its own decisions) networks into the concept of self-driving networks. In short, self-driving networks are autonomous networks that act according to high-level intent from their users while automatically adapting to network changes in the traffic (e.g., device/link failures) and user behavior (e.g., video streaming, Voice Over Ip (VoIP)). AI/ML-assisted methods verify and make decisions autonomously to ensure that operators' input intents are satisfied over time. But, for that, collecting information on the network is a crucial phase of the process, ensuring that the data is provided promptly and with a particular frequency to the algorithms - considering that there are no communication failures along its route.

INT is an emerging network monitoring approach in programmable networks (PAN et al., 2019) that allows increasing network visibility of fine-grained network events (e.g., micro-bursts (CHEN et al., 2018) and network load imbalance (TAMMANA; AGARWAL; LEE, 2018))). The INT concept has been fostered by the recent adoption of programmable data planes and domain-specific languages such as Protocol Oblivious Forwarding (POF) and Programming Protocol-independent Packet Processors (P4). More specifically, P4 provides a detailed data plane specification (The P4.org Applications Working Group, 2020) on how INT operates. In short, INT works by continuously collecting low-level data plane statistics (a.k.a. telemetry data) from the infrastructure in a per-packet manner. These telemetry data include internal data plane statistics such as queue occupancy, per-packet processing time, and aggregated/computed statistics such as inter-packet gap (SINGH et al., 2020).

In the classical INT operation – also known as INT eMbed Data (INT-MD) – network packets are instructed to properly collect telemetry data as they are routed through the network. The instructions are added into an INT packet – i.e., a packet carrying an

INT header – , which can be embedded into active network flows (HOHEMBERGER et al., 2019) or specially-crafted probing packets (CASTRO et al., 2021). These packets are then interpreted by INT-enabled forwarding devices, which collect required telemetry data. Figure 1 illustrates the whole INT procedure using probing packets. Observe there are three probing cycles collecting data from the network – for instance, probing cycle f_1 collects telemetry data from nodes A, E, F, G, H, and I, returning to origin A (i.e., steps (1) – (5)), and then to an INT collector.

Recent research (PAN et al., 2019; HOHEMBERGER et al., 2019; LIU et al., 2018; MARQUES et al., 2019) have made consistent efforts regarding the In-Band Network Telemetry Orchestration (INTO). The problem consists of efficiently using available resources (in this case, spare space on network packets) to collect data plane network statistics. In this context, Liu et al. (LIU et al., 2018) and Pan et al. (PAN et al., 2019) have focused on optimizing the usage of probing packets to collect INT data. At the same time, Marques et al. (MARQUES et al., 2019) and Hohemberger et al. (HOHEMBERGER et al., 2019) have focused on embedding telemetry data into production network packets. In turn, Castro et al. (CASTRO et al., 2020) leverages a shortest-path algorithm to reconstruct probe paths from link failures in the control plane.

1.2 Research Problem

Despite the efforts toward INTO, little has been done to provide fault-tolerant mechanisms for INT in the data plane. In case a network link fails, all of the INT monitoring mechanism that relies on that device is compromised. In Figure 1, for example, the failure of network link G–H directly affects probing cycles f_1 and f_2 (step 6). A naive solution to provide fault-tolerance to this problem consists of computing a novel solution or adapting existing ones upon a failure (e.g., (PAN et al., 2019) (CASTRO et al., 2020)). In this case, a control plane application (step (7)) would be triggered to compute a new telemetry solution. In Figure 1, for instance, the new solution comprises a detour of probing cycles f_1 and f_2 through an alternative/updated path (step (8)). Despite this solution, the recovery of the INT monitoring approach would take a few hundred milliseconds in the best case. This limitation is mainly due to the time required to identify the fault, the time spent to react (i.e., compute a new solution), and recovery (i.e., update the data plane). Consequently, the network-wide visibility required by monitoring applications might degrade in terms of coverage and freshness (MARQUES et al., 2019) during the faulty period.

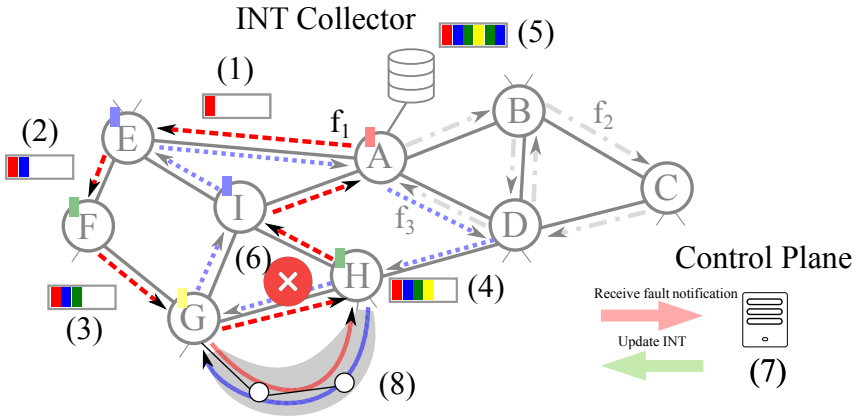

Figure 1 – Overview of INT planning.

1.3 Goals and Contributions

To fill this gap, in this work, we propose `InPatching`: an in-network approach to fast recovery INT-based monitoring approaches. In the event of faulty devices, `InPatching` autonomously (and without the control plane intervention) fix monitoring cycles by identifying the defective device and applying detours in affected probing cycles to ensure the required INT data is collected correctly. `InPatching` is mainly offloaded to the data plane, and, therefore, the recovery time of INT-based monitoring mechanisms can be faster than existing control plane strategies. To provide efficient detours to existing probing cycles, we formalize the fault-tolerant probing planning for INT by extending the existing orchestration model (CASTRO et al., 2021). Results show that *InPatching* outperforms control plane solutions by reducing the communication delay by up to 18X. The main contributions of this work can be summarized as follows:

- the proposal of an in-network strategy to quickly react to faulty network conditions;
- P4 code offloading to the TNA architecture;
- the formalization of the fault-tolerant probing planning for INT;
- an open-source code to foster reproducibility.

1.4 Outline

The remainder of this work is organized as follows. In Chapter 2, we discuss related work in the area of in-band network telemetry with a focus on the programmability of the data plan and its benefits. In Chapter 3, we introduce the `InPatching` design in programmable data planes. Also, in Chapter 3, we present and discuss the results of evaluating the proposed approach. Last, in Chapter 4, we conclude the work with final remarks and perspectives for future work.

2 BACKGROUND AND RELATED WORK

In this chapter, we start by exploring the idea of autonomous networks and the convergence of ideas that led us to the current notion of self-driving networks. Following, we review data plane programmability and the benefits of offloading a range of tasks to the data plane. Then, we review INTO. Finally, we review the most recent fast-rerouting mechanisms.

2.1 Autonomous networks

Self-driving networks are an increasingly closer reality. Among its primary requirements are: (i) the ability to interpret and validate user intentions in high-level languages; (ii) adapt to changes in the network – e.g., new devices, changing requirements, and network conditions – over time while maintaining compliance with user intents without its intervention. The first notion of networks that "run themselves" was introduced by Horn (HORN, 2001) where the self-* properties were presented (i.e., self-star) – self-awareness, self-protecting, self-optimizing, self-healing, and self-configuring. Similarly, Clark et al. (CLARK et al., 2003) suggested building self-healing networks without external intervention and proposed a Knowledge Plane relying on AI techniques to maintain network visibility. However, at the time, such practices needed to be more extensive, and a practical implementation did not take off. Later, the self-* properties were incorporated by Jacob et al. (JACOB et al., 2004) in the MAPE-K loop model – an IBM automation toolkit. FOCALE (STRASSNER; AGOULMINE; LEHTIHET, 2006) is among the first to propose a self-managing network. A breakthrough in this work was the introduction of a Policy Manager responsible for translating natural language (e.g., English) into vendor-specific device requirements. Similarly, the Autonomic Network Architecture (ANA) (BOUABENE et al., 2009) introduced a system-level abstraction and provided an Application Programming Interface (API) to manipulate the network elements.

The approaches above are limited to simplified AI models. It is because the existing collection methods at the time (e.g., Simple Network Management Protocol (SNMP)) provided low visibility over the network state and a low collection frequency, making detecting certain network anomalies (e.g., micro-bursts) unfeasible. For example, both MAPE-K loop (JACOB et al., 2004) and FOCALE (STRASSNER; AGOULMINE; LEHTIHET, 2006) had learning components. However, these components are based on storing temporal events and not in standalone decision models based on statistics – like those presented in ML. Recent efforts from academia (FEAMSTER; REXFORD, 2017) and industry (Juniper Networks, 2017) have tried to standardize self-driving networks shortly. To consolidate previous ideas of self-managing networks, European Telecommunications Standards Institute (ETSI) members released a white paper (ETSI, 2020) with the main challenges and roles to be played by new network architectures. Although not mandatory, the document mentions the use of current Software-Defined Network (SDN)-based network

technologies (e.g., Network Function Virtualization (NFV) (JACOBS et al., 2017)) to and "close" the loop of self-driving networks – i.e., no/minimal human interference in the network management process.

2.2 Data plane programmability

There are many examples of applications being offloaded to the data plane, such as Service Level Agreement (SLA) verification (MARQUES; LEVCHENKO; GASPARY, 2020), load balancing (HSU et al., 2020b), gray-failures (MOLERO; VISSICCHIO; VANBEVER, 2022; JIA et al., 2020) or even Distributed Denial of Service (DDoS) detection (LAPOLLI; MARQUES; GASPARY, 2019) that provide more fine-grained and low-latency solutions for a scenario of an increasing number of services (e.g., video streaming (YAMANSAVASCILAR et al., 2020)). These solutions allow several benefits as (i) CPU workload reduction on servers and (ii) less power and capital expenses while processing a high amount of traffic. However, to properly detect network anomalies and manage network behavior, there must be a way to collect metrics/statistics to provide more precise insights for the network operator. More specifically, future approaches need to consider the needs of applications (e.g., latency) promptly and with minimal human intervention — i.e., minimal control plane intervention. Traditionally speaking, network management tools are either based on polling (CASE M. FEDOR, 1989) or sampling (e.g., NetFlow (CLAISE et al., 2004), SFlow (PHAAL; PANCHEN; MCKEE, 2001)). These approaches incur (i) narrow network coverage since few telemetry devices are used, anomalies/events may evade the supervision of network operators, and; (ii) low scalability because a high sampling frequency may degrade network resources (e.g., link bandwidth). With that in mind, network programming languages such as P4 (BOSSHART et al., 2014) and POF (SONG, 2013) allow network operators to specify the internal pipeline of forwarding devices.

In contrast to traditional networks, P4 allows the customization of parsers, define headers - hence protocols - and the packet processing logic with match-action tables. Initially, it was designed for software/hardware switch programming. Still, now it is available to various devices/targets (e.g., Smart Network Interface Card (SmartNIC)s, network appliances, Application-Specific Integrated Circuit (ASIC)s, Neural Processing Unit (NPU)s, and Field Programmable Gate Array (FPGA)s). Figure 2 summarizes the abstract forwarding model. First, an incoming packet is processed by a programmable parser. If the packet is in an allowed format (i.e., correct header fields), it is forwarded to a set of user-defined match-action tables at the ingress pipeline. Then, these modifications are stored in a buffer and copied to the egress pipeline - where another group of user-defined tables again processes the packet. Finally, the packet is reconstructed by a deparser and emitted to an egress port. With this flexibility, we can collect per-packet data collection granularity. In recent years, INT (The P4.org Applications Working Group, 2020) is becoming the de-facto representative of network telemetry. Since its conception in 2015 at

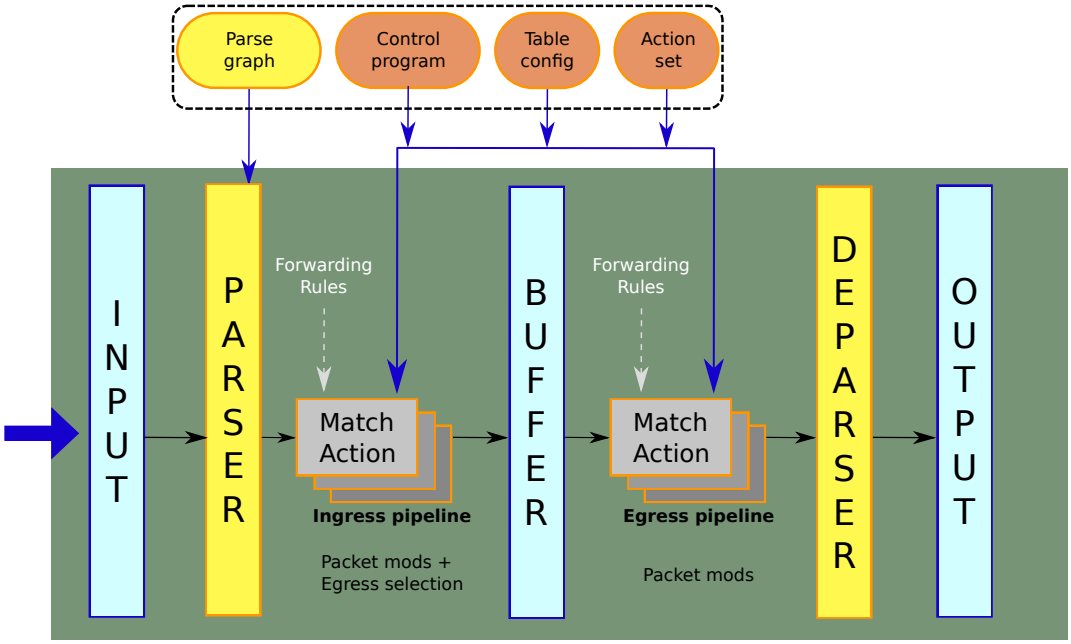

Figure 2 – P4 abstract forwarding model.

P4.org, INT has suffered several changes summarized into three distinct operation modes (see Figure 3).

INT eXport Data (INT-XD). In this mode, packet header modification is not allowed, and it is known by the use of "postcards", where a set of metadata is directly exported from the data plane based on Flow Watchlists.

INT eMbed instruct(X)ions (INT-MX). In contrast, INT-MX allows embedding per-packet instructions. In summary, the INT Source (the first node) embeds instructions, then all INT nodes in the telemetry path send telemetry data to an external collector.

INT-MD. Similarly to INT-MX, this mode allows embedding instructions to packet header fields. However, metadata may also be included, and only the last INT-enabled node (INT sink) in the telemetry path is responsible for exporting data to a collector.

Regardless of the monitoring mode chosen, there must be a way to coordinate the collection of metrics in the data plane, mitigating the use of network resources while taking into account variables such as (i) the frequency of information collection, where a very systematic collection can overwhelm buffer queues and available memory on the switch; (ii) device coverage and metrics collected, to support greater device visibility and anomaly location; (iii) the optimization of the goals above to reduce the cost of operation. Given this importance, INTO (MARQUES et al., 2019) problems are concerned with solutions that seek to coordinately minimize the activity of telemetry flows and the saturation probability of network resources (e.g., CPU memory, link bandwidth) while maintaining the network visibility.

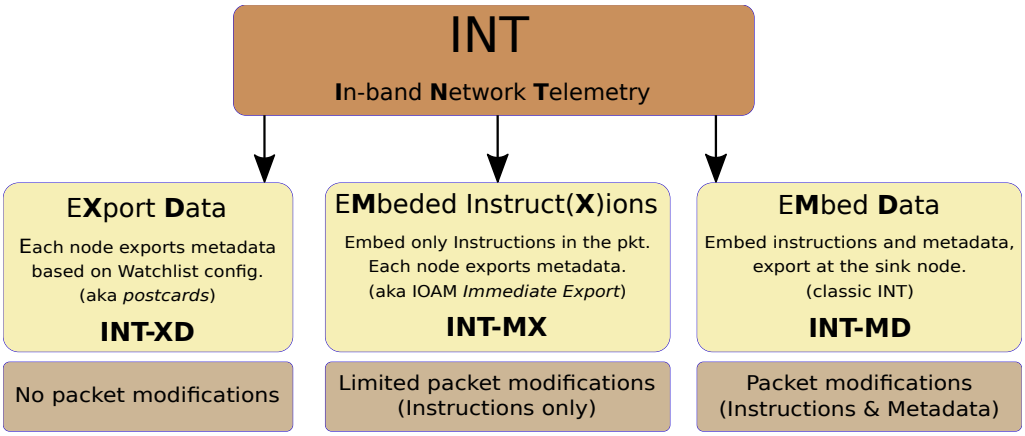

Figure 3 – INT operation modes.

2.3 Tofino™ Native Architecture

The concept of TNA was developed by Barefoot Networks, a company acquired by Intel in 2019, to revolutionize network design. Barefoot Networks introduced the Tofino™ family of network processing units, which followed a unique design philosophy. Unlike traditional Application-Specific Integrated Circuits (ASICs), the Tofino™ chip focused on hardware programmability, allowing software to dictate the device's behavior and enabling customization of the network processing pipeline. The core elements that unlocked the potential of the Tofino™ chip were the Barefoot Network Tofino™ Software Development Kit (SDK) and the P4 language (INTEL, 2021).

The Tofino™ chip was engineered to integrate with P4 and the P4 Runtime interface seamlessly. These technologies empower network operators to dynamically configure and control the forwarding plane of the device. By leveraging this capability, operators can tailor the network's behavior to the specific needs of their applications, resulting in improved performance and reduced latency.

Within the Tofino™ chip, the traffic manager assumes a crucial role in effectively managing network traffic flow. Its responsibilities include packet scheduling and queue management within the switch. The switch architecture comprises ingress pipelines that parse packets and extract metadata. In contrast, the egress pipeline utilizes the results of the ingress processing to determine the next destination for each packet.

A noteworthy advantage of the TNA approach is its ability to update the device's

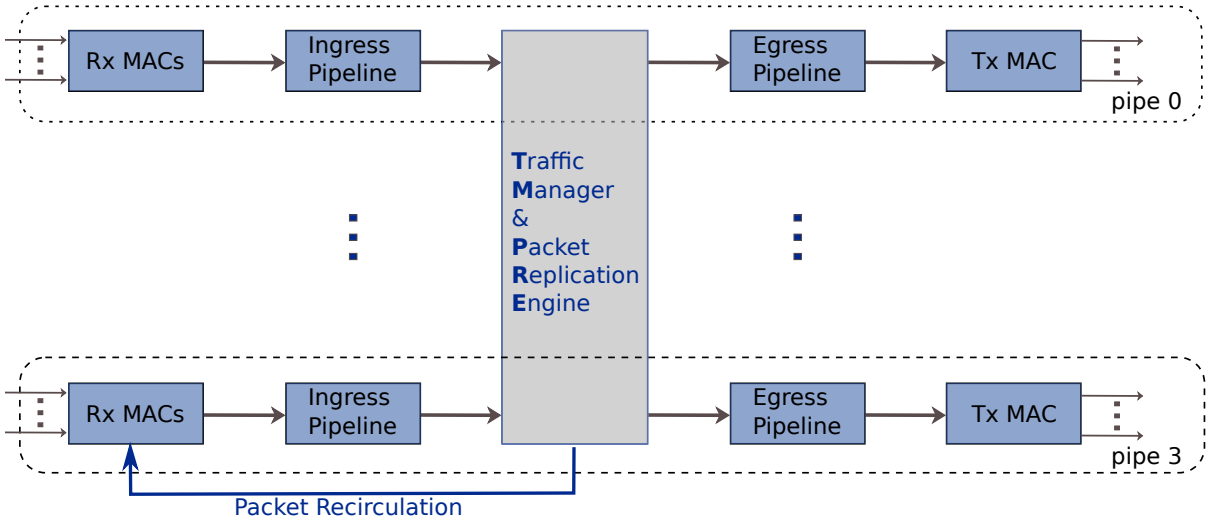

Figure 4 – Simplified TNA block diagram.

software without requiring modifications to the underlying hardware. This feature simplifies adding new features and functionalities to the network, reducing deployment time and cost. Figure 4 summarizes the organization of the architecture. First, the packet is received by a Receiver (Rx) port. Then, in the ingress pipeline, the packet undergoes a parsing process, extracting and verifying user-defined headers. In the ingress pipeline, the defined match+action tables are applied according to the user's logic and forwarded to the traffic manager. At this point, the packet can either be recirculated directly to an Rx port in the ingress again or undergo a deparser process and be forwarded to an egress pipeline, where the process repeats. Then, the packet is transmitted to the egress port. While the packet is traversing the ingress and egress pipeline, a set of Packet Header Vector (PHV)s are responsible for handling/storing data such as packet header fields and conditional operators (see Figure 5). More specifically, they carry information starting from the parser, then through MAU (a.k.a. "stages") to the deparser. Also, all the communication between different blocks happens via PHV. The stage used by PHVs may be shared and assigned to either the ingress or pipeline (never both), but both header fields and metadata may be packed into any container or combination. Despite this flexibility in allocating information in PHVs, the compiler is responsible for determining in which of the 12 stages – i.e., for each pipeline – the information should be allocated. Moreover, Tofino™ is designed to be fully programmable, allowing for seamless updates and incorporating new features and functionalities without any hardware changes. Ultimately, TNA, in combination with P4, empowers network operators to exert precise control over their networks, enabling them to customize the network to meet the specific requirements of their applications.

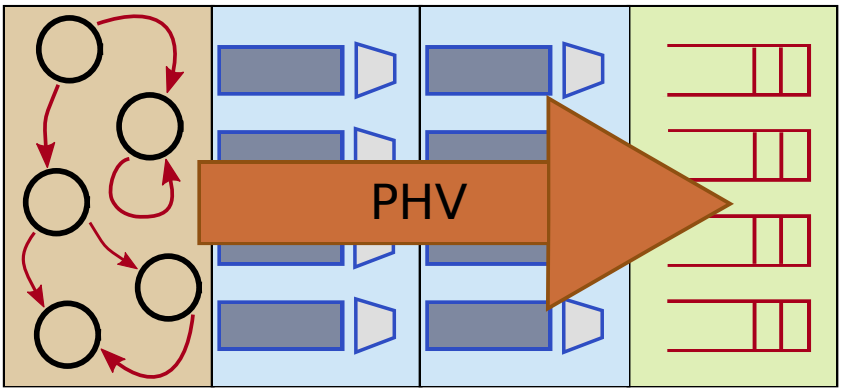

Figure 5 – PHV carries information through the TNA pipeline.

2.4 In-Band Network Orchestration

Existing INTO approaches rely either on using (i) active flows (HOHEMBERGER et al., 2019; MARQUES et al., 2019; BASAT et al., 2020; SCANO et al., 2021) or (ii) probe flows (RAMANATHAN; KANZA; KRISHNAMURTHY, 2018; LIU et al., 2018; PAN et al., 2019; BHAMARE et al., 2019; GENG et al., 2019; LIN et al., 2020; YUAN et al., 2022) to collect telemetry demands across the network topology. Marques et al. (MARQUES et al., 2019) propose two heuristic strategies for collecting telemetry data, namely, *concentrate* and *balance*. The first strategy strives to aggregate telemetry data on a restricted number of flows. In contrast, the second tries to distribute equally the telemetry data over a more comprehensive range of available network flows. Hohemberger et al. (HOHEMBERGER et al., 2019) is the first attempt to collect telemetry items in real-time coordinately. It solved the In-band Network Telemetry Orchestration Plan Problem (INTOPP) by designing a machine-learning-based model and formalizing the collection problem. It must satisfy both spatial and temporal requirements, i.e., the model considers the probes must collect items from specific devices and, simultaneously, at a certain rate to properly feed machine-learning applications on top of the network to detect anomalies (e.g., DDoS). Similarly, SDProber (RAMANATHAN; KANZA; KRISHNAMURTHY, 2018) performs a *random-walk* approach for embedding INT data into probe packets and increases the probe rate to areas where congestion tends to occur, while Netvision (LIU et al., 2018), Pan et at. (PAN et al., 2019) and Yuan et al. (YUAN et al., 2022) leverage *Euler Circuits* strategies to orchestrate probing packets across the network. In the first (LIU et al., 2018), the network operator operates an API to instruct the probes without manipulating the underlying infrastructure. At the same time, in the second (PAN et al., 2019), it embeds source routing into INT probes and develops an Euler trail-based algorithm to cover the whole network with non-overlapping INT paths. INT-probe (PAN et al., 2021; YUAN et al., 2022) extend Pan et al. (PAN et al., 2019) and formulates the constrained path planning into an extended multi-depot k-Chinese postman problem (MDCPP-set) and then reduces it to a solvable minimum weight perfect matching problem and recovers link failures with

the help of an adjacent list at a Redis database. Similarly, INT-react (YUAN et al., 2022) developed a path-planning algorithm to achieve resilient network-wide telemetry over large-scale networks. By eliminating the unnecessarily repetitive calculation and changing the underlying data structure, which turns the time complexity from INT-path (PAN et al., 2019) $O(k(3E + V - 15k/2))$ to $O(3E)$, where k is the number of odd-degree vertices, E is the edge number, and V is the vertex number. However, none of these works consider devices/links may fail. Consequently, they do not provide a way to (i) detect a failure and (ii) circumvent a fault entirely in the data plane to mitigate control plane communication and synchronization. With both aspects above, ensuring low recovery latency and SLAs – such as in 5G networks (e.g., ultra-low latency) – would become much easier to be met.

Scano et al. (SCANO et al., 2021) extend P4 INT to 5G. Packet flows carrying selected latency-sensitive services are proposed to encompass the INT header from the user equipment, allowing for rerouting packets when soft failures are detected (e.g., increased bit errors, occasional packet loss, link congestion). Then, the collected data is sent to the ONOS controller, which triggers optical connection rerouting. Patcher (CASTRO et al., 2020) is a fault-tolerant mechanism that leverages the shortest path algorithm (DIJKSTRA et al., 1959) performing "patches" on affected probe flows where device faults occurred - e.g., due to energy failure, misconfiguration. Still, a control plane must be informed about the failure location to perform the "patches" and communicate the changes to the corresponding data plane devices. The failures must be transmitted to a control plane where the "patches" are performed and informed to data plane devices, incurring high latency – intolerable for low-latency applications (e.g., VoIP). RedPlane (KIM et al., 2021) is a fault-tolerant system for stateful in-switch applications. It ensures that after a failure and reroute, the same application state becomes available at the replacement switch by providing a set of APIs in P4 and continuously replicating updates to the state store through the data plane while Bankhamer et al. (BANKHAMER; ELSÄSSER; SCHMID, 2019) contributes with three non-deterministic local fast rerouting algorithms to deal with multiple link failures with minimal (first algorithm) to no header field modification (remaining algorithms). PINT (BASAT et al., 2020) and LINT aim to reduce INT collection overhead. In the first work, hash function outcomes are leveraged to probabilistically determine when to collect INT information for each network device to reduce INT collection overhead. Similarly, LINT (CHOWDHURY; BOUTABA; FRANÇOIS, 2021) independently decides on selectively reporting telemetry data on passing packets. Specifically, a predictor function leverages the Exponentially Weighted Moving Average (EWMA) computed in the data plane to retrieve the moving average of a data stream according to the exponentially decaying weights of the items in the stream according to the order they appear. Similarly, DeltaINT (SHENG; HUANG; LEE, 2021) instructs each node to embed a state to only a subset of traversed packets, thereby reducing the INT bandwidth usage. It measures the difference between a previous and the most recent measurement embedded state for

each traversed packet. *Flexile* (JIANG et al., 2022) prioritizes the availability of critical flows at cloud provider WANs to reduce flow loss. The proposal is subdivided into an offline phase and an online phase. Critical failure states are identified for each flow in the first, while bandwidth is allocated to prioritized flows in the second. NetView (LIN et al., 2020) supports multiple telemetry frequency collections. In summary, a greedy-based telemetry coordinator handles user requests, prioritizing high-frequency query cluster demands over the remaining requests. On the other hand, Fast-INT (YANG et al., 2020) can implement specific INT monitoring tasks on target points to shorten the monitoring time and make it more efficient. It uses "perspective INT packets" for the network's global view and "inspection INT packets" to feed a reinforcement algorithm that checks actions and executes policies on target points of the network. Castro et al. (CASTRO et al., 2021) generates probing packets according to an Inter Linear Programming (ILP) model to cover all telemetry items (e.g., ingress timestamp) and verify link connectivity. However, the model does not consider device/link failures that may occur. Similarly, SIMON (GENG et al., 2019) generates a mesh of probes to cover all the network links in Data Center Network (DCN)s to enable network state reconstruction with LASSO (TIBSHIRANI, 1996) network tomography algorithm. Specifically, NICs retrieve state variables such as queuing times, link utilization, and queue composition on edge devices. However, SIMON is limited to operating in DCN networks where it has previous knowledge.

2.5 Fast Rerouting mechanisms

More recently, Fast Rerouting (FRR) mechanisms have been used to implement fault-tolerant routing schemes directly on the fast path in the data plane. This approach reduces path recovery time by requiring minimal or no control plane intervention. PURR (CHIESA et al., 2019) is an FRR primitive that supports multiple failures and avoids re-circulations with a single TCAM lookup. First, the switches send the packet on the first active port in a sequence. In failure, multiple mechanisms may be used to re-establish the connection. However, it does not consider probe cycle space and time requirements. Blink (HOLTERBACH et al., 2019) passively monitors non-negligible TCP re-transmissions to detect remote link failures that disrupt end-to-end connectivity. The Flow Selector monitors connections, and a Failure Inference module probes all the next hops for availability and chooses a new working one. Subramanian et al. (SUBRAMANIAN et al., 2021) and Hsu et al. (HSU et al., 2020a) provide policy-compliant paths. In the first (SUBRAMANIAN et al., 2021), D2R logically split the topology into domains to reduce the memory overhead of alternative path computation. When the failure is detected, the D2R data plane stores this information in a register. When a packet arrives, the data plane uses this updated local link state and computes an end-to-end route that does not use the failed link, avoiding packet drops. Nevertheless, if a domain becomes internally disconnected due to multiple failures, D2R may not find a route to the destination -

even if it exists -, while with Contra (HSU et al., 2020a), probes are generated according to high-level policies (i.e., provided by the user). Then, these policies are converted to automata and intersect with the network topology. Also, switches mark links as failed when there have been no probes along the link for "k" probe periods. Similarly, Yamansavascilar et al. (YAMANSAVASCILAR et al., 2020) combines fast fail-over groups and SDN route calculations to keep QoS and QoE for video streaming applications. It continuously monitors the latency values of different paths and selects the optimal alternative path based on the current network conditions. It replaces the secondary path with a better alternative path if it becomes overloaded or experiences higher latency. FlowStalker (CASTANHEIRA; PARIZOTTO; SCHAEFFER-FILHO, 2019) subdivides the network into clusters and introduces the concept of crawler packets. A Depth-First Search (DFS) search is then applied on each cluster determines a single route that spans the whole cluster to gather information without the intervention of a network controller. In turn, Omnimon (HUANG et al., 2020) splits queries into partial operations and merges them to be executed on network entities (e.g., end-hosts, switches) according to its resources' constraint to monitor flows across the entire network. Similarly, NetSeer (ZHOU et al., 2020) reduces the duplication of reported flow events by only reporting packets that experienced event flows (e.g., congestion) and merging them to discover drops and corruptions over links and recovering the flow information of the events. In turn, SwitchPointer (TAMMANA; AGARWAL; LEE, 2018) combines in-network programmability and the available end-host resources to collect and monitor telemetry data to debug network events. Tan and Kuo (TAN; KUO, 2022) present a two-mode FRR mechanism with the (i) optimistic mode and the (ii) fallback mode. After a packet encounters the first failed link, it is forwarded in the optimistic mode, aiming solely to optimize the quality of fail-over routes. If it fails to provide high quality, employ the fallback mode to guarantee packet delivery. Zheng et al. (ZHENG et al., 2021) design a system named SR-INT. They propose a procedure to replace an SR label field with a bundle of INT fields and allow to change labels on the source switch to route through an alternative path at the last switch of each path segment to route SR-INT packets judiciously. Similarly, SQR (QU et al., 2019) caches a small number of recently transmitted packets on a switch, and in the event of a link failure, it re-transmits them on the appropriate backup network path. Instead, Aceves, Hemmati (GARCIA-LUNA-ACEVES; HEMMATI, 2019) only stores a vector in each data plane device to provide on-demand or proactive loop-free route alternatives. Similar to NetView (LIN et al., 2020) Sel-INT (TANG et al., 2019) and TeleNoise (DEMIANIUK; GORINSKY; KOGAN, 2021) consider INT data collection frequency. Sel-INT (TANG et al., 2019) supports real-time compilation and dynamic adjustment of telemetry instructions, sampling rate, and other telemetry behaviors. It leverages Select Group Table (SGT)s to selectively insert INT headers into OVS-POF based on its bucket's weight and a certain probability. In contrast, TeleNoise (DEMIANIUK; GORINSKY; KOGAN, 2021) inserts a few "sync bits" and

introduces two primitives (group affiliation and group completion) to mitigate network noises (e.g., packet reordering, packet loss). Wong et al. (WONG; LEE, 2023) is the first to integrate auto bootstrapping, network monitoring, fast fail-over, and control plane security in an in-band-controlled P4 network. Unlike OpenFlow switches, P4 switches behave passively—waiting for communication from the controller to be identified. Therefore, P4IBN sends device discovery messages periodically and allows rerouting with a set of pre-installed paths.

Despite recent efforts (CASTRO et al., 2020; HOLTERBACH et al., 2019; CHIESA et al., 2019; SUBRAMANIAN et al., 2021; TAN; KUO, 2022) to provide routing fail-over mechanisms in the data plane, these solutions have limitations when it comes to recovering INT data collection. Specifically, they struggle to meet the uninterrupted demand for INT data since their fail-over mechanisms are not explicitly designed for that purpose. Consequently, the downtime experienced in INT is primarily attributed to the time required for path updates in the controller, which can take several hundreds of milliseconds depending on the network size. Even with pre-computed paths, the delay in receiving failure notifications at the controller and propagating/installing new forward entries in the data plane devices remains significant, resulting in a loss of network visibility.

To overcome these challenges, we introduce `InPatching` as a pioneering approach that combines the concepts of INTO and FRR. This unique integration allows us to achieve a recovery decision-making approach with minimal reliance on the control plane. With `InPatching`, the data plane gains the ability to autonomously and harmoniously respond to faulty network conditions, ensuring swift responsiveness and outperforming existing control plane solutions by a remarkable factor of 18X. It enables continuous and accurate data collection even during network failures while maintaining efficient coordination. The autonomous nature of our approach significantly reduces the dependency on the control plane, thereby minimizing the time required for path updates and eliminating propagation delays that hinder network visibility. Furthermore, our approach enables different probe streams to exchange information in advance to reduce the convergence time for fault identification and avoidance.

We demonstrate the superiority of `InPatching` over existing control plane solutions through extensive evaluations. The efficiency of our approach is evident as it outperforms control-plane-based techniques by a remarkable factor of up to 18X. By combining the strengths of INTO and FRR, `InPatching` establishes a new paradigm in fault-tolerant network probing and recovery, paving the way for enhanced network performance and reliability.

2.6 Outline

This section summarizes all the presented works, sorting them into specific categories. The categories below detail the columns of Table 1. It provides a good insight into

the focus of these works, their limitations, and where our strategy fits concerning them. Below we described the used categories:

- **DP/CP.** DP/CP indicates whether the implementation logic of the proposal is located in the data plane (DP) and/or in the control plane (CP).
- **Granularity.** Granularity level indicates the level of control one has over the topology. For example, consider an approach that examines each packet independently to make a routing or forwarding decision. We can say that this approach has a "per-packet" level of granularity.
- **Fault-tolerant.** An approach is said to be fault-tolerant if it is capable of identifying and dealing with failures (e.g., link failures) through some mechanism either directly in the data plane or with the assistance of the control plane.
- **Orchestration awareness.** The notion of orchestration, as the name suggests, is information that summarizes essential details of the proposal's logic and how it organizes the collection of telemetry information and/or handles failures along the routes of the collection probes or active flows.

Proposal	DP/CP	Granularity	Fault-tolerant	Orchestration awareness
(LIU et al., 2018)	no/yes	per-RTT	no	It generates probe packets. Euler Circuits split the probe routing
(TAMMANA; AGARWAL; LEE, 2018)	yes/yes	per-flow	no	End-host and switch view. Lacks visibility at the core of the network
(RAMANATHAN; KANZA; KRISHNAMURTHY, 2018)	no/yes	per-link	no	Provides control over the link probing rates
(PAN et al., 2019)	yes/yes	unclear	no	Source Routing (SR) and Euler trail-based paths
(GENG et al., 2019)	no/yes	per-flow	no	Edge-based probe subsets for network tomography
(CASTANHEIRA; PARIZOTTO; SCHAEFFER-FILHO, 2019)	yes/yes	per-packet	no	Modular switch clustering for efficient data collection
(HOHEMBERGER et al., 2019)	no/yes	unclear	no	Machine learning-driven telemetry collection for monitoring applications
(MARQUES et al., 2019)	no/yes	per-flow	no	Prioritizes telemetry item collection frequency based on the heuristic approach
(HOLTERBACH et al., 2019)	yes/yes	per-flow	yes	Active for important prefixes, Flow Selector, Failure Inference
(CHIESA et al., 2019)	yes/no	unclear	yes	Circular port sequence policy, but ignores space and time requirements
(LIN et al., 2020)	yes/yes	per-packet	no	Node-centric monitoring for failure location with query aggregation and frequency multiplexing
(JIA et al., 2020)	yes/yes	per-flow	yes	Probe multicasting and SR-based proactive rerouting for 'gray failures'
(YANG et al., 2020)	yes/no	per-packet	no	INT packets provide global-view for network monitoring and reinforcement algorithms
(CASTRO et al., 2020)	yes/yes	per-flow	yes	Performs "patches"/detours to handle single link failures
(TANG et al., 2019)	yes/yes	per-flow	no	Selective INT header insertion into OVS-POF using SGTs based on weight and probability
(BASAT et al., 2020)	yes/yes	per-packet	no	Each device randomly embeds INT data into a packet
(HUANG et al., 2020)	yes/yes	per-flow	no	Split and merge queries based on resource constraints for network execution
(ZHOU et al., 2020)	yes/yes	per-flow	no	ASIC tracing detects events and gather flow information over links
(YAMANSAVASCILAR et al., 2020)	no/yes	per -RTT	yes	Continuous monitoring of latency values, selecting optimal alternative path
(CASTRO et al., 2021)	no/yes	per-flow	no	An ILP model covers the whole network links/devices to collect INT metrics

(PAN et al., 2021)	yes/yes	per-flow	yes	Link failures are recovered leveraging an adjacent list at a Redis database
(SHENG; HUANG; LEE, 2021)	yes/yes	per-packet	yes	DeltaINT minimizes INT bandwidth by embedding state changes in a subset of packets
(KIM et al., 2021)	yes/no	per-flow	yes	RedPlane enables seamless integration and continuous state replication in P4 applications through APIs
(CASTANHEIRA; PARIZOTTO; SCHAEFFER-FILHO, 2019)	yes/yes	per-packet	no	Switch clusters. DFS is run in each cluster to determine a single route that spans the cluster
(SCANO et al., 2021)	yes/no	per-packet	yes	Cover software failures. INT headers collect latency metrics for edge-cloud traffic mobility
(DEMIANIUK; GORINSKY; KOGAN, 2021)	yes/no	per-group	no	It inserts a few "sync bits" to distinguish packet groups and tells how many packets were lost per-group
(ZHENG et al., 2021)	yes/yes	per-packet	yes	An FRR mechanism allows to label the source switches through an alternative path
(CHOWDHURY; BOUTABA; FRANÇOIS, 2021)	yes/no	per-flow	no	Selective reporting of passing packets to reduce data plane overhead
(SUBRAMANIAN et al., 2021)	yes/yes	per-packet	yes	Switch clusters. As failures are detected, link states are updated into registers and D2R computes a new route
(TAN; KUO, 2022)	yes/no	per-flow	yes	Optimistic and fallback modes in a fast rerouting framework for reliable packet delivery with high-quality failover routes
(HSU et al., 2020a)	yes/no	per-flow	yes	Probes ensure user-based routing policies. A switch exceeding 'k' cycles to receive a new probe is considered a failure
(JIANG et al., 2022)	yes/no	per-flow	yes	Offline algorithm that prioritizes certain flows and takes probability of link failures into account
(GARCIA-LUNA-ACEVES; HEMMATI, 2019)	yes/no	per-packet	yes	The data plane has a vector of loop-free route alternatives
(QU et al., 2019)	yes/no	per-packet	yes	SQR forwards packets normally but creates a cached copy for marked packets during the link failure detection delay

(BANKHAMER; ELSÄSSER; SCHMID, 2019)	yes/no	per-flow	yes	Algorithm 1 (3-Permutations): Nodes store random permutations, selecting based on hop counter. Algorithm 2 (Intervals): Nodes partition network based on unique IDs, considering failover edges within sets. Algorithm 3 (Shared-Permutations): Nodes agree on undisclosed permutations, reducing maximum load if unknown to adversary.
(YUAN et al., 2022)	yes/yes	per-flow	no	Euler-trail based probing paths with balanced path lengths for a more synchronized collection
(WONG; LEE, 2023)	yes/yes	per-flow	no	P4 switches wait for the controller. Controller sends messages to discover new switches and establish control paths. The messages monitor links and detect network failures.
Our work	**yes/no**	**per-RTT**	**yes**	**The controller pre-installs forwarding rules. The data plane automatically recovers the failure by performing detours until the link failured is corrected.**

Table 1 – Comparison of proposals for telemetry collection in network monitoring.

3 INPATCHING DESIGN

In this chapter, we introduce our mechanism entitled `InPatching`. First, we provide an overview of the idea of the algorithm and discuss the benefits of using it in the data plane. Next, the design and implementation details of the data plane and control plane are presented. Finally, the experimentation environment and the main results obtained are presented.

3.1 Overview

`InPatching` is an in-network fail-over approach to INT-based monitoring mechanisms. Figure 6 depicts the in-network process in a programmable network infrastructure. In the example, three probing cycles f_1, f_2, and f_3 are responsible for continuously (i) collecting telemetry data and (ii) checking network connectivity. For simplicity, we omit the telemetry data collected at each device in the figure and assume a single independent failure of network links. In the example, network link G–H has failed and therefore affected probing cycles f_1 and f_3 – that is, INT packets are not returning to the INT Collector with collected telemetry data.

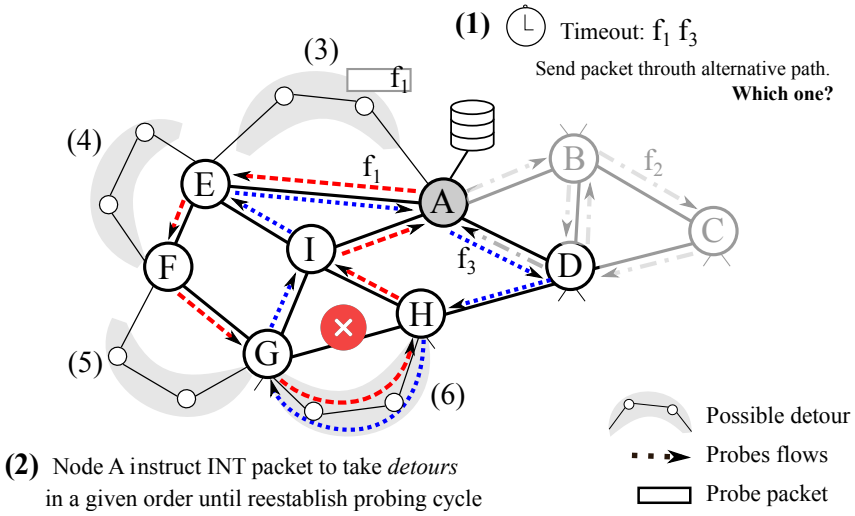

Figure 6 – Overview of the in-network InPatching strategy.

`InPatching` aims to offload the fail-over mechanism to the data plane to reestablish the monitoring with minimal to no control plane intervention. It relies (i) on alternative paths computed in advance and (ii) on a data plane heuristic strategy to select the proper alternative path. Upon a network link failure, probing packets start to time out in device A (in the examples, probing packets from f_1 and f_3), indicating to the data plane that probing packets have not arrived on time.

In turn, the data plane autonomously reacts by instrumenting the next packet of affected probing flows to take a detour in the main probing path to circumvent the

connectivity problem. In Figure 2, the data plane of device A is in charge of making that decision. The first attempt is to perform a detour between nodes A and E. The packet sent through this alternative path does not return (since the failure is on network link G–H). Then, the data plane attempts to use an alternative detour on nodes E and F, which also does not solve the connectivity problem. Eventually, the data plane uses an alternate path between nodes G and H, reestablishing the INT monitoring mechanism. Last, from time to time, the data plane attempts to utilize back the original forwarding path. Note that the data plane is taking heuristic decisions, and in the worst case, the number of attempts is $O(n)$, where n is the length of the probing path. As we discuss next, we can optimize these decisions to minimize the number of attempts, minimizing the overall recovery time.

Two building blocks are required to realize the potential of `InPatching`. The first is the data plane itself. As mentioned, the data plane must react to time-out conditions and instrument other data planes to take a detour. The second is the orchestration model (implemented in the control plane) responsible for constructing appropriate probing cycles and detours for network links.

3.2 Data plane design

The proposed data plane follows a distributed design similar to the well-known master-slave approach. One data plane logic is set as master and takes the major decisions (e.g., whether or not to take a detour). The others (i.e., the slave data planes) implement a simplified data plane logic to forward probing packets. It is essential to mention that all data plane logic has primary and alternative paths installed in advance by the control plane (we will discuss how we do that in the following subsection).

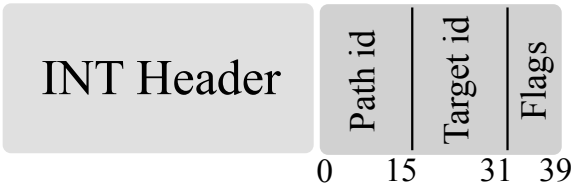

Figure 7 – `InPatching` header structure.

Our design works by extending the classical INT-based header struct. All probing packets have an INT header instructing which INT telemetry data must be collected at each forwarding device. The `InPatching` header struct is appended just after the INT header and is 5 Bytes long (i.e., 0.003% of a regular 1500B Maximum Transmission Unit (MTU) frame). These Bytes are used to monitor time-outs and to instruct the slave data plane on how to react. Figure 7 depicts the `InPatching` header struct. The first 16 bits represent the path id of a probing cycle. Then, the target id represents the slave data

plane *id* targeted by the master data plane. In other words, it represents the data plane where the detour decision is made. Last, there is an 8-bit word to describe possible control flags. `InPatching` utilizes these flags to notify the slave data plane to attempt to use the main path or to force them to stay in the alternative route. Our approach periodically injects a packet trying to return to the main course.

The master data plane logic needs to (i) keep track of probing cycle time-outs and (ii) react in case of a time-out by instrumenting slave data planes on possible detours. Figure 8 illustrates the whole procedure. The INT header is processed (steps 1-2) whenever a packet ingresses the device pipeline. It interprets INT instructions and collects INT data at a given device. The master data plane is uniquely identified by an *id*. It is responsible for (i) encapsulating packets in the `InPatching` header (step 4), (ii) keeping track of probing packet time-outs (steps 5 and 6), and (iii) choosing a slave/target data plane to perform a detour (step 7). For each probing cycle, the master data plane keeps an array of $|P|$ register (each of 48 bits) to store data plane timestamps, where P is the set of probing cycles in the network. When a packet ingress the master data plane, the data plane compares the packet ingress timestamp with the last seen packet timestamp of the same probing cycle P. If this difference exceeds a fixed threshold, the data plane assumes a time-out has happened. Then the master data plane notifies a slave to follow an alternative path (step 6). For that, the master stores a list of device *ids* in a probing cycle using an array of registers. The order of this *ids* in the array defines the order in which the data plane tries to apply a detour. This order is determined in advance by the control plane and can be adjusted based on failure probabilities, for example. When a detour is applied, the `InPatching` header is updated with the target data plane *id* (step 7). That information in the header field is used by the slaves to either apply alternative paths (steps 8-9) or not (step 10). Furthermore, whenever a packet returns to the master data plane (steps 11-12), we update the time-out data structure for each probing cycle to track the last seen probing packet. To properly control returning packets to the master, we use the field flags in the `InPatching` header. Last, the packet is sent out to a specific port in step 13.

3.3 Control plane design

As previously mentioned, the primary decisions of the `InPatching` approach are taken in the data plane. However, the control plane still plays a vital role by defining the probing paths (PAN et al., 2019; HOHEMBERGER et al., 2019; LIU et al., 2018; MARQUES et al., 2019), the detours, and the order in which they are applied by the data plane. In particular, the probe planning in the control plane directly impacts the efficiency of `InPatching`.

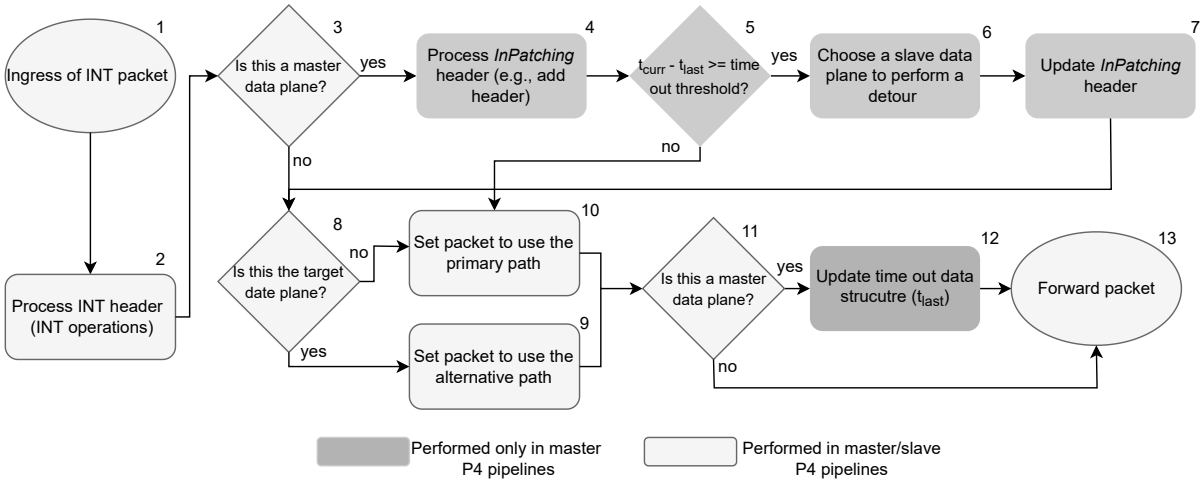

Figure 8 – Overview of the `InPatching` data plane procedure.

To understand the impact of probing paths on the efficiency of `InPatching`, let's consider the example in Figure 6. In case a failure occurs at a network link covered by only one probing path (e.g., network links E–F or B–C), the heuristic strategy taken by the `InPatching` data plane is in the worst case $O(n)$ (where n is the probing cycle length). It happens because affected probing paths f_1 and f_2 are disjointed, and independently of the order `InPatching` applies a detour, the worst case remains $O(n)$. However, if the failure happens at a network link shared by multiple probing paths, the decision can be optimized by cooperatively applying the heuristic by the data plane.

Suppose the master data plane knows the subset of overlapped probing paths. In the example of Figure 6, probing cycles $F_s = \{f_1, f_3\}$ share network links $S = \{(A-E), (G-H)\}$. In case of a network failure in S, the master data plane would reduce the search space to $O(\frac{|S|}{|F_s|})$. In that case, network flows in F_s would time out, and the master data plane could distribute the decision amongst them cooperatively. Whenever a probing flow finds a valid detour, it updates the master data plane accordingly.

Defining Optimized Probing Paths. To implement this unified solution by the data plane, the control plane runs an INTO model that can generate a valid set of probing cycles to cover the network infrastructure links by multiple probing cycles. Therefore, in the event of a network link failure, multiple affected INT cycles are promptly noticed and repaired these links.

The optimization problem described next ensures a valid INTO solution with a minimal number of probing cycles while providing at least a $\mathcal{K} \in \mathbb{N}^+$ cycle coverage per network link. We adopt a revised (and extended) version of the model proposed by (CASTRO et al., 2021). Similarly, we consider a programmable network infrastructure $G = (D, L)$ and a set of telemetry items V. Set D of network G represents P4-enabled forwarding devices $D = \{1, ..., |D|\}$, while set L links interconnecting devices $(d_1, d_2) \in (D \times D)$. There is a set of available telemetry data V, where each $v \in V$ has its size defined by the function $S : V \rightarrow \mathbb{N}^+$. Each device $d \in D$ can embed a subset of items

$V_d \subseteq V$ into a probing cycle packet $p \in P$. Packets in a probing cycle $p \in P$ have limited spare space to embed V data, defined by the function $U : P \to \mathbb{N}^+$. The set of probing cycles P is routed within the network G – that is, the packet is generated at an INT sink, routed through a subset of devices, and returns to its origin. A probing packet can visit a device $d \in D$ and not collect any associated telemetry items. We denote the origin of each cycle $p \in P$ as a fixed forwarding device $d_o \in D$.

The variable set of the optimization model is defined as follows. Variable $z_{p,v,i}$, $(\forall\, p \in P, v \in V, i \in D)$ indicate that a device i embeds INT data v into a probing packet from cycle p. Variable $x_{p,i,j}\,(\forall p \in P, (i,j) \in L)$ indicate that network link $(i,j) \in L$ is being used to route probing cycle $p \in P$. Last, variables $y_p\,(\forall p \in P)$ and $w_{p,i,j}\,(\forall p \in P, (i,j) \in L)$ are used, respectively, to count probing cycles used by the solution and by network link.

$$\textbf{Minimize} \quad \sum_{p=1}^{P} y_p \tag{3.1}$$

Subject to:

$$z_{p,v,i} \leq \sum_{j \in D} x_{p,j,i} \qquad \forall p \in P, i \in D, v \in V_i \tag{3.2}$$

$$z_{p,v,i} + x_{p,i,j} \leq 2 \cdot y_p \qquad \forall p \in P, (i,j) \in L, v \in V_i \tag{3.3}$$

$$\sum_{j \in D} x_{p,i,j} - \sum_{j \in D} x_{p,j,i} = 0 \qquad \forall p \in P, i \in D \tag{3.4}$$

$$\sum_{i \in S} \sum_{j \in S} x_{p,i,j} \leq |S| - 1 \qquad \forall p \in P, S \subseteq \{D - d_o\}, |S| \geq 2 \tag{3.5}$$

$$\sum_{p \in P} x_{p,i,j} + x_{p,j,i} \geq 1 \qquad \forall (i,j) \in L \tag{3.6}$$

$$\sum_{i \in D} \sum_{v \in V_i} z_{p,v,i} \cdot S(v) + \sum_{i \in D} \sum_{j \in D} x_{p,i,j} \leq U(p) \qquad \forall p \in P \tag{3.7}$$

$$x_{p,i,j} \leq \mathcal{B} \cdot w_{p,i,j} \qquad \forall p \in P, (i,j) \in L \tag{3.8}$$

$$\sum_{p \in P} w_{p,i,j} \geq \mathcal{K} \qquad \forall (i,j) \in L \tag{3.9}$$

$$z_{p,v,i} \in \{0,1\} \qquad \forall p \in P, v \in V_i, i \in D \tag{3.10}$$

$$y_p \in \{0,1\} \qquad \forall p \in P \tag{3.11}$$

$$x_{p,i,j} \geq 0 \qquad \forall p \in P, (i,j) \in L \tag{3.12}$$

$$w_{p,i,j} \in \{0,1\} \qquad \forall p \in P, (i,j) \in L \tag{3.13}$$

Constraint set (3.2) ensures that if telemetry item v is collected from forwarding device i, a probe should be routed through i. Constraint set (3.3) accounts for the number of probing cycles in use. In turn, constraint sets (3.4) and (3.5) ensure that cycles are well crafted. That is, constraint set (3.4) ensures flow conservation, while constraint set (3.5) is the well-known sub-tour elimination constraint. Then, constraint set (3.6) guarantees a probing cycle that covers at least one link direction. Constraint set (3.7) ensures the available probing packet capacity is not violated by the telemetry items collected or the network links being covered. Constraint sets (3.8) and (3.9) ensure a network link is covered by at least $\mathcal{K}$ probing cycles, where $\mathcal{B}$ is a significant natural number. Last, constraint sets (3.10)–(3.13) define the domains of output variables, while Equation (3.1) aims at minimizing the number of probing cycles.

Defining Detours. Another key element of `InPatching` is the detours performed by the probing cycles in the data plane in case of network link failure. We define detours as simple paths in G between two forwarding devices belonging to the same probing cycle P. We denote by $N_p = \{d_1, d_2, ..., d_|N_p|\}$ the ordered set of forwarding device of a probing path $p \in P$. For each ordered pair (d_1, d_2), (d_2, d_3), ..., $(d_{|N_p|-1}, d_{|N_p|})$ we define a simple alternative path between devices of a pair.

As previously mentioned, the data plane heuristically selects one alternative path to address the faulty condition in the event of a failure. The control plane also defines the order to instruct the master data plane on how to proceed. As for the experiments shown next, we adopt a simplified round-robin alternative following the order of nodes in the probing path P. However, other approaches are easily implemented, such as prioritizing high-probability faulty nodes first. Figure 9 summarizes the `InPatching` round-robin procedure. In this topology, we have three linearly connected switches. A is the master data plane, and the remaining switches are the slave data planes. Once a failure is detected by a time-out trigger at the master data plane (e.g., Time 1), A reads the next-hop detour list and selects the next switch (switch A itself) to perform the detour in the attempt to deviate the link failure. A detour is then performed at $A - B$ by A's alternative hop. However, the time-out condition persists. Next, the master data plane tries the next switch in the detour list (Time 2) – i.e., $B - C$ – successfully bypassing the link failure.

3.4 Evaluation

3.4.1 Setup

We implemented our `InPatching` data plane approach in P4 using BMv2 virtual switches and evaluated it in a Mininet environment. Probing cycles are defined according to our optimal Model 3.3. In contrast, probing packets are generated using a custom-made Scapy (BIONDI, 2010) code – a Python tool that allows us to handcraft customized packets. All experiments were performed on a machine equipped with an AMD Ryzen

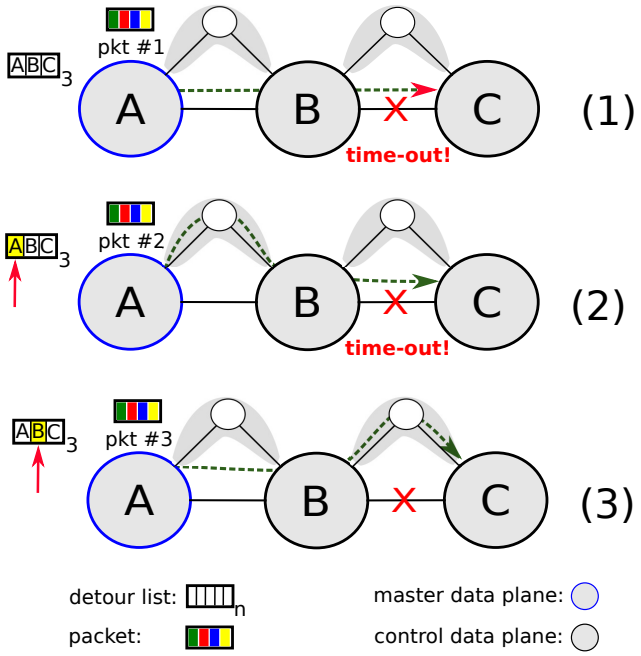

Figure 9 – Round-robin heuristic

Threadripper 3990X with 64 physical processors and 32 GB of memory, running Ubuntu 18.04.

Baseline. We compare `InPatching` against: (i) a traditional control-plane-based approach where requests are sent to a control plane, processed, and returned to the data plane; (ii) a *Euler-trail* based heuristic (PAN et al., 2019) named opp that minimizes overlapping probe circuits; and (iii) our previous theoretical model - where no failures were taken into account.

Reproducibility. Each experiment was repeated 30 times, which was enough to guarantee a confidence level higher than 95%. Our source code is available on GitHub (InPatching, 2022).

3.4.2 `InPatching` Data Plane vs. Control Plane Approaches

We first evaluate the effectiveness of `InPatching` in recovering the INT-based monitoring mechanism from single-link failures compared to a control-plane based solution (CASTRO et al., 2020) that performs "patches" on affected links by performing the shortest path algorithm (DIJKSTRA et al., 1959) on affected nodes/switches. To do that, we deploy a set of disjoint probing cycles in Mininet and inject link failures to network links in P. For this experiment, we set all network links in Mininet to have a 1ms delay, and we varied the time out in the data plane from 10ms to 100ms. The time-out is the reaction time when the data plane waits for an injected probing packet before reacting to it. More specifically, it is a threshold value to assume there is a port/link failure somewhere in the probing path. Therefore, whenever a time-out event occurs, we can either (i) send it to the `InPatching` mechanism, applying it over the next packet, or (ii) send it to the

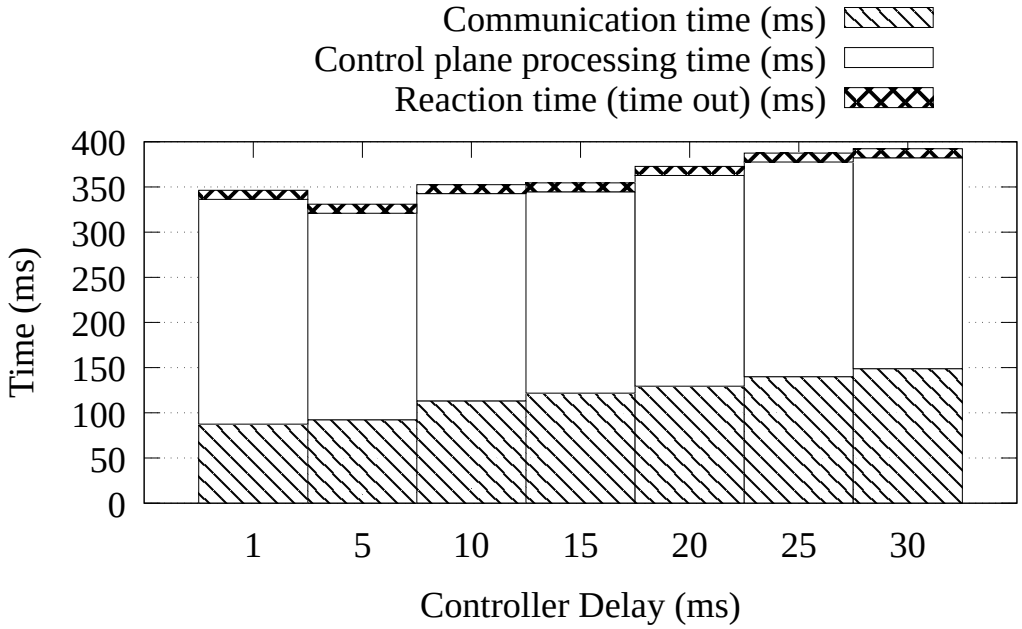

Figure 10 – Control plane approach average time (ms).

Figure 11 – Control plane approach.

control plane, and another mechanism will be responsible for computing a new route for the packet flow. Also, it is essential to mention that the data plane `InPatching` comes in three distinct "flavors" and all decisions. The first is stop-n-wait-based and is represented by $\texttt{InPatching}^{\omega}$. In this strategy, whenever a packet times out, the upcoming packets will follow the decision of the first one until the next time-out event (if any). The main advantage is that if a failure occurs at the first attempted link, fewer packets will be lost by bypassing the failure. The second strategy, represented by $\texttt{InPatching}^{\rho}$, is pipeline-based. In this approach, all pre-programmed next-hop choices are pipelined and forwarded to different neighbors simultaneously. In this way, it can find the failed link more rapidly than the previous one – which must wait for the next time-out event until a new neighbor chooses. Finally, the third approach is the cooperative `InPatching` ($\texttt{InPatching}^{\sigma}$) on which the behavior is based on the aforementioned stop-n-wait version ($\texttt{InPatching}^{\omega}$), with the main advantage that flows now share information about network failures in advance. More specifically, a different set of flows share a data structure that allows probing paths to overcome network failures more efficiently, thus reducing the overall convergence time.

Figure 11 shows the recovery time for the control plane, while Figure 12 and Figure 13 illustrates the same information for the data plane approaches. Figure 12 depicts the proposed stop-n-wait data plane approach ($\texttt{InPatching}^{\omega}$) considering an increasing value of time out. For this experiment, we consider a 5-hop probing cycle (i.e., link #1 to link #5). As we observe, the data plane increases the recovery time linearly as we increase the time out. For instance, for a data plane time out of 10ms, the recovery time of the

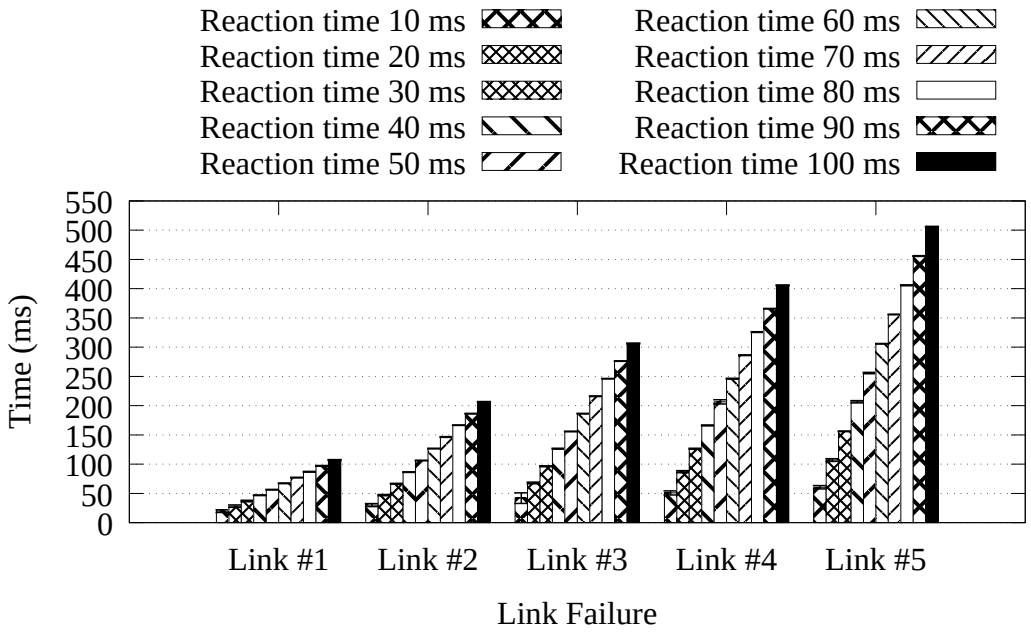

Figure 12 – `InPatching`$^{\omega}$ data plane approach.

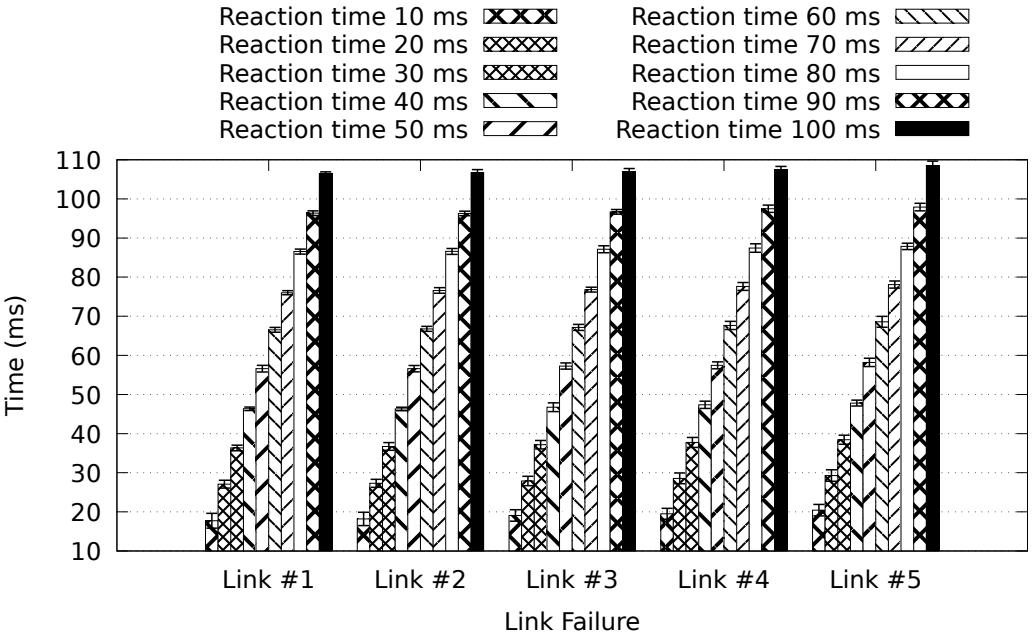

Figure 13 – `InPatching`$^{\rho}$ data plane approach.

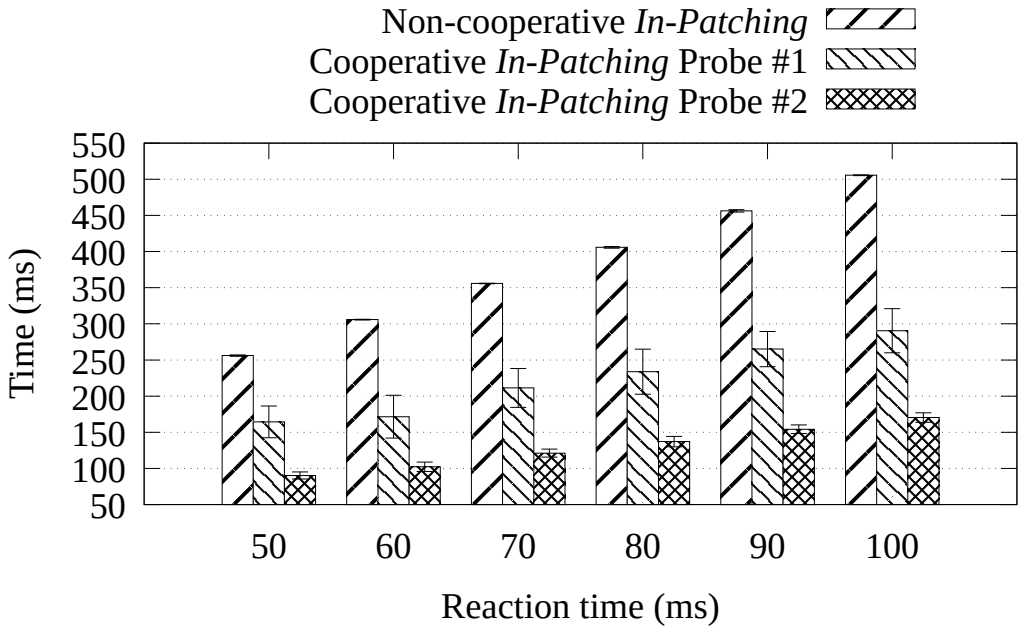

Figure 14 – Cooperative `InPatching` data plane vs non-cooperative.

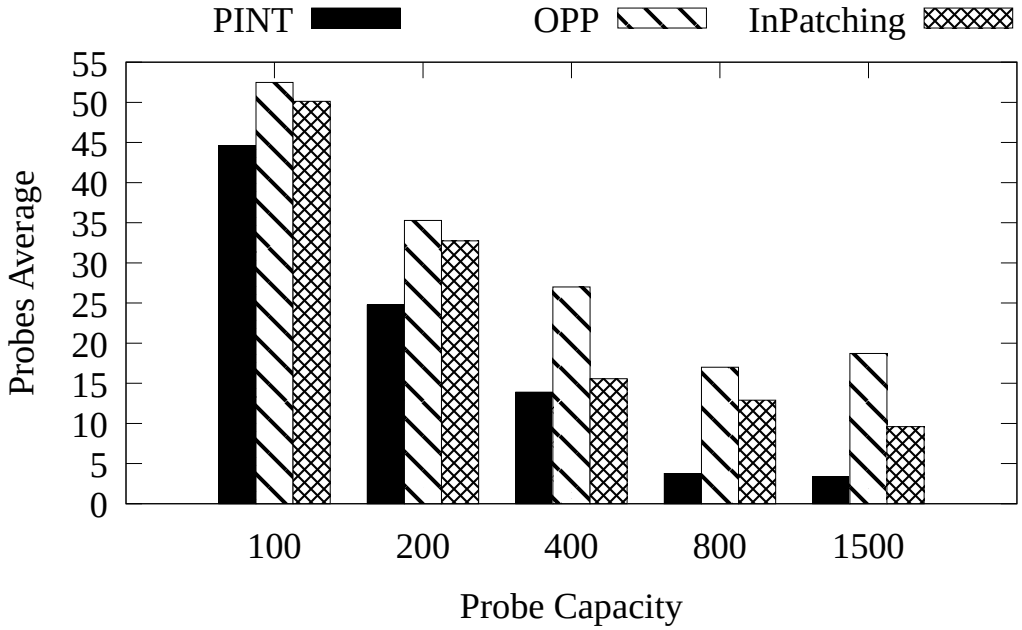

Figure 15 – `InPatching` optimal model

entire INT monitoring system is ∼19ms. Further, we can observe that the round-robin heuristic approach affects the data plane recovery time. The more distant the failure happens from the probing cycle origin, the more time is required to $\texttt{InPatching}^{\omega}$ to find a valid detour. However, even in the more distant link (e.g., link #5), the recovery time is below 60ms for a time out of 10ms. Finally, the results show minimal variance between the minimum and maximum obtained within the 30 executions. Similarly, Figure 13 depicts the pipeline strategy ($\texttt{InPatching}^{\rho}$). In contrast to $\texttt{InPatching}^{\omega}$, the reaction time remains constant for any link. It is because, in this strategy, whenever a packet times out, each upcoming packet selects a different node to perform a detour on simultaneously – hence the name *pipeline* – searching for the failed link in the topology and immediately informing the master data plane. Similarly to Figure 12, the experiments demonstrate little variation between the minimum and maximum values obtained during the executions. Finally, Figure 11 illustrates the recovery time in the control plane. In this experiment, we fixed the time out as 10ms. The control plane recovery time depends on (i) the control plane processing time (i.e., the time required to compute a solution), which encompasses an incoming cloned packet notification from the data plane, identification/notification of failed links across the topology to the control plane, alternative rule appliance leveraging the shortest path algorithm (DIJKSTRA et al., 1959); (ii) a user-defined reaction time (i.e., the time out) and; (iii) communication time (i.e., the time to send and receive information between infrastructure and controller) which is directly influenced by the delay in the link between the master data plane and the controller. When a packet is timed out by the data plane, it clones the packet and sends it to the control plane. In the case of the control plane, the network link failure order does not affect the reaction time since we are sending them to the control plane. We varied the control plane distance from the probing path origin (1 to 30 ms) to observe the communication time. In the best case (for 1ms control plane latency), `InPatching` is up to 18x quicker than the control plane (i.e., ∼ 350ms to the control plane vs. $19ms$ of `InPatching`).

3.4.3 The gain of overlapping probing INT cycles

Next, we evaluate the gains attained to `InPatching` when probing cycles are constructed by our model considering a given overlapping (i.e., $\mathcal{K} >= 2$) – i.e., $\mathcal{K}$ is the number of cycles overlapping each link in our topology. In our P4 implementation, we allow $\mathcal{K} = 2$, ensuring that at least two probing paths simultaneously cover the same network link. Figure 14 illustrates the recovery time for a reaction time (time out) varying from 50ms to 100ms. In case of a failure, two probing paths time out, and the `InPatching` data plane coordinately attempts to solve the failure by applying detours simultaneously for each probing path. In the experiment, we consider two probing cycles sharing at least one network link. In the event of a network link failure, both probing cycles attempt to recover the network state by applying a detour. The first one to find a valid detour notifies

the remaining ones in the master data plane. Figure 14 illustrates a non-cooperative – in this case, we chose the `InPatching`$^{\omega}$ – and the cooperative version of `InPatching` (`InPatching`$^{\sigma}$) Also, we fixed link #5 as the failed link, which is physically closer to probe #1. In the cooperative version, the first probing cycle (probe #1) is the one that finds a valid detour first, notifying the second probing cycle (probe #2) in advance, reducing the average convergence time. The cooperative `InPatching` (`InPatching`$^{\sigma}$) achieves a recovery time up to 50% and 300% faster than the non-cooperative version of `InPatching` (`InPatching`$^{\omega}$) for the first and second probing cycles, respectively. This time gain occurs because, as already mentioned, the cooperative flows share information about network failure points. Specifically, when one of the flows encounters the switch where the failure occurred, it notifies the other cooperative flows in advance about the switch where the failure occurred, reducing the total search time of the algorithm.

3.4.4 The cost of overlapping

Last, we evaluate how our proposed optimal model impacts the number of probing cycles generated in an INTO solution (in particular, Equations 3.8, 3.9, and 3.13). Our model uses IBM CPLEX Optimization Studio 12.9 to obtain optimum solutions. Our results are compared to (PAN et al., 2019; CASTRO et al., 2021). We follow a similar approach to them to generate a workload, setting $\mathcal{K} = 2$ as this is the parameter evaluated in the previous subsection. Figure 15 illustrates the number of probing cycles for increasing the size of probing capacity (from 100B to 1500B). As we increase the probing capacity, we observe a decrease in the number of deployed INT probing cycles, as we have more space on packets to collect INT data. For small packets (100B, 200B, 400B), our model requires, on average, 15% more probing cycles than the solution provided by (CASTRO et al., 2021). While for larger packets (800B and 1500B), our solutions require, on average, two additional cycles. Yet, our model produces 40% fewer cycles than (PAN et al., 2019).

3.4.5 Hardware Resource Usage

The discussed results so far have demonstrated how `InPatching` would behave in a virtualized environment, i.e., with `bmv2` switches. However, in an ideal scenario, it would be desirable to analyze these same experiments realistically. Offloading our code to hardware, e.g., to an architecture like TNA, incurs several limitations (see Section 4.2). However, for the scope of this work, only the compilation of the code with similar components and necessary logic for its operation has been performed, i.e., although the logic has not been tested with packet sending/receiving in physical topology. Finally, Table 2 and Table 3 summarize the minimum resource utilization required for the operation of `InPatching` in the TNA architecture.

3.4.5.1 MAU Resources

The presented Table 2 resources allocation offers insights into the stages of the pipeline of the TNA architecture. Firstly, stages 9 to 11 are not being utilized, indicating potential inefficiencies or unused portions within the architecture. Secondly, the Meter ALU column stands out, as it is consistently allocated 100% of its resources across all stages. This suggests the critical importance of the Meter ALU in performing calculations related to traffic metering or rate limiting, emphasizing its role in ensuring efficient network traffic management. Regarding the utilization of the Tind Result Bus, it is worth noting that despite not having any declared tables in the code, the necessity of using the Tind Result Bus for registers might depend on the specific implementation and requirements of the TNA architecture. Further analysis or clarification may be needed to determine the purpose or usage of the Tind Result Bus in such cases. Moreover, apart from the Meter ALU column, no other resource allocation exceeds 12.5% across all stages. This suggests a relatively balanced distribution of resources, ensuring efficient utilization of available capacity while avoiding overutilization in any particular aspect of the MAU. It reflects a well-designed allocation scheme that maximizes resource utilization without causing bottlenecks or excessive resource consumption.

- **Stage Number.** This column represents the stage number or level in the pipeline of the TNA architecture. Each stage performs specific operations on the incoming packets.
- **Exact Match Input xbar.** This column shows the percentage of exact match input crossbar utilization in each stage. The exact match input crossbar is responsible for routing packets to the appropriate lookup units for exact matching.
- **Gateway.** This column represents the percentage of gateway utilization in each stage. Gateways are responsible for forwarding packets to the appropriate next hop or output port based on the lookup results.
- **Static Random-Access Memory (SRAM).** This column indicates the percentage of SRAM utilization in each stage. SRAM is used to store intermediate results or configuration data within the MAU.
- **Map RAM.** This column shows the percentage of map RAM utilization in each stage. Map RAM stores the mapping information between input fields and the corresponding actions to be performed.
- **Very Long Instruction Word (VLIW).** This column indicates the percentage of VLIW instruction utilization in each stage. VLIW instructions allow multiple operations to be executed in parallel, enhancing the processing capabilities of the MAU.

Stage Number	Exact Match Input xbar	Gateway	SRAM	Map RAM	VLIW Instr	Meter ALU	Exact Match Search Bus	Tind Result Bus	Action Data Bus Bytes	Logical TableID
0	0.00%	0.00%	10.00%	16.67%	9.38%	100.00%	0.00%	31.25%	5.47%	31.25%
1	1.56%	6.25%	0.00%	0.00%	6.25%	0.00%	6.25%	12.50%	0.00%	12.50%
2	4.69%	6.25%	2.50%	4.17%	6.25%	25.00%	6.25%	12.50%	0.78%	12.50%
3	1.56%	6.25%	0.00%	0.00%	3.12%	0.00%	6.25%	6.25%	0.00%	6.25%
4	2.34%	12.50%	5.00%	8.33%	6.25%	50.00%	12.50%	18.75%	1.56%	25.00%
5	1.56%	0.00%	2.50%	4.17%	3.12%	25.00%	0.00%	6.25%	0.00%	6.25%
6	1.56%	6.25%	0.00%	0.00%	3.12%	0.00%	6.25%	6.25%	0.00%	6.25%
7	3.91%	6.25%	5.00%	8.33%	3.12%	50.00%	6.25%	12.50%	0.00%	12.50%
8	2.34%	6.25%	2.50%	4.17%	3.12%	25.00%	6.25%	6.25%	3.12%	6.25%
9	0.00%	0.00%	0.00%	0.00%	3.12%	0.00%	0.00%	6.25%	0.00%	6.25%
10	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
11	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Average	1.95%	5.00%	2.75%	4.58%	4.69%	27.50%	5.00%	11.88%	1.09%	12.50%

Table 2 – MAU ("stage") resources allocation.

- **Meter ALU.** This column shows the percentage of Meter Arithmetic Logic Unit (ALU) utilization in each stage. Meter ALU is responsible for performing calculations related to traffic metering or rate limiting.

- **Exact Match Search Bus.** This column represents the percentage of utilization of the exact match search bus in each stage. The exact match search bus is responsible for transferring packets or data between the lookup units and other components in the MAU.

- **Tind Result Bus.** This column indicates the percentage of utilization of the Tind result bus in each stage. The Tind result bus is responsible for transferring the index or address information obtained from the lookup units to other components within the MAU.

- **Action Data Bus Bytes.** This column represents the number of bytes transferred on the action data bus in each stage. The action data bus is responsible for carrying the action data or instructions between stages or components.

- **Logical TableID.** This column represents the logical table ID associated with each stage. It identifies the specific table or lookup unit being utilized in the MAU.

3.4.5.2 Tagalong Collection Resources

Table 3 presents the tagalong collection resource usage. A tagalong collection is a grouping or container that holds essential information about the packet, such as source and destination addresses, protocol information, and length associated with a particular packet

header. The table provides several important insights. Firstly, it is evident that only tagalong collections 0 to 2 are utilized in the Ingress stage (i.e., Gress = “I”). This suggests that the processing and manipulation of packet headers primarily occur during the ingress phase. Secondly, the analysis reveals that PHV containers of 16 bits experience higher usage compared to 8-bit or 32-bit containers. This observation can be attributed to a higher frequency of registers and variables declared with a width of 16 bits. It implies that the architecture prioritizes the storage and handling of data structures that require 16-bit representation. Moreover, considering the cumulative utilization across all collections, it is evident that approximately 30% of the available bits are allocated. This indicates a significant amount of unused capacity, potentially providing room for future expansions or enhancements to the system.

- **Collection.** This column represents the identifier or name of the collection.
- **Gress.** This column indicates the gress (stage) associated with each collection. "I" represents the ingress stage.
- **8b Containers Used.** This column shows the number of 8-bit containers used in each collection, along with the corresponding percentage of utilization.
- **16b Containers Used.** This column indicates the number of 16-bit containers used in each collection, along with the corresponding percentage of utilization.
- **32b Containers Used.** This column represents the number of 32-bit containers used in each collection, along with the corresponding percentage of utilization.
- **Bits Used.** This column shows the total number of bits used in each collection, which is calculated based on the utilization of containers.
- **Bits Allocated.** This column represents the total number of bits allocated for each collection. It indicates the total capacity available for storing data.

Collection	Gress	8b Containers Used	16b Containers Used	32b Containers Used	Bits Used	Bits Allocated
0	I	4 (100 %)	6 (100 %)	4 (100 %)	256 (100 %)	320 (125 %)
1	I	3 (75 %)	6 (100 %)	4 (100 %)	248 (96.9 %)	248 (96.9 %)
2	I	0 (0 %)	3 (50 %)	0 (0 %)	48 (18.8 %)	48 (18.8 %)
3		0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)
4		0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)
5		0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)
6		0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)
7		0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)	0 (0 %)
Total		7 (21.9 %)	15 (31.2 %)	8 (25 %)	552 (27 %)	616 (30.1 %)

Table 3 – PHV tagalong collection resources alocation.

4 FINAL REMARKS

This chapter summarizes our approach and describes our next research steps to finalize this work. We first provide an overview of the already accomplished results, then describe all required steps toward the final goal and the corresponding schedule of each activity.

4.1 Overview

This work introduced *InPatching*, an in-network fault-tolerant approach to INT monitoring solutions. To the best of our knowledge, this is the first work that combines the ideas of INTO and FRR. It occurs because `InPatching` performs (i) the orchestration of probes for telemetry collection directly in the data plane of the network while (ii) minimizing the role of the controller, reducing the network's reaction time and making it more autonomous to anomalous events. The main advantage of offloading this mechanism to the data plane is the increased ease of ensuring SLAs in applications that require low latency (e.g., remote surgeries, autonomous cars, VoIP). The key idea behind *InPatching* is to offload the fail-over mechanism to the data plane, allowing for the reestablishment of monitoring with minimal to no control plane intervention. This procedure relies on precomputed alternative paths in programmable switches and employing a data plane heuristic strategy to select the appropriate alternative path when a failure occurs.

The results of our experiments have demonstrated the effectiveness of `InPatching`. Specifically, we have found that our solution outperforms control plane-based solutions by a factor of 18X. Additionally, when coordinated with other probing flows, our approach can find a valid detour solution up to 3X quicker than existing methods. Importantly, our solution does not require more probing cycles, on average, than state-of-the-art In-Band Network Telemetry Orchestration approaches.

Moving forward, our subsequent research steps may focus on four main aspects. Firstly, while our current data plane heuristic strategy has shown promising results, there may be alternative heuristics that can further optimize the selection of alternative paths in the presence of failures or congestion. By exploring different heuristics, we can identify more efficient and practical approaches to enhance the overall performance of `InPatching`. By considering alternative strategies, we aim to recognize even more efficient methods for selecting the appropriate alternative path in case of failures or congestion. Secondly, we intend to implement our solution on programmable hardware. By leveraging the capabilities of programmable networking devices, such as programmable switches or network processors, we can achieve better control and flexibility in deploying and managing `InPatching` within real network environments. This implementation step will allow us to validate the practical feasibility of our approach and assess its performance in real-world scenarios. Moving on, Finally, as observed, the experiments assume that there is only probe traffic collecting information. However, this information is collected to determine the

behavior of real end-user traffic and applications autonomously. Therefore, it is expected to eventually carry out experiments considering the presence of user traffic for different applications in different scenarios to evaluate the impact of resources and decisions made by `InPatching`.

In future work, one can consider different paths to develop further and more robustly mature the idea of probe routing. For instance, an underexplored aspect in this study is establishing a strategy for placing master switches in the topology for load distribution. In the current implementation, a master switch is fixed, and all probe flows are managed by it. However, in a real-world scenario, the solution must be scalable, similar to the SDN controller placement problem, because (i) a single switch has limited memory to store control information for all flows, and (ii) there is a single point of failure in the network where the algorithm could cease to function.

4.2 Challenges and limitations

The TNA architecture introduced in Section 2.3 can be seen as a promising candidate for implementing a hardware version of our mechanism for several reasons already mentioned: (i) it supports P4 code, allowing our code to be rewritten for this platform; (ii) the proprietary software includes tools that facilitate port mapping, customization of channel speeds for different experimental scenarios. However, the code translation is not as straightforward due to several limitations in the architecture organization:

`RegisterActions`. Unlike the `bmv2` architecture, where built-in functions define read and write functions for registers, it is possible to define up to four `RegisterActions` for each register declared in the code. However, only one function can be called during packet processing in the ingress or egress pipeline. It is a limitation of our approach, which was not initially designed for the TNA architecture, as multiple register accesses are often required for certain conditions.

Packet Header Vectors. As presented in Section 2.3, PHVs handle packet header fields and other metadata along the packet processing pipeline. Among the metadata that can be allocated, code snippets for branching are included, and our strategy involves a considerable complexity of conditional code segments to ensure its correct functioning. However, the strategy of allocating PHVs for each stage of the pipeline often fails to find sufficient resources for the entire metadata structure of our code, which prevents its complete compilation. Among the issues presented, some possible solutions can be used in combination:

- **Code repositioning:** Certain conditional code segments can be rearranged to avoid the compiler's resource allocation strategies from using up all the resources of an MAU and make better use of the available resources.

- **Creating more complex `RegisterActions`:** Registers do not utilize PHV container memory. Conditions defined within a `RegisterAction` rely on Static Arithmetic Logical Unit (SALU), which is based on data stored in the registers and does not have dedicated memory, and, therefore, does not utilize PHV resources. Therefore, in more complex conditional segments, a register could be created to store a value indicating whether a condition evaluated in a RegisterAction was satisfied, thus avoiding using valuable PHV resources.

- **Using more tables:** In our current code version using the `bmv2` architecture, we solely utilized registers to store the information that should be maintained in the switch. However, the TNA architecture provides tables with match-action functionality, which can be leveraged to distribute and organize the metadata and associated operations efficiently. By utilizing multiple tables, we can allocate different parts of the metadata to different tables, thereby reducing the burden on individual tables and increasing the chances of successful compilation.

By implementing these solutions in combination, we could address the limitations of the TNA architecture and improve the compilation process for our code. Therefore, applying these strategies so far has yet to solve the compilation implications that incur from the compiler's strategies for PHV resource allocation. Initially, an attempt was made to use the `@pragma stage <stage> [entries]` directives available for table allocation in the TNA architecture in order to force the instantiation of a table to a specific stage. For example, let us assume that our table has 64 entries. Due to specific code requirements, half of these entries must be in stage 1. The `@pragma stage 1 32` directive can be placed above the table declaration.

However, it is essential to note that the TNA architecture does not explicitly support allocating specific table entries to different stages. The `@pragma stage` directive primarily influences the pipeline stage, where the control flow logic associated with the table is executed rather than dictating the allocation of individual entries.

A different approach may require more fine-grained control over table entry allocation across stages. It could involve restructuring the code or implementing custom logic within the table actions to distribute the entries according to specific requirements. Another option that has worked for some cases is using `RegisterActions` to reduce the memory used in conditional code segments within the algorithm. However, one disadvantage is the need to use an auxiliary register for each necessary condition throughout the implementation, which can hinder readability and flexibility for modifications.

While utilizing `RegisterActions` can help optimize resource usage and improve performance, it is essential to consider the trade-offs carefully. The increased complexity and potential reduction in code readability can make it harder to maintain and modify the code in the future.

To mitigate these challenges, it may be beneficial to thoroughly document the purpose and functionality of each `RegisterAction` and provide explicit comments to improve code understanding. Additionally, modularizing the code and organizing the `RegisterActions` in a structured manner can help alleviate some of the difficulties of maintaining and modifying the code.

Ultimately, the decision to use `RegisterActions` should be based on a thorough assessment of the specific requirements and constraints of the TNA architecture, considering both performance optimization and code maintainability aspects.

4.3 Future Work

In this section, we discuss future directions for this work, including topics not covered in the scope of this work and solutions to the current limitations presented.

Multiple master data planes. One of the first limitations to be observed in the current proposal is the simplification of experiments, where only one master data plane is used to manage the flows, meaning that all flows originate and return to the same switch. In the conducted experiments, a maximum of 2 flows were used simultaneously. However, in a real scenario, there could be hundreds or thousands of monitoring flows for application and service flows. Therefore, it is necessary to have a way to parameterize and position a certain number of master data planes in the physical topology.

Probe flow load balancing. Assuming that a set of master data planes exists, it is necessary to distribute/balance the flow load across these nodes to mitigate the impact on the available bandwidth and not hinder user traffic. Another way to achieve this is to consider the frequency at which these probes will be injected into the network (e.g., one packet per second).

Multiple link failures. Another identified future opportunity is to extend the logic of our mechanism to support multiple link failures along the probe paths since our approach can handle a single failure through a round-robin-based trial-and-error heuristic.

Failure selection heuristics. In addition to our round-robin-like strategy, it would also be interesting to investigate how other heuristics behave to minimize the total time required to detect failures directly in the data plane—for example, First In First Out (FIFO).

Evaluate hardware-based solution. So far, the code compiled for the TNA-like architecture – i.e., in terms of declarations and logic – is similar to the code for the virtual switch `bmv2` and, overall, has low resource utilization. However, the compiled code has not yet been adequately tested in an evaluation environment with packet injection and failures. This evaluation could confirm the already presented results or demonstrate other as-yet-unidentified aspects.

Background traffic injection. Lastly, background traffic injection is essential for placing and balancing probe flows. This particular need could generate parallel work. For example,

evaluating whether the best way to inject traffic is through traces – i.e., Packet Capture (PCAP) – or through a traffic generator tool and how this traffic should/can be generated is necessary.

BIBLIOGRAPHY

BANKHAMER, G.; ELSÄSSER, R.; SCHMID, S. Local fast rerouting with low congestion: A randomized approach. In: IEEE. **2019 IEEE 27th International Conference on Network Protocols (ICNP)**. [S.l.], 2019. p. 1–11. Cited 2 times in the pages 33 and 40.

BASAT, R. B. et al. Pint: Probabilistic in-band network telemetry. In: . New York, NY, USA: ACM, 2020. (SIGCOMM '20), p. 662–680. ISBN 9781450379557. Cited 3 times in the pages 32, 33, and 38.

BHAMARE, D. et al. Intopt: In-band network telemetry optimization for nfv service chain monitoring. In: IEEE. **ICC 2019-2019 IEEE International Conference on Communications (ICC)**. [S.l.], 2019. p. 1–7. Cited in page 32.

BIONDI, P. Scapy documentation (!). **vol**, v. 469, p. 155–203, 2010. Cited in page 46.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM 14**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Cited in page 28.

BOUABENE, G. et al. The autonomic network architecture (ana). **IEEE Journal on Selected Areas in Communications**, IEEE, v. 28, n. 1, p. 4–14, 2009. Cited in page 27.

CASE M. FEDOR, M. S. C. D. J. **Simple Network Management Protocol (SNMP)**. [S.l.], 1989. Disponível em: <https://www.hjp.at/doc/rfc/rfc1098.txt>. Cited in page 28.

CASTANHEIRA, L.; PARIZOTTO, R.; SCHAEFFER-FILHO, A. E. Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4. In: IEEE. **ICC 2019-2019 IEEE International Conference on Communications (ICC)**. [S.l.], 2019. p. 1–6. Cited 3 times in the pages 35, 38, and 39.

CASTRO, A. G. et al. Patcher: Towards fault-tolerant probing planning for in-band network telemetry. In: IEEE. **2020 IEEE Latin-American Conference on Communications (LATINCOM)**. [S.l.], 2020. p. 1–6. Cited 5 times in the pages 24, 33, 36, 38, and 47.

CASTRO, A. G. et al. Near-optimal probing planning for in-band network telemetry. **IEEE Communications Letters**, IEEE, v. 25, n. 5, p. 1630–1634, 2021. Cited 6 times in the pages 24, 25, 34, 38, 44, and 52.

CHEN, X. et al. Catching the microburst culprits with snappy. In: **Proceedings of the SelfDN**. New York, NY, USA: ACM, 2018. p. 22–28. ISBN 9781450359146. Cited in page 23.

CHIESA, M. et al. Purr: a primitive for reconfigurable fast reroute: hope for the best and program for the worst. In: **Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies**. [S.l.: s.n.], 2019. p. 1–14. Cited 3 times in the pages 34, 36, and 38.

CHOWDHURY, S. R.; BOUTABA, R.; FRANÇOIS, J. Lint: Accuracy-adaptive and lightweight in-band network telemetry. In: IEEE. **2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)**. [S.l.], 2021. p. 349–357. Cited 2 times in the pages 33 and 39.

CLAISE, B. et al. Cisco systems netflow services export version 9. RFC 3954, October, 2004. Cited in page 28.

CLARK, D. D. et al. A knowledge plane for the internet. In: **Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications**. [S.l.: s.n.], 2003. p. 3–10. Cited in page 27.

DEMIANIUK, V.; GORINSKY, S.; KOGAN, K. Telenoise: A network-noise module for in-band real-time telemetry. In: IEEE. **2021 IFIP Networking Conference (IFIP Networking)**. [S.l.], 2021. p. 1–9. Cited 2 times in the pages 35 and 39.

DIJKSTRA, E. W. et al. A note on two problems in connexion with graphs. **Numerische mathematik**, v. 1, n. 1, p. 269–271, 1959. Cited 3 times in the pages 33, 47, and 51.

ETSI. **Autonomous Networks, supporting tomorrow's ICT business**. 2020. Cited in page 27.

FEAMSTER, N.; REXFORD, J. Why (and how) networks should run themselves. **arXiv preprint arXiv:1710.11583**, 2017. Cited 2 times in the pages 23 and 27.

GARCIA-LUNA-ACEVES, J.; HEMMATI, E. Odvr: A unifying approach to on-demand and proactive loop-free routing in ad-hoc networks. In: IEEE. **2019 28th International Conference on Computer Communication and Networks (ICCCN)**. [S.l.], 2019. p. 1–11. Cited 2 times in the pages 35 and 39.

GENG, Y. et al. Simon: A simple and scalable method for sensing, inference and measurement in data center networks. In: **NSDI 19**. [S.l.: s.n.], 2019. p. 549–564. Cited 3 times in the pages 32, 34, and 38.

HOHEMBERGER, R. et al. Orchestrating in-band data plane telemetry with machine learning. **IEEE Communications Letters**, v. 23, n. 12, p. 2247–2251, Dec 2019. ISSN 1558-2558. Cited 4 times in the pages 24, 32, 38, and 43.

HOLTERBACH, T. et al. Blink: Fast connectivity recovery entirely in the data plane. In: **16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)**. [S.l.: s.n.], 2019. p. 161–176. Cited 3 times in the pages 34, 36, and 38.

HORN, P. Autonomic computing: Ibm's perspective on the state of information technology. New York, 2001. Cited in page 27.

HSU, K.-F. et al. Contra: A programmable system for performance-aware routing. In: **17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020**. [S.l.: s.n.], 2020. Cited 3 times in the pages 34, 35, and 39.

HSU, K.-F. et al. Adaptive weighted traffic splitting in programmable data planes. In: **Proceedings of the Symposium on SDN Research**. [S.l.: s.n.], 2020. p. 103–109. Cited in page 28.

HUANG, Q. et al. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In: **Proceedings of the ACM SIGCOMM**. [S.l.: s.n.], 2020. p. 404–421. Cited 2 times in the pages 35 and 38.

Huawei. **Huawei Core Network Autonomous Driving Network White Paper**. 2019. Disponível em: <https://carrier.huawei.com/~/media/CNBGV2/download/adn/Huawei-Core-Network-Autonomous-Driving-Network-White-Paper.pdf>. Cited in page 23.

InPatching. **InPatching - GitHub Repo**. 2022. Accessed on Nov, 2022. Disponível em: <https://github.com/arielgoes/InPatching>. Cited in page 47.

INTEL. **Intel Tofino**. 2021. [Access: July 04, 2023]. Disponível em: <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html>. Cited in page 30.

JACOB, B. et al. A practical guide to the ibm autonomic computing toolkit. **IBM Redbooks**, IBM Corp. International Technical Support Organization North Castle, NY, USA, v. 4, n. 10, p. 1–268, 2004. Cited in page 27.

JACOBS, A. S. et al. Affinity measurement for nfv-enabled networks: A criteria-based approach. In: IEEE. **2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [S.l.], 2017. p. 125–133. Cited in page 28.

JIA, C. et al. Rapid detection and localization of gray failures in data centers via in-band network telemetry. In: IEEE. **NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium**. [S.l.], 2020. p. 1–9. Cited 2 times in the pages 28 and 38.

JIANG, C. et al. Flexile: meeting bandwidth objectives almost always. In: **Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies**. [S.l.: s.n.], 2022. p. 110–125. Cited 2 times in the pages 34 and 39.

Juniper Networks. **The Self-Driving Network: The Future State of Operations for Cloud-Grade Networking**. 2017. Disponível em: <https://www.juniper.net/assets/fr/fr/local/pdf/pov/3200053-en.pdf>. Cited 2 times in the pages 23 and 27.

KIM, D. et al. Redplane: Enabling fault-tolerant stateful in-switch applications. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. [S.l.: s.n.], 2021. p. 223–244. Cited 2 times in the pages 33 and 39.

LAPOLLI, C.; MARQUES, J. A.; GASPARY, L. P. Offloading real-time ddos attack detection to programmable data planes. In: **2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [S.l.: s.n.], 2019. p. 19–27. ISSN 1573-0077. Cited in page 28.

LIN, Y. et al. Netview: Towards on-demand network-wide telemetry in the data center. **Computer Networks**, Elsevier, p. 107386, 2020. Cited 4 times in the pages 32, 34, 35, and 38.

LIU, Z. et al. Netvision: Towards network telemetry as a service. In: **IEEE ICNP**. [S.l.: s.n.], 2018. p. 247–248. ISSN 1092-1648. Cited 4 times in the pages 24, 32, 38, and 43.

MARQUES, J.; LEVCHENKO, K.; GASPARY, L. Intsight: Diagnosing slo violations with in-band network telemetry. In: **Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies**. [S.l.: s.n.], 2020. p. 421–434. Cited in page 28.

MARQUES, J. A. et al. An optimization-based approach for efficient network monitoring using in-band network telemetry. **Journal of Internet Services and Applications**, n. 1, p. 16, Jun 2019. Cited 5 times in the pages 24, 29, 32, 38, and 43.

MOLERO, E. C.; VISSICCHIO, S.; VANBEVER, L. Fast in-network gray failure detection for isps. In: **Proceedings of the ACM SIGCOMM 2022 Conference**. [S.l.: s.n.], 2022. p. 677–692. Cited in page 28.

PAN, T. et al. Int-probe: Lightweight in-band network-wide telemetry with stationary probes. In: IEEE. **2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)**. [S.l.], 2021. p. 898–909. Cited 2 times in the pages 32 and 39.

PAN, T. et al. Int-path: Towards optimal path planning for in-band network-wide telemetry. In: **IEEE INFOCOM**. [S.l.: s.n.], 2019. p. 1–9. Cited 8 times in the pages 23, 24, 32, 33, 38, 43, 47, and 52.

PHAAL, P.; PANCHEN, S.; MCKEE, N. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. RFC 3176, 2001. Cited in page 28.

QU, T. et al. Sqr: In-network packet loss recovery from link failures for highly reliable datacenter networks. In: IEEE. **2019 IEEE 27th International Conference on Network Protocols (ICNP)**. [S.l.], 2019. p. 1–12. Cited 2 times in the pages 35 and 39.

RAMANATHAN, S.; KANZA, Y.; KRISHNAMURTHY, B. Sdprober: A software defined prober for sdn. In: **Proceedings of the Symposium on SDN Research**. [S.l.: s.n.], 2018. p. 1–7. Cited 2 times in the pages 32 and 38.

SCANO, D. et al. Extending p4 in-band telemetry to user equipment for latency-and localization-aware autonomous networking with ai forecasting. **Journal of Optical Communications and Networking**, Optica Publishing Group, v. 13, n. 9, p. D103–D114, 2021. Cited 3 times in the pages 32, 33, and 39.

SHENG, S.; HUANG, Q.; LEE, P. P. Deltaint: Toward general in-band network telemetry with extremely low bandwidth overhead. In: IEEE. **2021 IEEE 29th International Conference on Network Protocols (ICNP)**. [S.l.], 2021. p. 1–11. Cited 2 times in the pages 33 and 39.

SINGH, S. K. et al. Revisiting heavy-hitters: Don't count packets, compute flow inter-packet metrics in the data plane. In: **ACM SIGCOMM Poster**. New York, NY, USA: ACM, 2020. p. 1–4. Cited in page 23.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: **Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: Association for Computing Machinery, 2013. (HotSDN '13), p. 127–132. ISBN 9781450321785. Cited in page 28.

STRASSNER, J.; AGOULMINE, N.; LEHTIHET, E. Focale: A novel autonomic networking architecture. 2006. Cited in page 27.

SUBRAMANIAN, K. et al. D2r: Policy-compliant fast reroute. In: **Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)**. [S.l.: s.n.], 2021. p. 148–161. Cited 3 times in the pages 34, 36, and 39.

TAMMANA, P.; AGARWAL, R.; LEE, M. Distributed network monitoring and debugging with switchpointer. In: **15th USENIX NSDI 18**. Renton, WA: [s.n.], 2018. p. 453–456. ISBN 978-1-931971-43-0. Cited 3 times in the pages 23, 35, and 38.

TAN, H.-K.; KUO, T.-W. Optimistic fast rerouting. In: IEEE. **ICC 2022-IEEE International Conference on Communications**. [S.l.], 2022. p. 1692–1697. Cited 3 times in the pages 35, 36, and 39.

TANG, S. et al. Sel-int: A runtime-programmable selective in-band network telemetry system. **IEEE transactions on network and service management**, IEEE, v. 17, n. 2, p. 708–721, 2019. Cited 2 times in the pages 35 and 38.

The P4.org Applications Working Group. **In-band Network Telemetry (INT) Dataplane Spec ification**. 2020. Disponível em: <https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf>. Cited 2 times in the pages 23 and 28.

TIBSHIRANI, R. Regression shrinkage and selection via the lasso. **Journal of the Royal Statistical Society: Series B (Methodological)**, Wiley Online Library, v. 58, n. 1, p. 267–288, 1996. Cited in page 34.

WONG, T.-S.; LEE, S. S. Design of an in-band control plane for automatic bootstrapping and fast failure recovery in p4 networks. **IEEE Transactions on Network and Service Management**, IEEE, 2023. Cited 2 times in the pages 36 and 40.

YAMANSAVASCILAR, B. et al. Fault tolerance in sdn data plane considering network and application based metrics. **Journal of Network and Computer Applications**, Elsevier, v. 170, p. 102780, 2020. Cited 3 times in the pages 28, 35, and 38.

YANG, F. et al. Fast-int: Light-weight and efficient in-band network telemetry in programmable data plane. In: IEEE. **2020 IEEE 92nd Vehicular Technology Conference (VTC2020-Fall)**. [S.l.], 2020. p. 1–5. Cited 2 times in the pages 34 and 38.

YUAN, Q. et al. Int-react: An o (e) path planner for resilient network-wide telemetry over megascale networks. In: IEEE. **2022 IEEE 30th International Conference on Network Protocols (ICNP)**. [S.l.], 2022. p. 1–11. Cited 3 times in the pages 32, 33, and 40.

ZHENG, Q. et al. Highly-efficient and adaptive network monitoring: When int meets segment routing. **IEEE Transactions on Network and Service Management**, IEEE, v. 18, n. 3, p. 2587–2597, 2021. Cited 2 times in the pages 35 and 39.

ZHOU, Y. et al. Flow event telemetry on programmable data plane. In: **Proceedings of the ACM SIGCOMM**. [S.l.: s.n.], 2020. p. 76–89. Cited 2 times in the pages 35 and 38.