FEDERAL UNIVERSITY OF PAMPA

Ariel Góes de Castro

Towards Probe Planning for In-band Network Telemetry

Alegrete
2021

Ariel Góes de Castro

Towards Probe Planning for In-band Network Telemetry

Monography submitted to the Undergraduate Program in Computer Science of Federal University of Pampa in partial fulfillment of the requirements for the Bachelor's degree in Computer Science.

Supervisor: Prof. Dr. Marcelo Caggiani Luizelli

Alegrete
2021

SERVIÇO PÚBLICO FEDERAL

MINISTÉRIO DA EDUCAÇÃO

Universidade Federal do Pampa

ARIEL GÓES DE CASTRO

Towards Probe Planning for In-band Network Telemetry

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação.

Dissertação defendida e aprovada em: 10, maio de 2021.

Banca examinadora:

_____

Prof. Dr. Marcelo Caggiani Luizelli

Orientador

UNIPAMPA

_____

Prof. Dr. Arthur Francisco Lorenzon

UNIPAMPA

Prof. Dr. Fábio Diniz Rossi

IFFar

Assinado eletronicamente por **ARTHUR FRANCISCO LORENZON**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/05/2021, às 10:18, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **Fábio Diniz Rossi**, **Usuário Externo**, em 12/05/2021, às 10:39, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **MARCELO CAGGIANI LUIZELLI**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 12/05/2021, às 14:28, 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 **0523597** e o código CRC **52A72162**.

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. In addition, I am grateful for all the friendships I created during my undergraduate period. I would like to thank the partnerships and the sacred nights in Napoli, highlighting the illustrious presence of: “Chico”, “Pipo”, Rafael, “Doug”, Marcelo, Robson and Diego. I would like to thank Victor for having been my companion throughout this journey, being always present no matter what. In addition, I would like to thank the people who helped me a lot to grow professionally and as a person throughout my graduation. I am not able to express everything in a few words, but I would like to thank some of them in particular. First, 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. Arthur Francisco Lorenzon for his sincere friendship and constant support expressed in different ways. Both were very close to me along these years and I believe that both are examples of people and professionals to be followed. Finally, I apologize for not mentioning everyone involved in this short thanks.

"When you're backed against the wall, break the goddamm thing down." – Harvey Specter

RESUMO

In-Band Network Telemetry (INT) é um mecanismo emergente para o monitoramento de infraestruturas de redes programáveis. Apesar de iniciativas recentes para orquestrar a coleta de estatísticas *in-band* dos dispositivios da infraestrutura, as abordagens existentes ainda são limitadas quanto a (i) coleta de dados de telemetria de forma eficiente, quando sujeito à restrições físicas de dispositivos programáveis e (ii) a recuperação do mecanismo de monitoramento quando sujeita a falhas nos dispositivo de encaminhamento. Neste trabalho, propõe-se o Probing Planning for In-Band Network Telemetry (P^2INT) e o Fault-Tolerant Probing Planning for In-Band Network Telemetry (FP^2INT). P^2INT coordena como os pacotes de probes ativos são gerados e roteados de modo a garantir que todos os enlaces sejam visitados e que todas as informações de telemetria da infraestrutura de rede sejam coletadas. Formaliza-se o problema de otimização que o P^2INT resolve com um modelo de Programação Linear Inteira Mista. Para resolver o modelo de maneira eficiente, propõe-se uma *math-heuristic* (isto é, uma heurística baseada em *mathematical programming*). A ideia chave da *math-heuristic* proposta é resolver partes do modelo de otimização iterativamente, determinando quais variáveis do modelo são otimizadas e quais são fixas. Ainda, formalizou-se o FP^2INT e propôs-se uma abordagem heurística, o *Patcher* – planejamento de probes tolerante a falhas para INT, o qual reconstrói ciclos de probes afetados por dispositivos falhos, garantindo que todos os enlaces e estatísticas *in-band* afetados mantenham-se ativos no monitoramento realizado. Este trabalho é o primeiro esforço de pesquisa na direção de resolver formalmente esses dois problemas. Os resultados mostram que P^2INT supera os trabalhos existentes na literatura por um fator de até 6x em relação ao número de ciclos de probes gerados, enquanto *Patcher* reduz o número de ciclos de probes necessários em até 5,5x em comparação com solução existentes, sem aumentar a sobrecarga em coletores INT.

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

ABSTRACT

INT is gaining traction as an advanced network monitoring approach. Despite a few recent initiatives to orchestrate the collection of in-band network statistics, state-of-the-art approaches fall short when it comes to efficiently (i) collect telemetry items while subjected to real-world constraints and (ii) considering the possibility of device failures (e.g., power failure, hardware failure). In this research work, we introduce (i) P^{2}INT and FP^{2}INT. P^{2}INT coordinates how probing packets are generated and routed to ensure that all links are covered so that the required in-band network telemetry data is collected. We theoretically formalize the problem as a Mixed-Integer Linear Programming model and propose an efficient mathematical programming-based heuristic to solve it, namely fix-and-optimize, which iteratively chooses which model's variables would be optimized, and which ones would be fixed (hence the name fix-and-optimize). Also, we theoretically formalize FP^{2}INT and propose a mechanism, namely *Patcher* – fault-tolerant probing planning for INT, which reconstructs probing cycles affected by failure nodes and optimizes them while ensuring all non-affected links/statistics are still being traversed/collected. To solve this problem efficiently, we introduce a heuristic that wisely finds a high-quality solution. To the best of our knowledge, this is the first attempt to formally define and solve these problems. Results show that that P^{2}INT outperforms the closest contender by a factor of up to 6x concerning the number of probing cycles generated, while *Patcher* reduces the number of probe cycles needed up to 5.5x compared to a state-of-the-art solution, while not increasing the INT collectors' load.

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

LIST OF FIGURES

Figure 1 – Example of int usage in a network infrastructure. 24
Figure 2 – Example of a solution for the Probing Planning problem, illustrating a snapshot where probing packets (f_1, f_2, f_3) collect telemetry data from selected network devices. 25
Figure 3 – SDN architecture overview. 28
Figure 4 – Programming a target with P4. 30
Figure 5 – P4 abstract forwarding model. 31
Figure 6 – Traditional and INT monitoring overview. 32
Figure 7 – Number of probing cycles for an increasing network size. 48
Figure 8 – Number of probing cycles for different probe capacity. 49
Figure 9 – Probe capacity usage. 49
Figure 10 – Transmission overhead. 50
Figure 11 – Collector load. 51
Figure 12 – Link overhead. 51
Figure 13 – Runtime. 52
Figure 14 – Reconstruction of the telemetry solution using *Patcher*. 54
Figure 15 – Number of probing cycles for an increasing network size. 59
Figure 16 – Collector load. 60
Figure 17 – Minimum distance to the closest collector. 61
Figure 18 – Average difference of $|\mathcal{C}_p \cap \overline{\mathcal{C}_p}|$. 61
Figure 19 – Average difference of $|\mathcal{T}_p \cap \overline{\mathcal{T}_p}|$. 62
Figure 20 – Trivial solution runtime. 62
Figure 21 – Patcher runtime. 63

LIST OF TABLES

Table 1 – OpenFlow action set. 29
Table 2 – Primary metadata in v1model switch. 34

LIST OF SYMBOLS

AMD Advanced Micro Devices

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

BGP Border Gateway Protocol

DCN Data Center Network

DDoS Distributed Denial of Service

DFS Depth-First Search

DPDK Data Plane Development Kit

FP^{2}INT Fault-Tolerant Probing Planning for In-Band Network Telemetry

FPGA Field Programmable Gate Array

HPCC High-Performance Computing Cluster

IBM International Business Machines

IETF Internet Engineering Task Force

INT In-Band Network Telemetry

ISP Internet Service Provider

KDN Knowledge-Defined Network

MDT Model-Driven Telemetry

MIB Management Information Base

MILP Mixed-Integer Linear Programming

MTU Maximum Transmission Unit

NIC Network Interface Card

NP Nondeterministic Polynomial Time

NSH Network Service Header

OID Object Identifier

ONOS Open Network System Operating System

P^{2}INT Probing Planning for In-Band Network Telemetry

P4 Programming Protocol-independent Packet Processors

POF Protocol Oblivious Forwarding

PTP Precision Time Protocol

QoE Quality of Experience

RAM Random Access Memory

RTT Round Trip Time

SDN Software-Defined Network

SLA Service Level Agreement

SNMP Simple Network Management Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol

VNS Variable Neighborhood Search

CONTENTS

1 INTRODUCTION 23
1.1 Context and Motivation 23
1.2 Research Problem 25
1.3 Objectives and Contributions 25
1.4 Outline 26

2 BACKGROUND AND RELATED WORK 27
2.1 Network Programmability 27
2.1.1 Control Plane Programmability 27
2.1.2 Data Plane Programmability 29
2.1.3 Network Monitoring 30
2.1.3.1 Traditional Network Monitoring vs INT monitoring 30
2.1.3.2 In-band Network Telemetry 32
2.2 Related Work 33
2.2.1 Programmable Control Plane 34
2.2.2 Programmable Data Plane 35

3 OPTIMAL AND SCALABLE PROBE PLANNING FOR IN-BAND NETWORK TELEMETRY 41
3.1 Problem Overview 41
3.2 Proposed Model 41
3.3 A Math-Heuristic Approach to P^{2}INT 44
3.3.1 Overview 44
3.3.2 Obtaining an initial solution 45
3.3.3 Neighborhood selection and prioritization 46
3.4 Results 47
3.4.1 Setup 47
3.4.2 Results 48

4 FAULT-TOLERANT PROBING PLANNING FOR IN-BAND NETWORK TELEMETRY 53
4.1 Problem overview 53
4.2 Model description and notation 54
4.3 Proposed Heuristic Approach 56
4.4 Evaluation 58
4.4.1 Setup. 58
4.4.2 Results 59

5 FINAL REMARKS 65

5.1 Achievements 65
5.2 Future Work 66

BIBLIOGRAPHY 69

ANNEX A – ORCHESTRATING IN-BAND DATA PLANE TELEMETRY WITH MACHINE LEARNING 75

ANNEX B – PATCHER: TOWARDS FAULT-TOLERANT PROBING PLANNING FOR IN-BAND NETWORK TELEMETRY 81

ANNEX C – NEAR-OPTIMAL PROBING PLANNING FOR IN-BAND NETWORK TELEMETRY 89

ANNEX D – THE ACTUAL COST OF PROGRAMMABLE SMARTNICS: DIVING INTO THE EXISTING LIMITS 95

Index 111

1 INTRODUCTION

In this chapter, we discuss the problem of orchestrating probes for In-band Network Telemetry. First, we give a brief introduction to INT networks, followed by the problem definition and its constraints. Then, we formally define the problem and our contributions to this research.

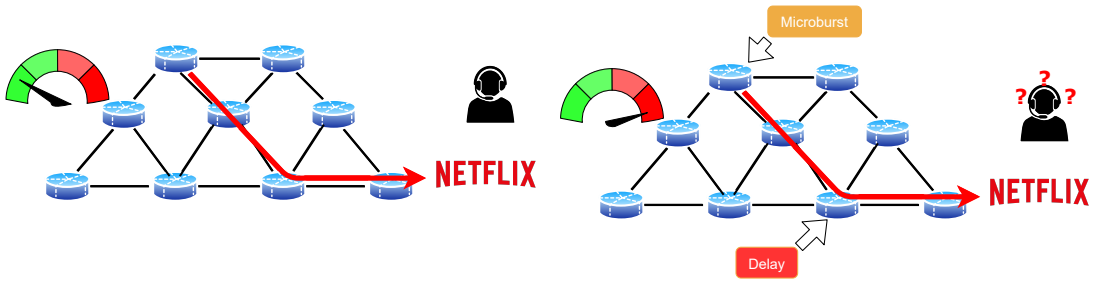
1.1 Context and Motivation

INT has recently emerged as a promising near real-time network monitoring to improve network visibility (JEYAKUMAR et al., 2014; LIU et al., 2018; GENG et al., 2019) in contrast to traditional solutions (e.g., Simple Network Management Protocol (SNMP)) that lack network coverage and scalability. Since its inception, INT has been successfully applied to a series of use cases, including the short-lived network behaviors (e.g., microburst) detection and network anomalies (e.g., routing violations and black holes), to name a few. Due to the rich spectrum of benefits behind INT adoption, there is increasing attention from the Internet Engineering Task Force (IETF) [1] and the whole networking ecosystem. It is noteworthy the INT concept has been unlocked and fostered by the rapid adoption of programmable data planes and domain-specific networking description languages (e.g., Programming Protocol-independent Packet Processors (P4) (BOSSHART et al., 2014)).

As an example of a possible utilization of INT, let us consider that you are at the commodity of your home (see Figure 1) watching some streaming video service (e.g., Netflix) which has a set of minimum requirements, i.e., an Service Level Agreement (SLA) (e.g., minimum bandwidth) to work properly. First, the quality of the video and sound are crystal clear (Figure 1a). Suddenly, the video starts “freezing" (Figure 1b) and it becomes an unpleasant experience for you – i.e., Quality of Experience (QoE). Then, you decide to call your Internet Service Provider (ISP) to report the situation and hope for a quick and time-efficient solution. However, most of the time, neither the user nor the network operator has enough information from the infrastructure to provide a quick and a time-manner way to troubleshoot and correct network anomalies. Therefore, the monitoring of networks is increasingly necessary to keep the quality of services in the network. In the illustrated example, one could collect telemetry information from the infrastructure (e.g., from switches, routers) along the path taken from your house to the Netflix server you are accessing to assist the network operator in taking corrective actions.

In short, INT consists of instrumenting the collection of low-level network monitoring statistics directly from the data plane. INT allows to collect information either (i) passively (in-band) or (ii) actively (probe packets). In the first, we leverage free space in real traffic packets to embed telemetry information. However, the in-band approach has

[1] For instance: <https://tools.ietf.org/html/draft-song-opsawg-ntf-03>



(a) Initial situation.

(b) Failure situation.

Figure 1 – Example of int usage in a network infrastructure.

some limitations: (i) free space is variable for each traffic packet; (ii) there is little to no control over the route taken by the information collected on the devices – i.e. until they are directed to a collector for further analysis – making it difficult, for example, to collect data at a specific point in the network. On the other hand, probing packets consist of specially-crafted packets that instruct programmable forwarding devices to collect telemetry data and prevent aforementioned problems, but the probes must be orchestrated and injected cautiously into the network so that an excessive amount does not generate processing overhead in the routing devices. Therefore, in both problems, we chose to use active monitoring (i.e., probing packets) to obtain more control in the orchestration of information collection. Figure 2 illustrates the entire INT process. In the first step, we generate the probing packets to instrument the telemetry data collection along a given path. For example, the red flow (i.e. f_1) – that is routed through the forwarding devices A, E, F, G, H, and I – carries instructions to collect telemetry data from devices A to H. In the second step, the collected telemetry data is extracted and reported to an INT collector.

Recent investigations have made the first efforts towards the orchestration of INT data collection to improve network-wide visibility. Liu et al. (LIU et al., 2018), Pan et al. (PAN et al., 2019), and Geng et al. (GENG et al., 2019) have focused on performing network telemetry through active INT-based probing packets. These strategies have relied either on *Euler Circuits* (LIU et al., 2018; PAN et al., 2019) or on actual routing paths (GENG et al., 2019) to instrument the forwarding of probes. In turn, Marques et al. (MARQUES et al., 2019) and Hohemberger et al. (HOHEMBERGER et al., 2019) have focused on the embedding of INT data into production network packets. Marques et al. (MARQUES et al., 2019) designed heuristic approaches to orchestrate how network flow packets collect network telemetry data, and Hohemberger et al. (HOHEMBERGER et al., 2019) designed a machine-learning-based model that wisely chooses and collects INT data based on its importance.

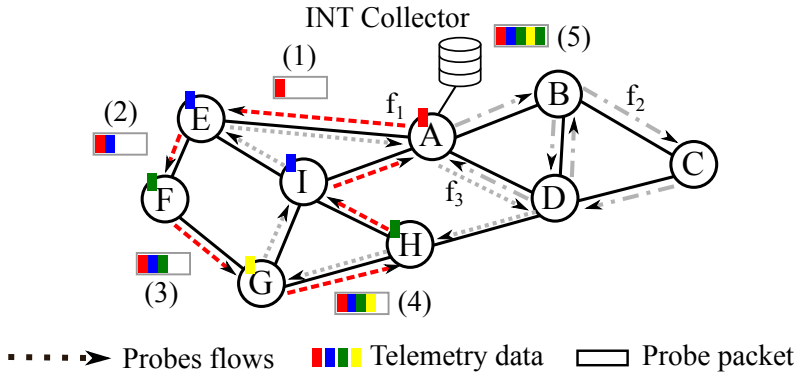


Figure 2 – Example of a solution for the Probing Planning problem, illustrating a snapshot where probing packets (f_1, f_2, f_3) collect telemetry data from selected network devices.

1.2 Research Problem

Despite current efforts towards near real-time in-band network telemetry, the coordination of INT probing packets to collect network information efficiently in terms of time and space usage is still full of gaps and challenges. For example, little has yet been done to provide fault-tolerant orchestration of INT mechanisms in programmable network infrastructures. The first attempts (LIU et al., 2018; PAN et al., 2019) to tackle this problem contributed with initial steps but suffer from (i) uncoordinated probing packet generated and (ii) neglected capacity constraints. Pan et al. (PAN et al., 2019) utilizes a straightforward Depth-First Search (DFS)-like a strategy to generate *Euler Circuits*, leading to dozens of probing packets that increase the monitoring overhead towards the INT collectors. Furthermore, relaxing capacity constraints from probe packets make the problem easier to solve – but unrealistic from an operational perspective. Although existing solutions have either focused on INT probes to monitor link connectivity (e.g., (LIU et al., 2018; PAN et al., 2019)) or focused on the collection itself of INT data (e.g. (MARQUES et al., 2019; HOHEMBERGER et al., 2019)), they still miss how to jointly optimize the way to collect telemetry data and cover network links while being fault-tolerant.

1.3 Objectives and Contributions

The proposed study has two main goals: (*i*) formalize the Probe Planning for In-Band Network Telemetry problem; and (*ii*) design efficient and scalable algorithmic methods to timely compute quality-wise solutions.

Those goals unfold into a set of contributions of this work, described below. We theoretically formalize P^{2}INT as a Mixed-Integer Linear Programming (MILP) model and propose a math-heuristic to tackle the problem. The model consists of a generalization of two well-known optimization problems – namely, Capacitated Arc Routing problem and Bin Packing problem (GAREY; JOHNSON, 1979) and, therefore, it is an Nondetermin-

istic Polynomial Time (NP)-hard problem. P^{2}INT coordinates how to generate and route probe packets to ensure that all links are visited while all required INT data is collected. In summary, it works as follows. First, we compute a feasible solution to the problem. Then, we leverage Variable Neighborhood Search (VNS) to iterate probing cycle sets and merge them while prioritized subsets with higher potential for improvement. Also, we theoretically formalize FP^{2}INT and propose a polynomial-time heuristic that wisely finds a high-quality solution with a mechanism namely *Patcher*. In the event of faulty forwarding devices, *Patcher* efficiently rebuild and fix monitoring cycles by applying "patches" to ensure that all links are visited and the required INT data is collected correctly. It only rebuilds part of the solution affected by faulty nodes (i.e., network devices). In summary, *Patcher* aims to maintain all links covered and telemetry collected from available network devices.

This work that tackles orchestrating INT probes across entire network infrastructures – to cover all links and collect all telemetry items (i.e., P^{2}INT) while it enables the network infrastructure to be fault-tolerant – with regards to device failures and to keep INT applications alive (i.e., FP^{2}INT).

1.4 Outline

The remainder of this work is as follows. In Chapter 2, we overview the current literature on traditional network monitoring, as well as for INT monitoring, and discuss the state-of-the-art efforts towards network monitoring on programmable networks. In Chapter 3, we introduce P^{2}INT formally presenting a MILP model and a math-heuristic approach to the problem. Next, Chapter 4 introduces the formalization of FP^{2}INT and *Patcher*. Finally, Chapter 5 summarizes the topics covered, our results, and planned future work.

2 BACKGROUND AND RELATED WORK

In this chapter, we overview recent advances in network programmability, from the control and data plane aspects. Then, we review the most prominent research studies in this domain.

2.1 Network Programmability

The increasing number of network services (e.g., video streaming) and the complexity behind network infrastructures hinder the ability of network providers to deliver the best out of network application requirements (e.g., SLAs). The inability to properly manage networks at fine-grained level information is due to the difficulty to change its structure, inhibiting efforts to solve problems and innovate network management. There are two main reasons for that: (i) the control and data plane are physically coupled and (ii) there is a vast amount of different protocols running on devices.

With full network programmability, greater visibility can be achieved to guarantee different requirements of services running on the network infrastructure i.e., to collect and gather more fine-grained information (see Table 2). Those pieces of information are inaccessible in traditional switches and assist the network operator at detecting different network problems (e.g., micro-burst, heavy hitters) with higher accuracy than most traditional methods, which relies on sampling (e.g., NetFlow (CLAISE et al., 2004), SFlow (PHAAL; PANCHEN; MCKEE, 2001)) or *polling* (SNMP (CASE M. FEDOR, 1989)). The aforementioned methods are not suitable for different reasons (simultaneously): (i) network coverage is not guaranteed, since few telemetry targets result may evade the supervision of network operators, missing key events to detect problems and anomalies; (ii) scalability is essential for growing networks, such as large data centers running dozens of different services with its heterogeneous requirements (e.g., minimum bandwidth, maximum Round Trip Time (RTT)). This happens due to the control logic being linked to forwarding devices (routers, switches), narrowing operators' actions that are usually pragmatic standard-driven using existing protocols. Problems like these can be easily tackled with Sofware-Defined Networks (SDN) (Kreutz et al., 2015). SDN is a paradigm that proposes to decouple the control logic from forwarding devices (data plane) – allowing flexible operation and decisions over the network infrastructure.

2.1.1 Control Plane Programmability

SDN has emerged as an alternative for easing network programmability by decoupling the control plane from the data plane, i.e., physically separating the device's control logic from its forwarding engine. By decoupling the control logic from the network devices, SDN allows many different network-management tasks to be more tactile while increasing network reliability and performance for a variety of services. This de-

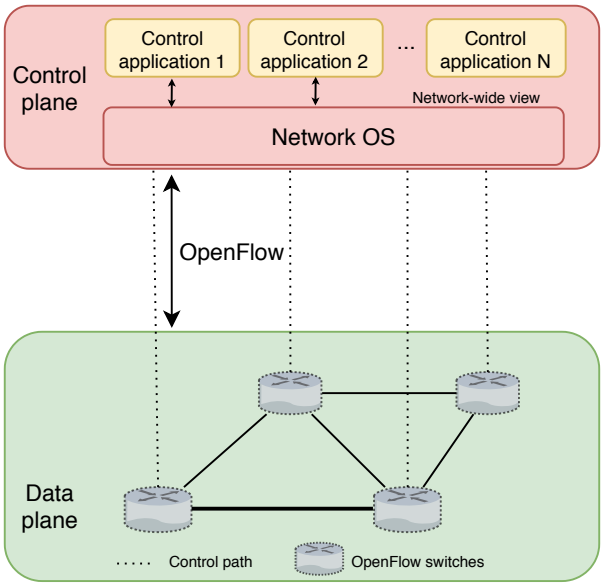


Figure 3 – SDN architecture overview.

coupling lowered network service providers' timescale to develop and deploy new network services. An example of this separation is OpenFlow. Recently, OpenFlow has been widely adopted by the industry as an open interface that enables the intercommunication between control and data plane. OpenFlow allows network operators to instruct how network devices behave by modifying flow tables, which enables switches to assume different network functions such as switches, routers, firewalls – to name a few.

OpenFlow came at an opportune time when the industry was urgent for technologies embracing more pragmatism in terms of network programmability, allowing to tackle different challenging network events for troubleshooting (e.g., heavy hitters). OpenFlow enables network hardware and software to evolve separately and eases the path for the replacement of proprietary hardware and firmware by open source Network Operating Systems (NOS) (e.g., (CASADO et al., 2006; CASADO et al., 2007; GUDE et al., 2008; BERDE et al., 2014)).

Figure 3 shows the SDN architecture overview. At the bottom (represented in green) there is the data plane. The data plane is a southbound interface composed of OpenFlow switches (or forwarding devices) responsible only for processing and forwarding data packets. In its turn, the control plane (represented in red) represents the "intelligence" of the SDN and offers a logically- centralized control. It is the northbound interface that allows operators to instruct on how to manage the incoming data, where a network OS (e.g., NOX (GUDE et al., 2008)) handles the requests from control applications and translates them onto OpenFlow directives. The interaction between both planes is managed by OpenFlow protocol, responsible for the insertion of instructions that match a set of headers to perform different actions (e.g., to drop, or to forward network packets) by updating flow tables in switches, allowing the device to behave like a switch, router, firewall, or something in between.

However, OpenFlow is limited by the data plane-supported features. For example, OpenFlow 1.0 supports 12 header fields and that has grown to 42 in OpenFlow 1.5.1. For instance, suppose we need to deploy a new network link protocol. In this case, even having access to control plane programmability and OpenFlow interface, it is not possible to use OpenFlow to instrument forwarding devices that do not support this new protocol. In other words, as OpenFlow operates manipulating forwarding tables (e.g., by matching on a given header), the hardware/software forwarding devices must offer support for it. Table 1 summarizes some OpenFlow actions that can be performed on packets.

Table 1 – OpenFlow action set.

Action	**Description**
copy TTL inwards	apply copy TTL inward actions to the packet.
pop	apply all tag pop actions to the packet.
push-MPLS	apply MPLS tag push action to the packet.
push-PBB	apply PBB tag push action to the packet.
push-VLAN	apply VLAN tag push action to the packet.
copy TTL outwards	apply copy TTL outwards action to the packet.
decrement TTL	apply decrement TTL action to the packet.
set	apply all set-field action to the packet.
qos	apply all QoS actions, such as a meter, and set_queue to the packet.
group	if a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list
output	if no group actions are specified, forward the packet on the port specified by the output action
priority	Set packet priority.

2.1.2 Data Plane Programmability

More recently, network programming languages such as P4 and Protocol Oblivious Forwarding (POF) eased data plane programmability by allowing network programmers to fully specify the internal pipeline process of forwarding devices. P4 has emerged recently as an alternative to low-level data plane programming (e.g., C-like or Data Plane Development Kit (DPDK)-based programming). P4 allows programmers to specify protocols, parsers, and how incoming packets are processed by forwarding devices in a target-independent way using different solutions (e.g., software forwarding devices, Field Programmable Gate Array (FPGA)s, Neural Processing Unit (NPU)s).

The P4 workflow process to program a target is summarized in Figure 4. First, a P4 code, an architecture model, and a P4 compiler are both provided by the user. P4 programmers write programs for a specific architecture, which defines a set of programmable components at the target as well as their external data plane interfaces. When compiling a set of P4 programs, two artifacts are produced: (i) a data plane configuration that

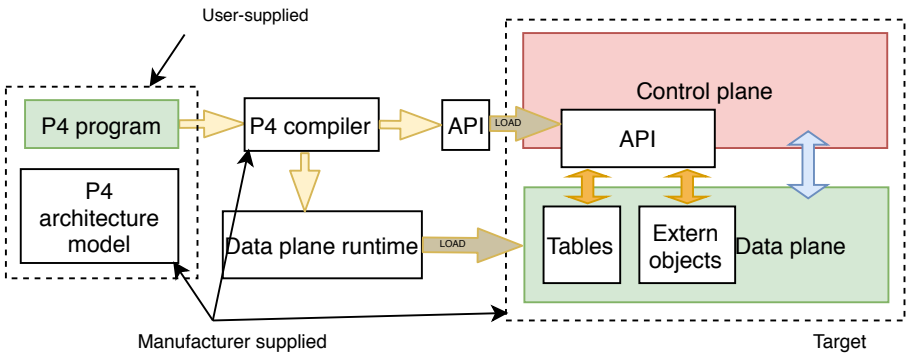


Figure 4 – Programming a target with P4.

implements the forwarding logic described in the input program and (ii) an Application Programming Interface (API) for managing the state of the data plane objects from the control plane. The control plane communicates with the data plane using the same channel, but tables and other objects are not fixed anymore, since they are defined by the P4 program.

P4 was initially designed to program switches/forwarding devices. However, its scope has been broadened to cover a large variety of devices (e.g., FPGAs, Network Interface Card (NIC)s, Application-Specific Integrated Circuit (ASIC)s) – as long as the constructs can be implemented on all of those platforms with minimal resource usage. In Figure 5, we illustrate the programmable forwarding pipeline model. Incoming packets are forwarded via a programmable parser and are followed by multiple stages of match-action, arranged in series, parallel, or a combination of both. The model is derived from OpenFlow, but assumes some generalizations: (i) OpenFlow assumes a fixed parser, whereas P4 supports a programmable parser to allow new headers to be defined; (ii) OpenFlow assumes the match + action stages are in series, whereas P4 supports either in parallel or in series stages; (iii) P4 model assumes actions to be composed of protocol-independent primitives supported by the switch. Hence, P4 programs are target-independent. A compiler can map a variety of different forwarding devices (i.e., in the same architecture), ranging from relatively slow software switches to the fastest ASIC-based switches.

2.1.3 Network Monitoring

2.1.3.1 Traditional Network Monitoring vs INT monitoring

Network Monitoring has the purpose to enable proper management operations, providing network operators information about the network as the basis for the management decisions on traffic engineering and anomaly detection (e.g., micro-bursting, heavy hitters). Traditional monitoring, in the operational model (ZIMMERMANN, 1980), may be described by a series of steps as in SNMP explained below.

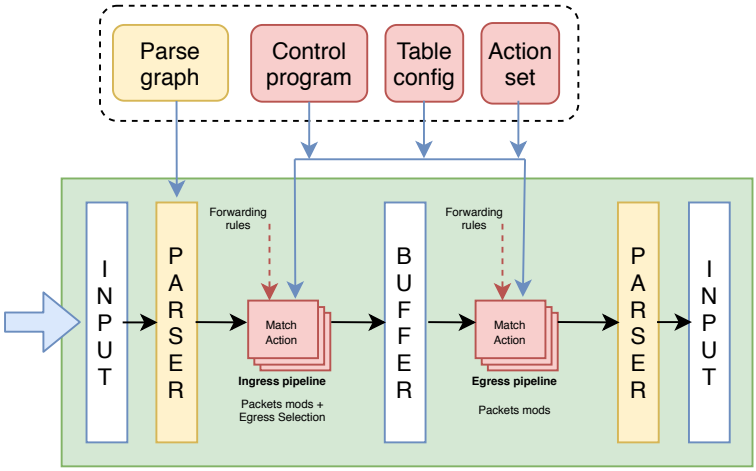
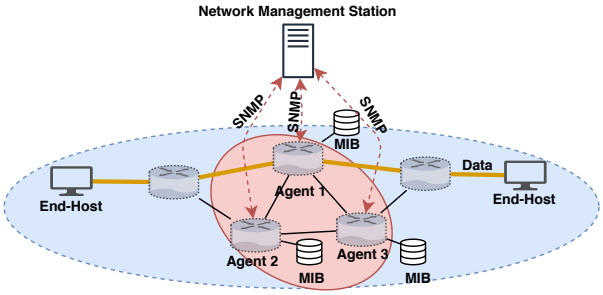


Figure 5 – P4 abstract forwarding model.

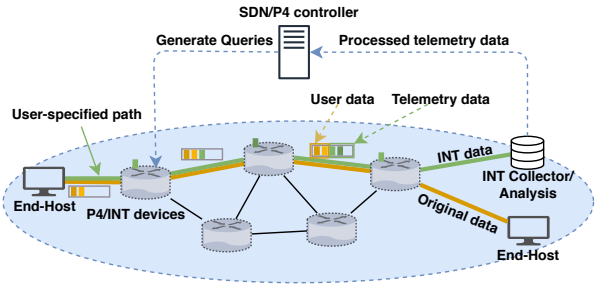
SNMP consists mainly of a collection of network management stations and a subset of agent elements (see Figure 6a). To identify each monitoring element, there is an Object Identifier (OID) for each device. Identified devices may read or set SNMP messages stored in its Management Information Base (MIB), which defines object properties hierarchically and keeps track of its operations. The elements communicate by exchanging messages (polling) through User Datagram Protocol (UDP) packets using SNMP protocol for further perform data measurement and data processing. The process works as follows. First, data is collected from target devices at a certain rate, constituting the collection phase. Right after, the collected data is aggregated and preprocessed into some statistical format. Finally, the itemized data is transmitted to a station where further analysis is performed, generating statistics that help identify particular events. Also, the results may be exported. For example, (ALEXANDROV; KAZYMOV; PROKOSHIN, 2018) uses a data viewer called Grafana (GRAFANA, 2020) to monitor short-term and long-term traffic statistics.

Accurately monitor different events on the network allows network operators to solve network problems faster rather than spending a longer period looking for the problem itself. Yet, traditional monitoring has striven to deliver "near" real-time visibility over the network infrastructure because of insufficient network coverage and scalability. SDN allows for improving how network monitoring performs. Figure 6 illustrates an overview of the two models. The main differences between the traditional and INT model rely on the collection phase and transmission phase. We can schedule (with a global view) which devices the telemetry data is to be collected and at what frequency, with minimal intervention from the operator. Also, open protocols (e.g., OpenFlow) allows operators to defined specific data structures for their needs.

Despite that, there are existing limitations regarding data-plane programmability. SDN only allows accessing coarse-grained data from devices by matching flow tables modified by the control plane. Therefore, it is impossible to access fine-grained internal-state



(a) Traditional monitoring.



(b) INT monitoring.

Figure 6 – Traditional and INT monitoring overview.

information (e.g., data-plane packet processing time) to provide higher visibility of the network state and help identify problems timely with better accuracy.

2.1.3.2 In-band Network Telemetry

Traditional active and passive measurement techniques are either inaccurate or resource-consuming. INT allows access to internal-state information (e.g., packet-processing time) at the data plane. Network monitoring generally performs in an in-band fashion – i.e., it forwards using the same links as traffic packets. Also, INT monitors in one of two ways. In the first (i.e., active measurement), telemetry data is carried in network packets of the active network by embedding an instruction header to user packets and directing network devices to add the requested data into the packet along its route. By reaching its final destiny, as in Figure 6b, all the telemetry data collected are sent to a collector (INT sink) for further analysis where statistical information is extracted, whereas the user data is sent to its original destination. The processed telemetry data is then sent to an SDN controller where compliance requirements are checked and decisions are translated into new queries on the data plane's devices. In the second, the process is similar. However, specially-crafted probe packets are used to only carry telemetry information on what telemetry data is to be collected. Probe packets are similar to regular network packets. The difference is they are artificially created. These packets can follow a specific format (LUCKIE; MCGREGOR; BRAUN, 2001) or be encapsulated in well-known protocols such as UDP, Transmission Control Protocol (TCP), or Network Service Header

(NSH). One of the advantages of this approach is that it has more space for collecting information since there is no user data embedded. However, an excessive number of probes can also generate overhead in the pipeline of the routing devices. In this case, the collection of telemetry data is only limited by the packet Maximum Transmission Unit (MTU) – since there is no traffic data embedded in it.

The INT model is currently realized by high-level programming languages such as P4. P4 language allows by default to access a wide range of telemetry data from forwarding device architectures. For instance, the v1model architectural switch (BAS C. CASCONE, 2020) describes the code specifications of a simple switch. We can find in this architectural reference which metadata fields (or telemetry files) are available to be collected. Examples of metadata include packet-queuing size, queuing timestamp, and queuing depth (see Table 2 for a complete list of currently available metadata). Observe that custom-made metadata can be designed by P4 programmers (for instance, heavy-hitters data structures). It is also possible to count packets and bytes traversing the device and mark certain packets to be dropped at the end of the processing, all being performed in the data plane.

As an example, consider we want to collect telemetry information to monitor network infrastructure. Regardless of the approach (e.g., using actives flows, probe packets), by using P4 we specify each packet to collect a variety of internal-state variables (e.g., switch IDs, queue length, process time) along its path, allowing a more fine-grained level of information available to the operator. By doing this, we can detect, e.g., micro-bursts, which were previously unachievable with traditional monitoring.

2.2 Related Work

Next, we discuss the most prominent studies related to in-band network telemetry. Recent advances in forwarding devices (e.g. Cisco (2018), Arista (2018), Juniper (2018), Huawei (2018)) have enabled to continuously push telemetry information (i.e. via streaming) to data collectors – known as Model-Driven Telemetry (MDT) (WU J. STRASSNER; ZHANG., 2016). In this context, Putina et al. (PUTINA et al., 2018) proposed a mechanism for real-time detection of Border Gateway Protocol (BGP) anomalies, relying on machine-learning techniques and Model-Driven Telemetry (MDT)-based telemetry data streaming. Other studies have focused on the concept of in-band network telemetry (INT) (BOSSHART et al., 2014). INT enables the inclusion of "telemetry instructions" into available fields of packet headers. These fields instruct INT-capable devices what telemetry data they should collect and write into the respective packets. Kim et al. (KIM et al., 2015) show that it is possible, for instance, to discover which switches are causing HTTP latency spikes in a network. In their evaluation, they considered an emulated scenario (using Mininet (LANTZ; HELLER; MCKEOWN, 2010)) and took advantage of the TCP options field to push INT instructions to collect queue time spent in the switch.

Table 2 – Primary metadata in v1model switch.

Field	Notes	Size (bits)
ingress_port	The port on which the packet arrived.	9 bits
packet_length	The number of bytes in the packet.	32 bits
egress_spec	Specification of an egress. Maybe a physical port, a logical interface, or a multicast group.	9 bits
egress_port	The physical port to which this packet instance is committed.	9 bits
enq_timestamp	Queue timestamp at entry queue.	32 bits
enq_qdepth	Queue depth at entry queue.	19 bits
deq_timedelta	Queue depth at the packet dequeue time.	32 bits
deq_qdepth	Queue depth at exit queue.	19 bits
instance_type	Represents the type of instance of the packet (normal, ingress clone, egress clone, recirculated, resubmitted). The representation of this data is target-specific.	32 bits
ingress_global_timestamp	Ingress global timestamp.	48 bits
egress_global_timestamp	Egress global timestamp.	48 bits
mcast_grp	Multicast group id (key for the mcast replication table.	16 bits
egress_rid	Indicates that a verify_checksum() method has failed.	16 bits
checksum_error	1 if a checksum error was found, 0 otherwise.	1 bit
priority	Set packet priority.	3 bits

In the last years, several studies to enhance network visibility have been proposed. These approaches rely either on a programmable control plane (e.g., using OpenFlow) or, more recently, on a programmable data plane to tackle several monitoring tasks within different scenarios (e.g., ISPs or Data Center Network (DCN)s).

2.2.1 Programmable Control Plane

Handigol et al. (HANDIGOL et al., 2014) designed NetSight, a platform that uses a *postcard* approach. It introduces the concept of "packet histories", i.e. information of each switch state and header modifications along the routes. This is done by the use of *postcards* in an out-of-band or in-band fashion. A protocol intercepts and stores SDN-controller messages to generate a *postcard* for each packet matching the rule. The *postcards* are then used by applications such as a debugger or a packet history logger

to help detect problems such as loops and congested links. Everflow (ZHU et al., 2015) extended INT concept by exploring the "match-and-mirror" functionality of commodity switches. Everflow uses the INT concept to filter packets that satisfy given patterns (i.e. *matching*) and send (*mirroring*) them to multiple data analyzers, which then can send "guided probes" to investigate potential faults.

Tu et al. (Van Tu; Hyun; Hong, 2017) innovates by introducing Open Network System Operating System (ONOS), an INT-based monitoring system. ONOS has distributed SDN controllers that create flow-level rules "match-action" fashion to instrument INT packets to collect telemetry data at the packet level. The information analyzed can then be displayed at, e.g. a web service and help identify problems in the infrastructure. Gupta et al. (GUPTA et al., 2018) designed SONATA, a high-level interface to express telemetry queries. Based on programmable data plane constraints, monitoring queries are partitioned and processed in multiple devices – ensuring therefore that monitoring queries and the packet forwarding still operate at line rate for high traffic volumes and rates. Other studies (BASAT et al., 2017; SIVARAMAN et al., 2017) have proposed to execute specific telemetry operations (e.g. heavy hitters identification) directly in the data plane. However, telemetry operations are limited by the available capabilities (e.g. memory) in forwarding devices.

(ADRICHEM; DOERR; KUIPERS, 2014; YU et al., 2015; RAMANATHAN; KANZA; KRISHNAMURTHY, 2018) focused efforts on probe usage for network monitoring. OpenNetMon (ADRICHEM; DOERR; KUIPERS, 2014) continuously monitors per-flow metrics, like throughput, delay and packet loss based on predefined link-destination pairs. Similarly, SLAM (YU et al., 2015) uses the time of arrival of the first and last switches from a path to estimate the delay between links. In turn, SDProber (RAMANATHAN; KANZA; KRISHNAMURTHY, 2018) proactively generates probes to inspect the links to identify delays and congestion. The probes are routed using a random walk approach to traverse (with different rates) each link in the network with different frequencies adjusted iteratively.

2.2.2 Programmable Data Plane

Mazières et al. (JEYAKUMAR et al., 2014) introduced the seminal work on in-band telemetry. They proposed the concept of Tiny-Packet Program (TPP) by modifying Ethernet packets with a uniquely identifiable header that contains a set of instructions (at most 5) allowing end-hosts to actively query networks' internal state for DCNs. However, the set of instructions is not expressive enough to implement all data plane network tasks (e.g., per-packet scheduling).

PathDump (TAMMANA; AGARWAL; LEE, 2016) is a mechanism designed to identify and debug anomalous behaviors in programmable network infrastructure. The approach is based on the route taken by network packets and on the subsequent analysis

of it. For that, PathDump keeps track of the packet's route, employing INT instruction in the forwarding devices. In turn, SwitchPointer (TAMMANA; AGARWAL; LEE, 2018) further advances that approach, proposing to collect end-host telemetry information to enhance the debugging capabilities – in addition to in-network telemetry information. The telemetry data collected in the infrastructure is stored internally in the routing devices, constituting a distributed "storage service" of monitoring information. This approach reduces the cost and potential impact of constantly transmitting telemetry data to analyzers.

Other studies (MESTRES et al., 2017; HYUN; HONG, 2017; TU et al., 2018) leverages the advances of both the control plane and the data plane (e.g., P4) to design self-learning network architectures to control and operate networks. (MESTRES et al., 2017) redefines Knowledge-Defined Network (KDN) (CLARK et al., 2003) and suggests a knowledge plane that adopts AI and a cognitive system. Hyun and Hong (HYUN; HONG, 2017) proposes a model that consists of four planes that act together: (i) the data plane where INT metadata is generated, extracted, and transmitted to (ii) the management plane, which is responsible to store and aggregate the collected data for further analysis in the (iii) knowledge plane. In (iii), the gathered data serve as the input to a machine learning algorithm and is converted to requirements that aid Software-Defined Network (SDN) controllers on (iv) the control plane to deploy INT requirements into programmable switches. Similarly, INTCollector (TU et al., 2018) proposes a telemetry system where INT probes help the collector identify called "events" on the data plane by the usage of thresholds to achieve dynamic granularity. The events and recent data are stored in a database where the SDN controller (e.g., ONOS (Van Tu; Hyun; Hong, 2017)) queries information to learn about the network behavior in an automated fashion.

SpeedLight (YASEEN; SONCHACK; LIU, 2018) presents the design and implementation of a synchronized network snapshot protocol to collect network measurements in the data plane. It takes into account the data plane can perform extremely fine-grained in-band processing of network traffic, but it is limited by computation and resources available – while the control plane has opposite trade-offs. SpeedLight proposes to split the responsibilities to take snapshots into the data plane and the control plane processing units at switches/routers and snapshot observers. The process is as follows: (i) a host acts as a snapshot observer that broadcasts a request on which device to take a snapshot of a given metric at a given moment in the future; (ii) the control planes running on each device synchronize using a protocol like Precision Time Protocol (PTP) to initialize the snapshot; (iii) the data plane processes the queries, while the control plane gathers the snapshots and detects its completion). In turn, FlowStalker (CASTANHEIRA; PARIZOTTO; SCHAEFFER-FILHO, 2019) operates entirely on the data plane to avoid communication time bottleneck between the controller and programmable switches to gather telemetry items in a distributed fashion. The network is subdivided into clusters,

based on a metric (e.g., latency time) that enables the information to be gathered as a cluster event without the controller's interference. The monitoring occurs in two phases: (i) proactive phase: counts each incoming packet of a given flow and compares to a low threshold that identifies flow targets; (ii) monitor target flows and collect per-flow (e.g., packet and byte counts) metrics and per-packet (e.g., timestamps) metrics. Besides, the gathering process is only triggered by a high threshold, which avoids communication overhead to the controller. However, the system lacks network query expressiveness due to the existing hardware limitations while considering only a subset of flows to be monitored timely.

Liu et al. (LIU et al., 2018), Pan et al. (PAN et al., 2019), Geng et al. (GENG et al., 2019), Basat et. al (BASAT et al., 2020) and Lin et al. (LIN et al., 2020) have focused on performing network telemetry through active INT-based probing packets. These strategies have relied either on *Euler Circuits* (LIU et al., 2018; PAN et al., 2019) or on actual routing paths (GENG et al., 2019; BASAT et al., 2020) to instrument the forwarding of probes. Liu et al. (LIU et al., 2018) proposed NetVision, an attempt to provide an architectural design to offer network telemetry as a service. NetVision enhance network visibility by dynamically changing the routing policies applied to network flows. (LIU et al., 2018) and (MARQUES et al., 2019) propose two heuristic strategies for collecting telemetry data, namely, *concentrate* and *balance*. The proposed heuristics assign network flows to forwarding devices. Once the assignment is defined, network flows collect telemetry information from forwarding devices. The strategy named *balance* tries to distribute equally the telemetry data over available network flows. On the other hand, the other strategy – named *concentrate* – strives to aggregate telemetry data on a restricted number of flows.

Pan et al. (PAN et al., 2019) were the first attempt to tackle this problem. However, these solutions have (i) generated uncoordinated probing packets, and (ii) neglected existing capacity constraints. For example, Pan et al. (PAN et al., 2019) utilize a straightforward DFS-like strategy to generate *Euler Circuits*, leading to dozens of probing packets. This increases the overhead to transmit probing packets to INT collectors as they need to get to their respective sink. Further, relaxing capacity constraints of probing packets make the problem easier to solve in practice – but unrealistic from the operational point of view. Similarly, Lin et al. (LIN et al., 2020) present NetView, a telemetry system that supports different telemetry frequencies, monitoring each device using active probes. Each probe contains a (i) forwarding stack for each hop information and a (ii) telemetry stack that gathers telemetry data along the path of the probe. Users can request telemetry queries through an API where a telemetry coordinator is responsible for generating probes considering different frequencies of data collection and prioritizing high-frequency query cluster demands to the lower ones.

Geng et al. (GENG et al., 2019) and Zhou et al. (ZHOU et al., 2020) leverage

programmable switches and NICs. Geng et al. (GENG et al., 2019) propose SIMON, a measurement system that reconstructs the network state by the collection of key network state variables such as queuing times, link utilization, and queue composition. SIMON takes advantage of existing NICs being able to retrieve data on an edge-based approach. A mesh of probes reconstructs DCN networks by covering the paths on a per-packet or per-flow basis. The reconstruction is based on network tomography and uses the LASSO (TIBSHIRANI, 1996) inference algorithm that timely feeds a multi-layered neural network and provides high accuracy to detect different problems. However, due to network tomography limitations, SIMON limits itself to only operate in DCN networks where it has full knowledge about the topology. In turn, NetSeer (ZHOU et al., 2020) selects packets that experience flow events, minimize false-positives (duplication of reported flow events), and aggregate sequential event (e.g., congestion) packets from flow into a single flow event to aid on the location of anomalies (e.g., packet drops) in the network.

Basat et al. (BASAT et al., 2020) provide PINT, an in-band telemetry framework for High-Performance Computing Cluster (HPCC) networks, that bounds the amount of information added to each packet. PINT considers most applications do not require perfect telemetry information and allows the encoding of requested query data per packet as low as one bit on multiple packets. A query engine decides on an execution plan the probability of running each query set on packets along its path. Instead of collecting all items (per-packet per-switch), PINT aggregates data onto packets successfully reduces overhead by the use of switch IDs instead of INT meta header in packets which reduces the processing time at switches. However, PINT does not necessarily cover the whole topology. Besides, it provides an only-aggregated view of the network state e.g., tracing flows needs significantly fewer packets, but small flows may consist of a single packet, which prevents from splitting the query telemetry data.

Most of the recent work (CHEN; HAO; GLOVER, 2016; TIRKOLAEE; MAHDAVI; ESFAHANI, 2018; ARAKAKI; USBERTI, 2019) do not consider there are (i) one or more collectors (i.e., different depots) disposed on the network infrastructure, (ii) items to be collected at each device (i.e., node constraints), and (iii) links to be satisfied (i.e., edge constraints) concomitantly. (ARAKAKI; USBERTI, 2019) proposes a path-scanning heuristic PS-Ellipse, for the known Capacitated Arc Routing Problem (CARP). It constructs a feasible solution from an efficiency rule that selects the most promising edges to serve next. In a similar way, (CHEN; HAO; GLOVER, 2016), (TIRKOLAEE; MAHDAVI; ESFAHANI, 2018) present different meta-heuristic approaches to tackle the problem. (CHEN; HAO; GLOVER, 2016) proposes a hybrid metaheuristic approach (HMA-CARP) which incorporates local refinement and randomized tabu-search procedures to the solution, while (TIRKOLAEE; MAHDAVI; ESFAHANI, 2018) proposes a hybrid simulated annealing based on a heuristic algorithm for Periodic Capacitated Arc Problem (PCARP) on urban waste collection. This version of the problem differs from

CARP, mainly by (i) different vehicle (probe) capacity and (ii) maximum amount of time to generate valid trips. In turn, Ma et. al (MA et al., 2014) investigate the problem of identifying individual link metrics in a network from end-to-end path measurements with a minimum number of monitor placements, under the assumption that link metrics are additive and constant i.e., the combined metric over multiple links is the sum of individual link metrics. Under an extensive formulation, it is shown the minimum number of monitors necessary for satisfying link metrics is at least k (k=3, but typically more), irrespective of the network topology. However, the Minimum Monitor Placement (MMP) algorithm only solves optimally (linear time) *iff* the assumed paths of the packets are routed in a *cycle-free* fashion i.e., monitors can direct measurement packets to selected paths as long as they do not contain cycles.

As can be observed, current research efforts related to the *in-band* telemetry are still restricted to mechanisms that mostly utilize collected telemetry data for new monitoring solutions (JEYAKUMAR et al., 2014; ZHU et al., 2015; TAMMANA; AGARWAL; LEE, 2016; TAMMANA; AGARWAL; LEE, 2018; CHEN et al., 2019). The studies introduced by (MARQUES et al., 2019) and (LIU et al., 2018) represent the first steps towards the orchestrating of in-band network telemetry. However, they have focused on the orchestration of static and *offline* scenarios. Furthermore, none of these works consider, at a design level, that devices may fail for a multitude of reasons (e.g., power failure, misconfiguration), and thus, may compromise the network telemetry system, in addition to not explicitly consider flow capacity constraints. By not considering capacity constraints, the problem tackled by (MARQUES et al., 2019) becomes computationally tractable. However, it limits the operation of the proposed strategies in a production environment. As a first step in the direction of dynamic and online orchestration, it advances the state-of-the-art by dynamically modeling the orchestration problem of in-band telemetry items in a dynamic way and with the aid of machine learning models (e.g., (HOHEMBERGER et al., 2019), (GENG et al., 2019)). Hohemberger et al. (HOHEMBERGER et al., 2019) is the first attempt to coordinately collect telemetry items in real-time. They design a machine-learning-based model and formalize the problem of collection to satisfy spatial and temporal requirements i.e., consider items must be collected from specific devices and at a certain rate, respectively to properly feed machine-learning applications on top of the network to detect anomalies (e.g., Distributed Denial of Service (DDoS)). Similarly, Pan et at. (PAN et al., 2019) utilizes *Euler Circuits* and DFS-like strategies to orchestrate probing packets across the network. However, these works construct a static solution and do not consider devices may fail.

As will be shown later, our approach can outperform state-of-the-art, coming up with feasible, high-quality solutions for larger scenarios in a timely fashion. To the best of our knowledge, our work is the first to simultaneously (i) collect all telemetry data from devices, considering there may be one or more collectors, (ii) verify every link at the

network infrastructure, and (iii) fault-tolerant with regards network devices.

3 OPTIMAL AND SCALABLE PROBE PLANNING FOR IN-BAND NETWORK TELEMETRY

In this section, we present the Probe Planning for In-Band Network Telemetry (P^2INT) problem. First, we show a brief introduction to the problem. Then, an optimization model formally defines the problem and its constraints. Finally, we show a near-optimal approach to the problem and evaluate it against the current state-of-the-art.

3.1 Problem Overview

The P^2INT problem consists of defining optimized probing cycles to cover network infrastructure, i.e., in terms of telemetry data and network connectivity. It is noteworthy that the complete network coverage, both in terms of links and nodes, enables the assessment of end-to-end metrics based on different composition rules (e.g., multiplicative, additive, and concave) (FILHO et al., 2018). This approach is crucial for operating in large-scale networks such as the 5G device-to-device ecosystem, where path-based measurements are prohibitive due to the massive number of available paths.

The P^2INT problem is not trivially solved. First, probing packets are space-bounded (i.e., w.r.t bytes), and therefore it is infeasible (in most cases) to collect all network telemetry data with a single packet. Second, routing a probing packet is challenging. Probes need to be routed in such a manner that telemetry data requirements are met while avoiding extra overheads on production network traffic (e.g., excessive generation of probing cycles).

Figure 2 illustrates a network infrastructure with nine programmable forwarding devices (ranging from A to I), each having exactly one equal-sized telemetry data (represented by colored rectangles). These telemetry data represent data planes' internal states (e.g., queue occupancy or processing time), which are used by specialized monitoring applications (HOHEMBERGER et al., 2019) (e.g., DDoS detection). In the example, probing packets are limited to collect at most five telemetry data. There exists a set of active probing cycles (i.e., f_1, f_2, and f_3) which are responsible for continuously (i) collecting telemetry data and (ii) checking network connectivity. Probing cycles f_1, f_2, and f_3 are routed and instrumented to collect a given subset of telemetry data. For instance, probing cycle f_1 collect telemetry data from forwarding devices A to H, while probing cycle f_3 from devices D and I. Observe that all network links are covered by at least one probing cycle.

3.2 Proposed Model

The proposed optimization model considers a physical network infrastructure $G = (D, L)$ and a set of telemetry items V. Set D in network G represents programmable

forwarding devices $D = \{1, ..., |D|\}$, while set L consists of unidirectional links interconnecting pair of devices $(i, j) \in (D \times D)$. Similarly to the literature (MARQUES et al., 2019; HOHEMBERGER et al., 2019), we consider that there exists a set of telemetry items V available. Each forwarding device $i \in D$ is able to embed a subset of items $V_i \subseteq V$ into a probing packet. Each telemetry item $v \in V$ has its size defined by function $S : V \to \mathbb{N}^+$.

We consider there is at most P probing cycles (i.e., $P = \{1, 2, ..., |P|\}$) to collect telemetry items from forwarding devices D. Packets in a probing cycle are encapsulated in a forwarding protocol, and therefore the amount of available space to embed telemetry items in packets is bounded by a constant, defined by function $U : P \to \mathbb{N}^+$ (e.g., $U(p)$ lower or equal to the MTU data link). Probing cycles P are routed within the network infrastructure G – i.e., the packet is generated in a given source device, is routed through a subset of devices, and returns to its origin. We denote the cycle taken by the probing $p \in P$ as function $\mathcal{C} : P \to \{D_1 \times ... \times D_{|D|}\}$. Probing cycles $p \in P$ can collect telemetry items from forwarding devices $i \in \mathcal{C}(p)$. The set of telemetry items collected by probing cycle $p \in P$ is represented by pairs $(i, v) : i \in D, v \in V_i$ and is given by the function $\mathcal{T} : P \to \{D \times V\}$. A feasible cycle satisfy the upper-bound $U(p)$, that is $\sum_{i \in \mathcal{C}(p)} \sum_{v \in V_i : (i,v) \in \mathcal{T}(p)} S(v) \leq U(p)$. Observe that a given cycle $p \in P$ can visit a forwarding device $i \in \mathcal{C}(p)$ and not necessarily collect the set of telemetry items associated. We denote the origin (starting/ending device) of each cycle $p \in P$ as function $O : P \to D$. Therefore, our model is generic to consider single- and multi-source probing cycle scenarios (i.e., cycles might start at different forwarding devices).

Given the problem input, the optimization problem seeks a feasible solution that minimizes the number of generated probing cycles, while visiting all network links and collecting the required telemetry data. The model output is denoted by a 3-tuple $\chi = \{Z, X, Y\}$. Variables from $Z = \{ z_{p,v,i}, \forall p \in P, v \in V, i \in D\}$ indicate that a forwarding device i embed telemetry item v into a probing packet from cycle p. Variables from $X = \{ x_{p,i,j}, \forall p \in P, (i, j) \in L\}$ indicate that network link $(i, j) \in L$ is used to route probing cycle $p \in P$. Last, variable $Y = \{ y_p, \forall p \in P\}$ is used to keep track of probing cycles used by the solution. Other auxiliary variable sets $f_{p,i,j}$ and $b_{i,p}$ are used to ensure cycle connectivity and sub-tour elimination. Next, we describe the MILP formulation for this problem.

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

Subject to:

$$\sum_{p \in P} z_{p,v,i} = 1 \qquad \forall i \in D, v \in V_i \tag{3.2}$$

$$z_{p,v,i} \leq \sum_{j \in D} x_{p,j,i} \qquad \forall p \in P, i \in D, v \in V_i \tag{3.3}$$

$$z_{p,v,i} + x_{p,i,j} \leq \alpha \cdot y_p \qquad \forall p \in P, (i,j) \in L, v \in V_i \quad (3.4)$$

$$\sum_{j \in D} x_{p,i,j} - \sum_{j \in D} x_{p,j,i} = 0 \qquad \forall p \in P, i \in D \quad (3.5)$$

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

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

$$\sum_{j \in D} x_{p,j,i} + \sum_{j \in D} x_{p,i,j} \leq \gamma \cdot b_{i,p} \qquad \forall p \in P, i \in D \quad (3.8)$$

$$\sum_{j \in D} f_{p,i,j} - \sum_{j \in D} f_{p,j,i} = -1 \cdot b_{i,p} \qquad \forall i \in (D - O_p), p \in P \quad (3.9)$$

$$f_{p,i,j} \leq M \cdot x_{p,i,j} \qquad \forall p \in P, (i,j) \in L \quad (3.10)$$

$$z_{p,v,i} \in \{0,1\} \qquad \forall p \in P, v \in V_i, i \in D \quad (3.11)$$

$$y_p \in \{0,1\} \qquad \forall p \in P \quad (3.12)$$

$$x_{p,i,j} \geq 0 \qquad \forall p \in P, v \in V_i, i \in D \quad (3.13)$$

$$f_{p,i,j} \geq 0 \qquad \forall p \in P, (i,j) \in L \quad (3.14)$$

$$b_{i,p} \geq 0 \qquad \forall p \in P, i \in D \quad (3.15)$$

Constraint set (3.2) ensures that generated probing cycles collect the required network telemetry data. Constraint set (3.3) ensures that if telemetry item v is collected from forwarding device i, then there should have a probe being routed through i. Constraint set (3.4) accounts for the number of probing cycles in use. Constraint set (3.5) ensures flow conservation on probing cycles. In other words, they generate probing cycles without ramification or self-loops. In turn, constraint set (3.6) guarantees a probing cycle covers at least one link direction. Constraint set (3.7) ensures that the available capacity $U(p)$ is not violated either by the telemetry items collected or by the network links covered. Constraint sets (3.8), (3.9), and (3.10) are the well-known sub-tour elimination constraints, ensuring that generated probing cycles are strongly connected (GOLDEN; WONG, 1981). Last, constraint sets (3.11)–(3.15) define the domains of output variables. Constant α and γ assume sufficient large values (i.e., $\alpha \geq |P|$ and $\gamma \geq |L|$)).

3.3 A Math-Heuristic Approach to P^{2}INT

To tackle the P^{2}INT complexity and come up with near-optimum solutions, we introduce a fix-and-optimize approach, a mathematical programming-based heuristic (a.k.a., math-heuristic). Fix-and-optimize consists of iteratively choosing which model's variables would be optimized, and which ones would be fixed (hence the name fix-and-optimize). It is important to notice the less variables are fixed, the closer it gets to the optimum as there are there is more room for improvement – for this reason the quality of the solution is "near-optimum".

3.3.1 Overview

To minimize the number of probing cycles in the solution χ, the Fix-and-optimize strategy optimizes just a few probing cycles at once, to merge them by reallocating telemetry data to other cycles. We select only a subset of cycles at a time to have more control over the use of resources (e.g., memory). The model is executed using IBM ILOG CPLEX (see Section 3.4, which uses a variation of a branch-and-bound algorithm. The tree derived by this approach guarantees the optimum, but it can use an excessive amount of memory. Despite this, one of the advantages is being able to iteratively get closer to the solution without exceeding the capabilities of the device. We leverage INT-MD (eMbed Data) (The P4.org Applications Working Group, 2020) mode to perform the INT operations on the probes – e.g., collect and route probe packets. In this mode, both instructions and metadata are embedded into the packets. First, the source node embeds the INT instructions indicating which metadata and hops are to be followed – e.g., this could be achieved with source-routing. Then, both transit nodes – i.e., INT-enabled forwarding devices along the probe path – and the source node aggregate metadata along the path (INT switches). Finally, the sink node (destination INT switch) strips the instructions and aggregated data and (selectively) sends the data to a monitoring system. Algorithm 1 presents an overview of the proposed approach. We first compute a feasible solution χ to the P^{2}INT problem (discussed in Subsection 3.3.2) (line 1). Then, we iteratively select a subset of k probing cycles (lines 5-10), and enumerate the list of variables $x_{p,i,j} \in \mathcal{D} \subseteq X$ related to them (line 11); variables listed in $\mathcal{D}$ will be subject to optimization, while others will remain unchanged (line 12). The decision to make only a subset of cycles free at a time is linked to the computational cost in terms of time and space. When solving the problem in this way, it is possible to gradually improve the initial solution and stop at any point during the execution.

We take advantage of meta-heuristic VNS (Variable Neighborhood Search) to systematically iterate over subsets of probing cycles. Further, we prioritize subsets with higher potential for improvement – i.e., probing cycles that might be merged. (discussed in Subsection 3.3.3). For each subset of probing cycles, we submit its decomposed set of

Algorithm 1 Overview of the fix-and-optimize heuristic.

Input: T_{global}: global time limit, T_{local}: time limit for each solver run, $\mathcal{K}_{init}, \mathcal{K}_{end}$: initial/final neighborhood size, $\mathcal{K}_{inc}$: increment for neighborhood size, $NoImprov_{max}$: max. rounds without improvement

Output: χ: best solution found to the optimization model

1: $\chi \leftarrow$ initial feasible solution
2: **if** a feasible solution does not exist **then** fail **else**
3: $k \leftarrow \mathcal{K}_{init}$
4: **while** T_{global} is not exceeded **and** $k \leq \mathcal{K}_{end}$ **do**
5: $\mathcal{N}_k \leftarrow$ current neighborhood, i.e. tuples of k probing cycles
6: $\mathcal{N}_{k,shr} \leftarrow$ tuples from $\mathcal{N}_k$, whose cycles share devices
7: $\mathcal{N}_{k,any} \leftarrow \mathcal{N}_k \setminus \mathcal{N}_{k,shr}$
8: $NoImprov \leftarrow 0$
9: **while** $\{\mathcal{N}_{k,shr}, \mathcal{N}_{k,any}\} \neq \emptyset$ **and** $NoImprov \leq NoImprov_{max}$ **do**
10: $\mathcal{T} \leftarrow$ next unvisited neighbor (w.r.t Equation 16)
11: $\mathcal{D} \leftarrow$ list of variables $x_{p,i,j}$ from cycles in neighbor $\mathcal{T}$
12: $\chi' \leftarrow$ solution χ optimized by the solver, under time limit T_{local}, and making variables not in $\mathcal{D}$ as fixed
13: **if** χ' is a better solution than χ **then**
14: update χ to reflect solution χ';
15: $k \leftarrow \mathcal{K}_{init}$;
16: **break**
17: **else**
18: $NoImprov \leftarrow NoImprov + 1$
19: **end if**
20: **end while**
21: **if** no improvement was made **then** $k \leftarrow k + \mathcal{N}_{inc}$ **end if**
22: **end while**
23: **return** χ
24: **end if**

variables $\mathcal{D}$ along with χ to a mathematical programming solver. The goal is to obtain a set of values to those variables listed in $\mathcal{D}$, so that a better solution is found. In case there is no improvement, we roll back and pick the probing cycle subset that follows. We run this process iteratively until a better solution is found. Once it happens, we replace the incumbent solution with χ' (line 14), and restart the process (i.e., $k \leftarrow \mathcal{K}_{init}$). This loop continues until we have explored the most promising combinations of available cycles, or T_{global} execution time is exceeded.

3.3.2 Obtaining an initial solution

The first step of Algorithm 1 (line 1) is generating a feasible solution χ. The solution must satisfy all constraints, though not necessarily a high-quality one, in terms of used probing cycles. There are different ways to generate initial solutions to P²INT. We adapted two existing approaches.

The first consists of adapting the `Edge Randomization` (`ER`) heuristic – an approach widely applied in Arc Routing Problems. `ER` starts a probing cycle from a random

unvisited network link. Then, the algorithm randomly chooses an adjacent forwarding device and collects as many network telemetry items as possible. While the probing capacity is not depleted, the algorithm keeps repeating this procedure. Once it happens, the probing cycle returns to its origin using the shortest-path approach. The procedure is repeated until all network telemetry items are collected and all network links are visited.

The second approach is based on recent state-of-the-art work `PathPlanning` proposed by Pan et al. (PAN et al., 2019). As previously mentioned, their proposal does not consider probe capacity nor data plane telemetry items. We adapt their DFS-like algorithm to consider both requirements in the best effort approach. Our adaptation of the proposed DFS-like strategy only moves on to a next network link *iff* there is enough capacity on the current probe to collect telemetry data and return to its origin. Observe that other strategies could be useful for generating χ (e.g., turning the optimization model into a factibility one by removing the objective function).

3.3.3 Neighborhood selection and prioritization

To solve the P^{2}INT problem efficiently, we choose a subset of probing cycles $\mathcal{D} \in X$ that will be optimized. We explore the search space using VNS. In a nutshell, VNS organizes the search space in k-neighborhoods. Each neighborhood is determined as a function of the incumbent solution (χ) and a neighborhood size k. We build a neighborhood as a combination of any k probing cycles. Formally, we define a k-neighborhood as a set composed of k-tuples $\mathcal{N}_k = \{\, p \,|\, p \subseteq P \wedge y_p = 1 \,\}$. The number of neighbors in a k-neighborhood is given by the binomial coefficient $\binom{\sum_{p=1}^{P} y_p}{k}$. We focus only on active probing cycles to build our neighborhood. The reason is that the other variables in the P^{2}INT model are easily inferred once the probing cycles are defined. Note that other auxiliary variables used by the model are also not part of the neighborhood structure. With the “neighborhoods" structures, it is possible to solve the problem in parts and quickly converge to the best solution – being able to stop the execution at any point. This is desirable because loading all the variables in the model can generate a very large derivation tree of the solver (i.e., *CPLEX*, see Section 3.4.1) taking up a lot of memory space and more time to make the necessary cuts in the search for the optimal solution.

The time required by the solver to optimize a solution χ and a subset $\mathcal{D} \subseteq X$ is often small (in the order of msec.). However, processing every candidate subset $\mathcal{D}$ from the entire k-neighborhood is impractical, especially for large network instances. For this reason, we prioritize those neighbors that might lead to a better solution. We prioritize tuples in the k-neighborhood set $\mathcal{N}_k$ according to two observations: *(i)* it is more probable to merge probing cycles if they are not over-committed (i.e., the higher the residual capacity, the better); and *(ii)* it is easier to merge cycles that are close to each other. We define a tuple priority, as a function of its residual capacity. The residual

capacity of a tuple $\mathcal{T} \in \mathcal{N}_k$ is given by $r : \mathcal{T} \rightarrow \mathbb{R}^+$, according to Equation 16.

$$r(\mathcal{T}) = \sum_{p \in \mathcal{T}} \left(U(p) - \left(\sum_{i \in D} \sum_{v \in V_i} z_{p,v,i} \cdot S(v) + \sum_{i \in D} \sum_{j \in D} x_{p,i,j} \right) \right) \tag{16}$$

We break down a k-neighborhood set into two distinct sets. The first one is formed by tuples whose probing cycles sharing forwarding devices ($\mathcal{N}_{k,shr}$) – i.e., $\cap_{p \in \mathcal{T}} \neq \emptyset$. The second set is formed by remaining tuples in $\mathcal{N}_k$ ($\mathcal{N}_{k,any} = \mathcal{N}_k \setminus \mathcal{N}_{k,shr}$), *i.e.* those tuples whose cycles do not share any forwarding device. We first process the tuples of $\mathcal{N}_{k,shr}$. Then, we process the remainder ones ($\mathcal{N}_{any}$). Last, Fix-and-Optimize takes as input $NoImprov_{max}$. It indicates the maximum number of iterations without improvement that is allowed over a given neighborhood. We stop processing the current neighborhood once $NoImprov$ exceeds $NoImprov_{max}$ (line 9).

3.4 Results

3.4.1 Setup

The proposed model was run using IBM International Business Machines (IBM) *CPLEX Optimization Studio* 12.9 to obtain optimum solutions, while the proposed heuristic approach was implemented using Java language. Experiments were performed on a machine with Advanced Micro Devices (AMD) Threadripper 2920X processor and 80 GB of Random Access Memory (RAM), using the Ubuntu 16.04 operating system. We considered different physical network instances generated with Brite (MEDINA et al., 2001), following the Barabasi-Albert model (ALBERT; BARABÁSI, 2000). We used physical network infrastructures varying from 10 to 200 forwarding devices and fixed a seed value to ensure that the instances remain the same for different sizes. We vary the amount of available space to embed telemetry items in probing packets (i.e. $U(p)$) from 100 to 1500 Bytes. Further, we assume that forwarding devices have from 2 to 8 possible telemetry items to export[1], varying its size $S(v)$ uniformly from 2 to 20 Bytes (PAN et al., 2019). P^{2}INT considers the following parameters $T_{global} = 6h$, $T_{local} = 600s$, $\mathcal{K}_{init} = 2$, $\mathcal{K}_{end} = 4$, $\mathcal{K}_{inc} = 1$, and $NoImprov_{max} = 15$. The fine-tuning and sensitive analysis of these parameters is out of the scope of this work. Each experiment is repeated 30 times to ensure a confidence level of 95% or higher.

Baseline. We consider the baseline to be the optimum solution. We compare the optimum against (i) the `Edge Randomization` (`ER`), (ii) the recent state-of-the-art work proposed by Pan et al. (PAN et al., 2019), namely `PathPlanning` (`PP`) and the (iii) `Fix-and-Optmize` (`FixOpt`) metaheuristic.

Reproducibility. Our implementation is publicly available in order to encourage full reproducibility of our experiments[2] and foster the design of new solutions.

[1] In-band Network Telemetry: <https://p4.org/assets/INT-current-spec.pdf>

[2] Available implementation: <https://anonymous.4open.science/r/de4b2622-2b86-457b-82a0-fe2ad10004d8/

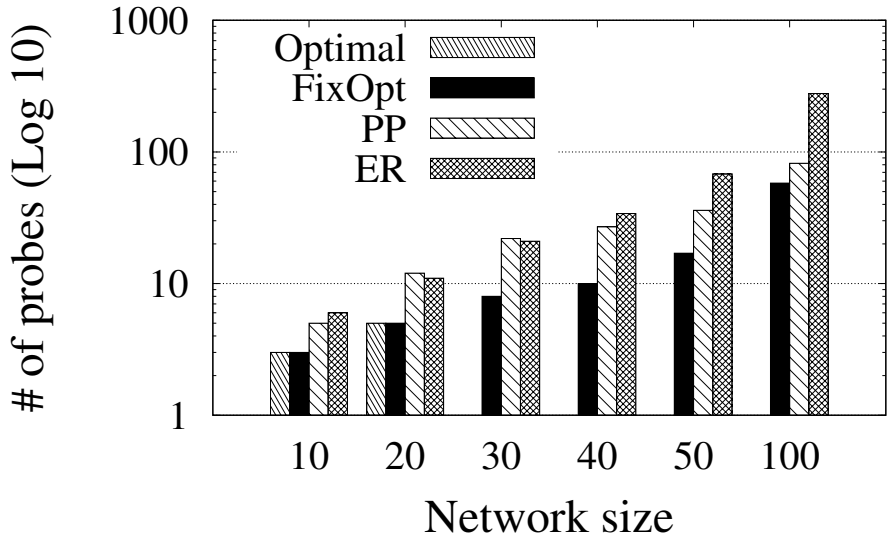


Figure 7 – Number of probing cycles for an increasing network size.

3.4.2 Results

We analyze the quality of the proposed approach by evaluating: (i) the number of probing cycles generated; (ii) the resource usage of probing cycles; (iii) the data transmission overhead to INT collectors; (iv) the INT collector usage; and (v) the network link coverage.

Probing cycles. Figure 7 illustrates the amount of probing cycles generated for an increasing size of network infrastructures (from 10 to 100). P^{2}INT comes up with quality-wise solutions compared to the optimal and the state-of-the-art approach `PP`. From a network with 30 nodes to 40 nodes there is an inversion in the quality of the `ER` and `PP` heuristics. Because `ER` is an approach with random decisions, there is no guarantee of the quality of the solution and, in some cases, it can surpass other approaches (e.g., `PP`). Fix-and-Optimize solution is able to approach the optimal value for small-scaled network infrastructures (up to 20 nodes)[3] At the same time, the `PP` produces solutions with up to 2x the number of probes considering small networks. For medium- to large-scale networks, P^{2}INT produces (on average) solutions with 2.2x and 3.70x fewer cycles compared to `PP` and `ER`, respectively. This behavior is explained by the ability of P^{2}INT to jointly route probing packets and collect telemetry items.

Probe scalability. Figure 8 depicts the impact of probing packet capacity concerning the number of generated cycles. For this evaluation, we show the results of a 50 node network infrastructure[4]. As the available probe capacity increases, we observe a sharp reduction in the number of probing cycles – as there is more room to accommodate network telemetry

>

[3] Optimal solutions for large-scale ($|D| > 20$) networks are unfeasible due to NP-hardness. For small instances, the computing time surpasses 24h.

[4] The results for other network infrastructures follow the same behavior.

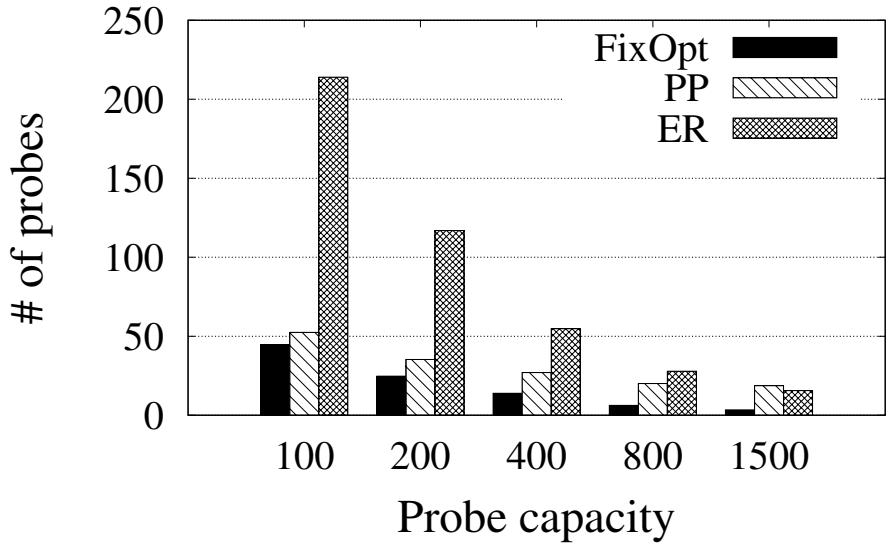


Figure 8 – Number of probing cycles for different probe capacity.

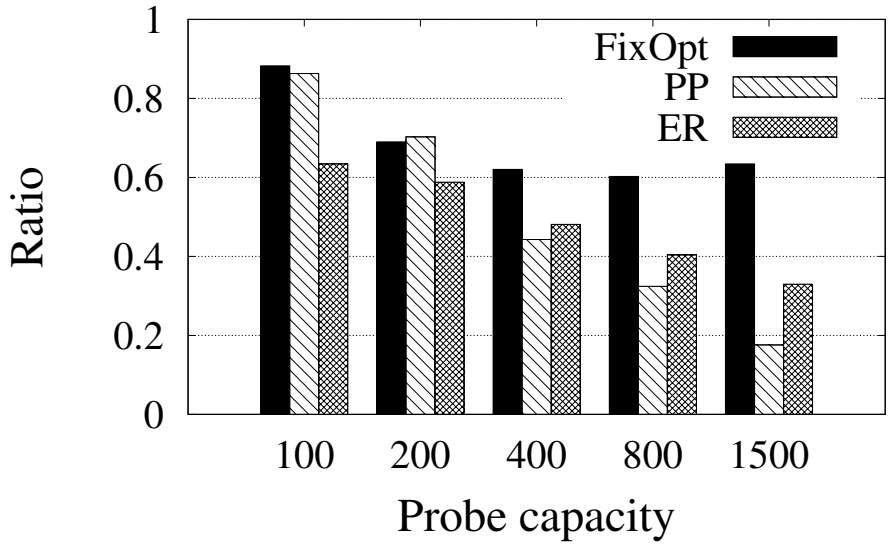


Figure 9 – Probe capacity usage.

data. When comparing P^{2}INT to its contenders, we observe that it is able to generate solutions with up to 5.5x and 4.6x fewer cycles than `PP` and `ER`, respectively – e.g., to $U(p) = 1500$. It is also important to note `ER` may generate more cycles than necessary to cover the network and its telemetry items when the probe capacity is constrained (e.g., 100 to 400 Bytes) and this happens due to its random nature in the choice of places in the creation of the cycles. On the other hand, if there is enough space to embed items, this behavior gives ER the chance to achieve better probe capacity usage, compared to the state-of-the-art, which reduces the total number of cycles (e.g., $U(p) = 1500$).

Probe capacity. Figure 9 illustrates the average probing capacity usage by generated cycles. Observe that P^{2}INT can utilize up to 3x more available capacity than `PP` (e.g., $U(p) = 1500$). On average, P^{2}INT utilizes 70% of available resources, while the other

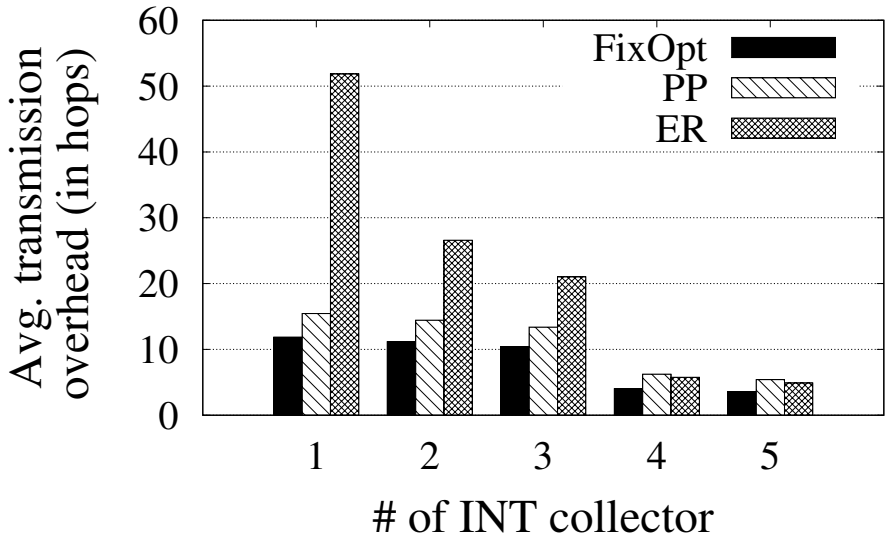


Figure 10 – Transmission overhead.

strategies (PP and ER) use 48% and 50%, respectively. By using available resources efficiently, P²INT produces solutions with minimum overheads – i.e., since fewer probe packages will be needed to cover the whole infrastructure.

Network efficiency. Probing cycles might be computed in a way that they are not routed through an INT collector. In this case, at some point in the generated cycle, the collected INT data must be sent to a given monitoring sink. For this evaluation, we consider that there are up to five INT collectors placed optimally according to the requirements of each solution. In other words, the INT collectors were placed connected to a forwarding device in such a way that the distance (in hops) to probing cycles is minimized. Figure 10 illustrates the transmission cost (in hops) to the closest INT collector. We opt to illustrate this cost in hops as the volume of transmitted data depends on the probe capacity/usage and the frequency that probe packets are generated. Observe that the more INT sinks are available in the infrastructure, the lower is the transmission overhead. It is expected since with more collectors, there is a greater probability that a cycle will find a shorter path, decreasing the transmission cost. Also, note that P²INT keeps this transmission cost as low as possible even when there exists just one INT collector. On average, PP and ER generate solutions with 1.38x and 2.26x higher transmissions overheads than P²INT, respectively.

Collector load. Figure 11 shows the collector load as the number of probing cycles assigned to each INT collector (from 1 to 5 collectors). Considering two INT collectors, solutions can cover 83% (Fix-and-Optmize), 82% (PP) and 78% (ER) of all probing cycles. Also, as explained in Figure 8, ER performs poorly when space in the probe is limited. It creates an expressive number of probes and thus, may increase the average probes per collector – especially when there are not many collectors available (e.g., 1 collector).

Network coverage. Figure 12 depicts the network link coverage as the average probing

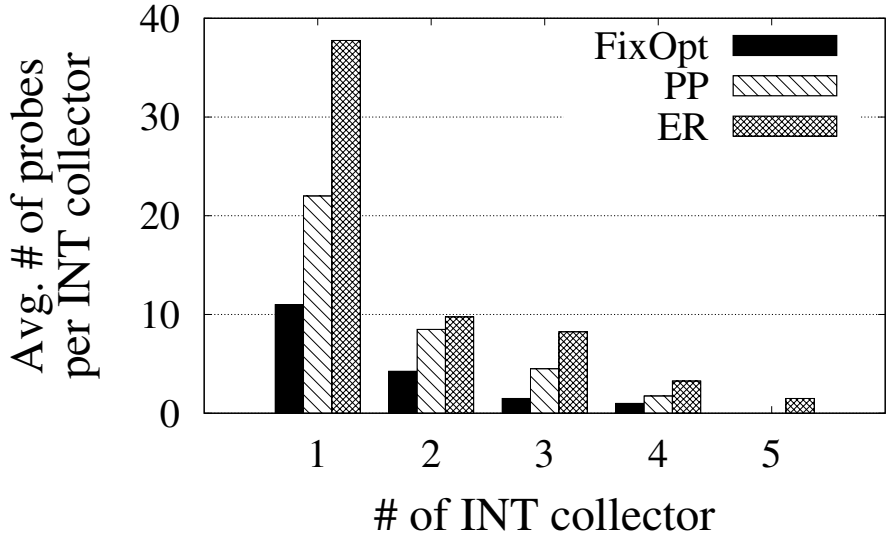


Figure 11 – Collector load.

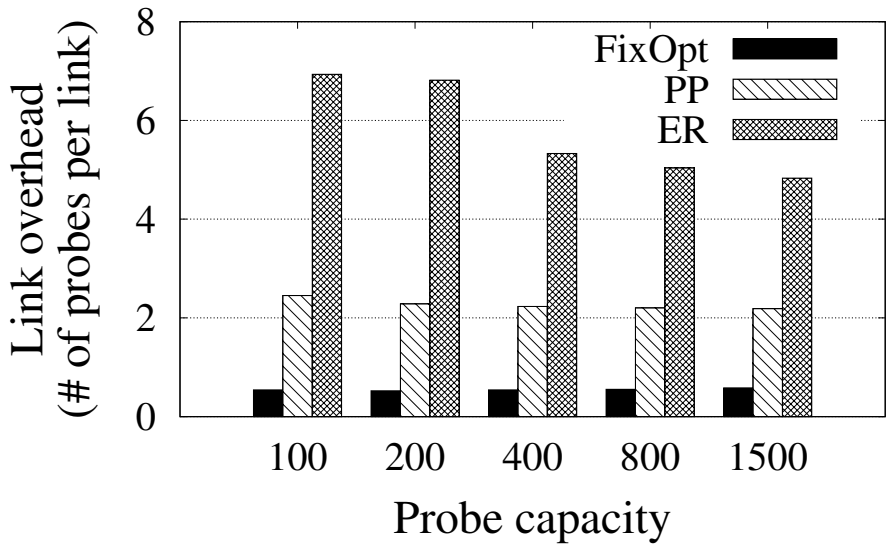


Figure 12 – Link overhead.

cycles per network link. Higher values indicate that network links are being over-covered (i.e., multiple times), representing a waste of resources. P^{2}INT, on average, keeps this value close to one, while the other strategies produce solutions with network links begin covered by up to seven probing cycles (i.e., 7x more than the necessary).

Runtime. Figure 13 shows the average runtime for an increasing size of infrastructure. Both heuristics (ER, PP) are capable of running in practically constant time for small instances and have a linear behavior. Our Approach, on the other hand, has an increase of about 10x when doubling the size of the instance (e.g., 10-20). Despite being more time consuming than the other strategies, our approach can come up with near-optimal solutions. Yet, we believe that the Probe Plannning Problem solutions can be executed in an offline manner and, therefore, there is no need for sub-second runtimes.

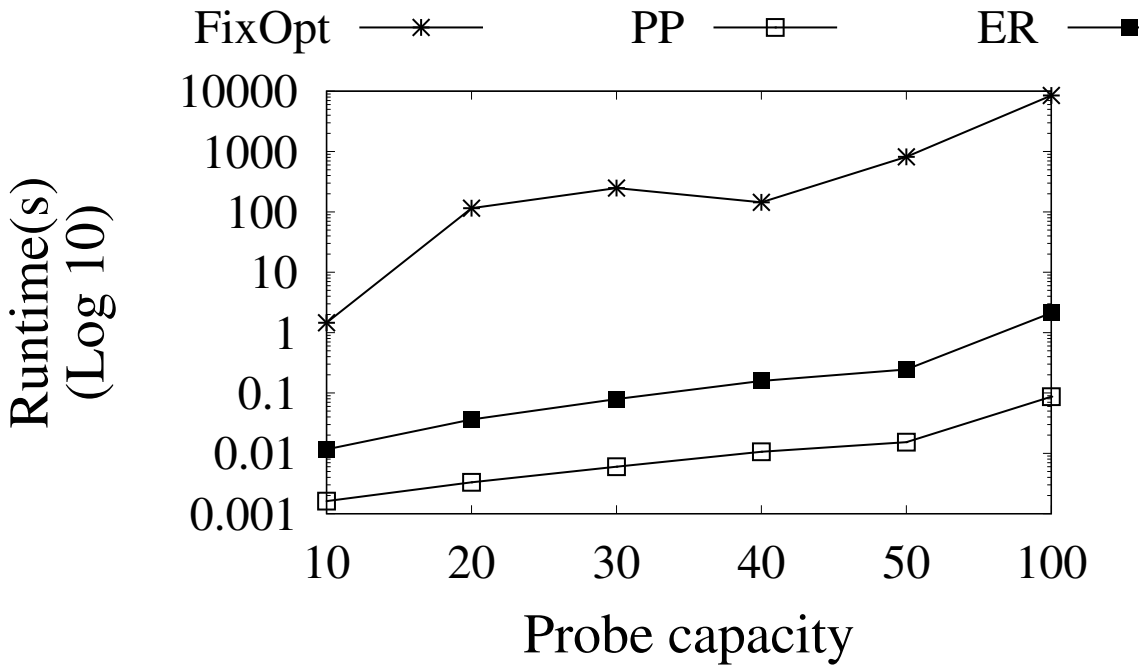


Figure 13 – Runtime.

4 FAULT-TOLERANT PROBING PLANNING FOR IN-BAND NETWORK TELEMETRY

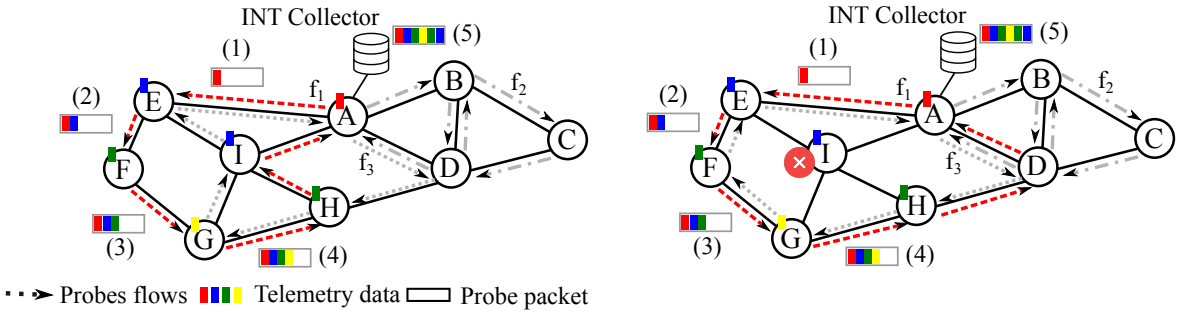
In this chapter, we theoretically formalize the Fault-Tolerant Probing Planning for In-band Network Telemetry (FP^{2}INT). First, we present a brief introduction to the problem. Then, an optimization model formally defines the problem and its constraints. Finally, we show an efficient heuristic approach to the problem and evaluate it against the state-of-art.

4.1 Problem overview

The probing planning problem consists of designing probing cycles to cover a network infrastructure in terms of telemetry data and links – that is, collecting data plane telemetry statistics in near real-time using probing packets. By covering the network infrastructure, INT enables building and maintaining an updated network-wide state – which is crucial to large-scale networks. While previous studies have focused on building feasible solutions to the in-band probing planning problem (e.g., (PAN et al., 2019; LIU et al., 2018; HOHEMBERGER et al., 2020)), *Patcher* is the first effort to tackle fault-tolerance at design. By "fault-tolerant", we consider that *Patcher* deals with failures in device-level, while still covering the whole network and its telemetry requests. In this work, a failure represents an inoperative network device (e.g., switch, router) caused by different reasons (e.g., power failure), so the interconnected links connected to this device are inoperative and the device cannot forward any information whatsoever – i.e., in this point, all pre-defined policies to work around the failure already failed. *Patcher* focuses on building a feasible solution when (multiple) node failures occur. Suppose a given set of programmable devices fail. To keep the in-band network telemetry monitoring alive, the control plane (or the data plane) has to react and reorganize existing probing cycles to keep the network-wide visibility timely. Figure 14b illustrates the case when a single programmable device fails (i.e., node I). In this case, it affects the monitoring cycles performed by f_1 and f_2. *Patcher* focuses on efficiently rebuilding those cycles to ensure that all links are still being covered while collecting the required INT data from the data planes.

For that, *Patcher* tries to minimize the changes in the current solution by applying "patches" on affected cycles (and, therefore, the name *Patcher*). In this case, we fix the probing cycle f_1 (affected by the failure) by rerouting the probing cycle through nodes $G \rightarrow H \rightarrow D \rightarrow A$, instead of $G \rightarrow I \rightarrow A$ (where node I has failed).

Re-optimizing probing cycles is an NP-hard problem and, therefore, it is not trivially solved. The hardness of this problem is due to (i) existing space constraints on probing packets (in the figure, they support up to six telemetry data); and (ii) ensuring



(a) Scenario without a failure node.

(b) Scenario with failure nodes.

Figure 14 – Reconstruction of the telemetry solution using *Patcher*.

cycle connectivity without any sub-tour (i.e., nodes on a cycle are all interconnected). A naive solution consists of generating all probing cycles in the event of node failures. Although it represents a feasible solution, it impacts the operation of active (and unaffected) monitoring cycles as all probing cycles would need to be reprogrammed by the control plane. *Patcher*, in contrast, focuses only on affected cycles – i.e., probing cycles with at least a faulty node – and, therefore, can be used by the control plane to instrument probing cycles in the event of failures, or directly by the data plane as a fast-failover mechanism.

4.2 Model description and notation

Input. The optimization model considers as input a physical network infrastructure $G = (D, L)$, a set of telemetry items V, a set of active probing cycles P, and a set of faulty nodes $D^* \subseteq D$. Set D in network G represents programmable devices $D = \{1, ..., |D|\}$, while set L consists of unidirectional links interconnecting pair of devices $(i, j) \in (D \times D)$. There exists a set of telemetry items V available in the network G. Each device $i \in D$ is able to embed a subset of items $V_i \subseteq V$ into a probing packet. Each telemetry item $v \in V$ has its size defined by function $S : V \to \mathbb{N}^+$. Conversely, set $V^* = \bigcup_{d \in D^*} V_d$ represents the set of telemetry items in faulty nodes D^*. Telemetry items are collected by probing cycles $P = \{1, ..., |P|\}$. We denote the routing cycle followed by probing $p \in P$ as function $\mathcal{C} : P \to \{D_1 \times ... \times D_{|D|}\}$. Probing cycles $p \in P$ can collect telemetry items from forwarding devices $i \in \mathcal{C}(p)$. The capacity of probing packets (e.g., the MTU) is upper bounded by a given constant, defined as $U : P \to \mathbb{N}^+$. The subset of telemetry data collected by probing cycle $p \in P$ is represented by a set of pairs $(i, v) : i \in D, v \in V_i$ and is given by the function $\mathcal{T} : P \to \{(D \times V) \times (D \times V), ..., (D \times V)\}$. For simplicity, a pair $(i, v) \in \mathcal{T}$ is indexed as $_{[1]}$ and $_{[2]}$ to refer to the first and second element, respectively.

Constraints. Next, we describe the main feasibility constraints related to the problem. The problem is subject to (i) network telemetry coverage constraints; (ii) net-

work link coverage constraints; (iii) cycle capacity; and (iv) cycle connectivity constraints.

(i) Network telemetry coverage: all network telemetry statistics available on data plane devices $V_i : (\forall i \in D)$ need to be collected by at least on active probing cycle P. For each network device $i \in D$, we keep track of collected telemetry items by existing probing cycles. Formally,

$$\Big|\Big(\bigcup_{\substack{(p\in P,(j,v)\in\mathcal{T}(p)\\ |\, i=j)}} (j,v)_{[2]}\Big)\bigcap V_i\Big| = \big|V_i\big| \qquad : (\forall i \in D) \quad (4.1)$$

(ii) Network link coverage: Network links L from the network G need to be covered by at least on probing cycle P. For each network link $(k,l) \in E$, we count the occurrences on existing probing cycles P (which should be greater than one). Formally,

$$\Big|\bigcup_{\substack{(p\in P,i=1\\ |\,\mathcal{C}(p)_{[i]}=k\\ \wedge\mathcal{C}(p)_{[i+1]}=l)}}^{i\leq(|\mathcal{C}(p)|-1)} (\mathcal{C}(p)_{[i]},\mathcal{C}(p)_{[i+1]})\Big| \geq 1 \qquad : (\forall (k,l) \in L) \quad (4.2)$$

(iii) Cycle capacity: Probing cycles are upper bounded by $U(p)$ to embed data plane statistics and accounting for used links and nodes. Usually, INT procedures utilize up to 1 byte to store device/link ID. We sum the cycle length (i.e., $|C(p)|$) with the telemetry usage to avoid creating unrealistic, lengthy cycles.

$$\Big(\sum_{(i,v)\in\mathcal{T}(p)} (j,v)_{[2]} \cdot S(v)\Big) + |C(p)| \leq U(p) \qquad : (\forall p \in P) \quad (4.3)$$

(iv) Cycle connectivity: Probing cycles need to be well constructed. A valid cycle $\mathcal{C} : P \rightarrow \{D_1 \times ... \times D_{|D|}\}$ is the one that starts and ends at the same device (Equation 4), while all devices $i \in \mathcal{C}(p)$ are pairwise strongly connected, i.e., any pair of devices in $\mathcal{C}(p)$ are reachable to each other. To describe this property, we recall a auxiliary function $\delta : (P \times D \times D) \rightarrow \{true, false\}$ that returns *true* in case there exists a path between node i and j in probing path P, i.e. $\mathcal{C}(p)_{[i]} \rightarrow ... \rightarrow \mathcal{C}(p)_{[j]}$, where $(\mathcal{C}(p)_{[i]},\mathcal{C}(p)_{[i+1]}) \in L$. Otherwise, function δ returns *false*.

$$\mathcal{C}(p)_{[1]} = \mathcal{C}(p)_{[|\mathcal{C}(p)|]} \qquad : (\forall p \in P) \quad (4.4)$$

$$\delta(p,i,j) = true \qquad : (\forall p \in P), \forall (i,j) \in (C(p) \times C(p)) \quad (4.5)$$

Given the feasibility constraints defined above, we assume there exists an assignment function $\mathcal{A} : (G,V) \rightarrow (\mathcal{C},\mathcal{T})$ that, given a network infrastructure G, and a set of telemetry items V, it returns a feasible solution $(\mathcal{C},\mathcal{T})$, with respect to constraints *(i)*, *(ii)*, *(iii)*, and *(iv)*.

Objective. Given a feasible solution $(\mathcal{C},\mathcal{T})$ and a set of faulty nodes D^*, the optimization problem seeks a new assignment $\overline{\mathcal{A}} : (G - \{D^*\}, V - \{V^*\}) \rightarrow (\overline{\mathcal{C}},\overline{\mathcal{T}})$ that

minimizes the number of changes in the current solution $(\mathcal{C}, \mathcal{T})$. In other words, the solution of $\mathcal{A}$ and $\overline{\mathcal{A}}$ should be as similar as possible, despite the required changes with respect of faulty nodes D^* The objective function aims to minimize the changes made on current probing cycles by maximizing the intersection between the current and the new solution. Equation (6) describes the objective function. Observe that parameters α and β are used for weighting the importance of probing cycle structures (i.e., α) and telemetry items assignments (i.e., β).

$$\textbf{Maximize} \quad \alpha \cdot \sum_{p \in P} |\mathcal{C}_p \cap \overline{\mathcal{C}_p}| + \beta \cdot \sum_{p \in P} |\mathcal{T}_p \cap \overline{\mathcal{T}_p}| \tag{4.6}$$

4.3 Proposed Heuristic Approach

To tackle the above problem efficiently – i.e., in terms of time and space – and provide a quality-wise solution, we propose a heuristic procedure that only rebuilds parts of probing cycles affected by the faulty nodes. Next, we overview the ideas behind our proposed heuristic, and then we discuss the pseudo-code and its complexity analysis.

The heuristic procedure aims to maintain all network links covered while collecting all telemetry items from the remaining devices working. The main idea consists of reconstructing only affected cycles by faulty nodes Algorithm 2 summarizes *Patcher*. Our heuristic receives as input a feasible solution $(\mathcal{C}, \mathcal{T})$ and the set of faulty nodes D^*. Such a viable initial solution can be obtained, for example, through a constructive heuristic that solves the CARP problem - i.e., that covers all network links (see Section 3.3.2). In lines 1-5, we "patch" all affected cycles[1] by directly connecting the predecessor and the successor of a given faulty node $i \in D^*$. We perform this new interconnection using the shortest path. As we do not control the length of this new interconnection, it could be that the applied "patch" violates the probe's capacity. In this case, we exclude telemetry items from the affect cycles until it no longer violates the probe's capacity (lines 6-11). Then, the algorithm iterate over the unsatisfied items U_{dv} (i.e., those removed from probing cycles) and look for a probing $p \in P$ that can collect it again (lines 12-19). If the existing probing paths P could not collect all the remaining items, we then create new probing cycles for that purpose (lines 20-29). We sort telemetry items U_{dv} with respect to forwarding devices D to ensure that we collect items in a given order. Finally, we apply a local search procedure (line 30) that further optimizes our incumbent solution. The local search procedure strives to minimize the number of probing cycles by removing cycles in which forwarding devices have already been covered by other probes. We iterate over each pair of probing cycles, checking whether or not it is possible to delete. Our proposed approach has a worst-case time complexity of $\mathcal{O}(D \cdot V \cdot P)$, considering the most consuming code section (lines 12-19).

[1] We consider that there is some mechanism (e.g., timeout) responsible for informing that the device is not working for the rest of the network infrastructure.

Algorithm 2 Overview of the patcher procedure.

Input: $(\mathcal{C}, \mathcal{T})$: initial solution; D^*: subset of faulty nodes.

for all devices $i \in \mathcal{C}(p) : (\forall p \in P)$ **do**
  if $i \in D^*$ **then**
    $\mathcal{C}(p) \leftarrow \mathcal{C}(p) - i \cup \text{ShortestPath}(\mathcal{C}(p)_{[i-1]}, \mathcal{C}(p)_{[i+1]})$
  end if
end for
for all cycles $p \in P$ such that Equation (3) is not satisfied **do**
  while capacity $U(p)$ constraint is not satisfied **do**
    exclude a collected item, until it does
    Save item on U_{dv}
  end while
end for
for all unsatisfied items $(d, v) \in U_{dv}$ **do**
  for all $p \in P$ the cycles **do**
    if cycle p has residual capacity $U(p) - S(v) \geq 0$ **then**
      $U_{dv} \leftarrow U_{dv} - (d, v)$; $U(p) \leftarrow U(p) - S(v)$
      $\mathcal{T}(p) \leftarrow \mathcal{T}(p) \cup (d, v)$
    end if
  end for
end for
Sort $(d, v) \in U_{dv}$ regarding device d.
while $U_{dv} \neq \varnothing$ **do**
  $P \leftarrow$ new probe containing a cycle $C(p_{new})$ which prioritizes unsatisfied items U_{dv}
  while $(d, v) \in U_{dv}$ **do**
    if cycle p has residual capacity $U(p) - S(v) \geq 0$ **then**
      $U_{dv} \leftarrow U_{dv} - (d, v)$; $U(p) \leftarrow U(p) - S(v)$
      $\mathcal{T}(p) \leftarrow \mathcal{T}(p) \cup (d, v)$
    end if
  end while
end while
Apply local search
return new solution $(\mathcal{C}, \mathcal{T})$

Algorithm 3 Overview of the local search procedure.

Input: $(\mathcal{C}, \mathcal{T})$: initial solution containing the probing cycles, $\mathcal{D}$: subset of failure nodes.
Output: χ: best solution found after the procedure

$\chi \leftarrow (\mathcal{C}, \mathcal{T})$
for all cycles $p_i \in P$ **do**
  for all other cycles $p_j \neq p_i$ **do**
    if a $p_j \in P$ cycles contains all of p_i's devices and $\mathcal{T}(p_i) = \varnothing$ **then**
      $P \leftarrow P - p_i$
      break
    end if
  end for
end for
return χ

We further design a local-search procedure (Algorithm 3) that maintains all the constraints (Section 4.2) satisfied. The idea of this procedure is to reduce non-optimized

probing cycles. From the initial solution (line 1) found by *Patcher*, we iteratively try to reduce the number of cycles. Our goal is to look for other cycles that satisfy the same subset of devices along its path and do not collect telemetry items – i.e., underused probes (lines 2-9). If this condition is satisfied, the repeated cycles are excluded to simplify our solution (line 5).

4.4 Evaluation

4.4.1 Setup.

All experiments were performed on a machine with an AMD Threadripper 2920X processor and 80 GB of RAM, using the Ubuntu 16.04 operating system. We considered physical network instances that were generated with Brite (MEDINA et al., 2001), following the Barabasi-Albert model (ALBERT; BARABÁSI, 2000). We used physical network infrastructures varying from 10 to 100 forwarding devices. We vary the amount of available space to embed telemetry items in probing packets (i.e. $U(p)$) from 100 to 1500 Bytes. Further, we assume that forwarding devices have 8 possible telemetry items to export[2], varying its size $S(v)$ uniformly from 2 to 20 Bytes (PAN et al., 2019). Yet, we consider that there exist a single faulty node (i..e, $|D^*| = 1$) in the network infrastructure G. In each execution, we vary this set to cover all devices $i \in D$, ensuring that all devices fail individually. We consider $\alpha = 1$ and $\beta = 1$. As future work, we left the evaluation of higher values for $|D^*|$, as well the fine-tuning of α and β.

We compare the results obtained by *Patcher* against (i) the `Edge Randomization (ER)` (BELENGUER et al., 2006), and (ii) the recent state-of-the-art work proposed by Pan et al. (PAN et al., 2019), namely `Path Planning` (`PP`). `Edge Randomization` is a heuristic procedure used to build feasible solutions to the arc routing problem. Specifically, it is a variant of the Path Scanning (GOLDEN; DEARMON; BAKER, 1983) algorithm for the Capacitated Arc Routing Problem (CARP), where each link has a demand to be satisfied and a single starting point. We modified such a strategy to the problem at hand as follows. We randomly select a starting device $d \in D$ to start the probing cycles p. While the capacity $U(p)$ is not violated, we collect all telemetry items V_i of the current node i and randomly select the next forwarding device to further expand the current probing cycle. At the point that the capacity $U(p)$ is reached, we return to the origin forwarding device (the procedure ensures that there is enough capacity to make the way back). In turn, `Path Planning` proposes a DFS-like strategy to create an *Euler-trail* based algorithm. A trail is a walk in a graph without repeated edges. In turn, a *Euler trail* visits each edge exactly once. In this strategy, Euler circuits are used. It is a special Euler trail on which the starting/ending point is the same.

[2] In-band Network Telemetry: <https://p4.org/assets/INT-current-spec.pdf>

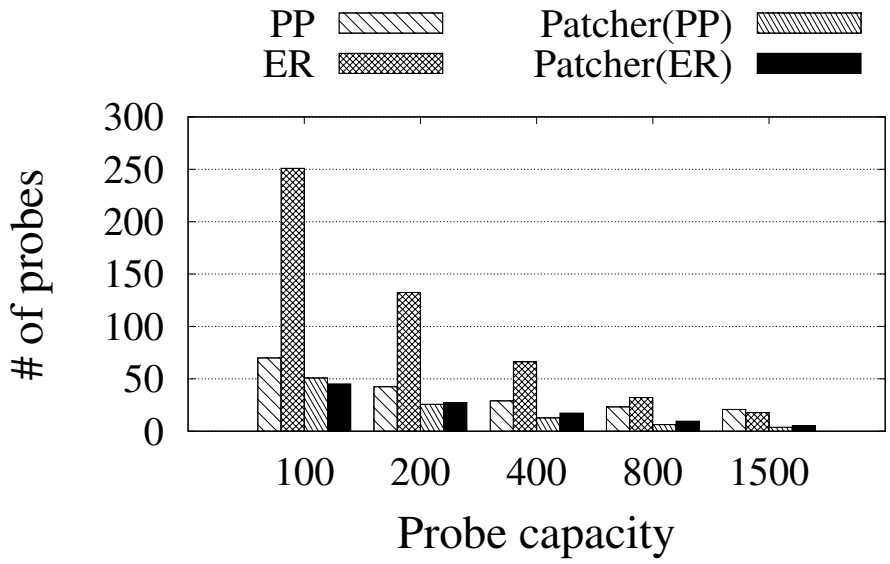


Figure 15 – Number of probing cycles for an increasing network size.

Our initial solution to the problem (Chapter 4) is given according to the above algorithms (i.e., `ER` and `PP`). On the event of a failure in devices D^*, we apply *Patcher* on top of existing solutions – mentioned as *Patcher*(`ER`) or *Patcher*(`PP`). Alternatively, we re-execute algorithms `ER` and `PP` from scratch (considering faulty nodes D^*) as a baseline comparison.

Metrics. We focus our evaluation on five metrics: (i) number of probing cycles, (ii) INT collector overhead (i.e., the number of probing cycles assigned to a given INT collector), (iii) transmission overhead (i.e., the distance between the probing cycles and the closest INT collector), (iv) the number of changes in current solution regarding links (i.e., the difference between sets $\mathcal{C}_p$ and $\overline{\mathcal{C}_p}$), and (v) the number of changes in current solution regarding telemetry data (i.e., the difference between sets $\mathcal{T}_p$ and $\overline{\mathcal{T}_p}$).

4.4.2 Results

Number of probing cycles. Figure 15 illustrates the average amount of probing cycles after a given faulty node, varying probing cycle from 100 to 1500. Note that, in all graphs, we show the average considering all faulty nodes D^*. *Patcher* can substantially reduce the number of probing cycles after an observed fault in comparison to `ER` (up to 5.5x) and `PP` (up to 1.38x). it is mostly due to reorganizing existing probing cycles instead of rebuilding a new solution from scratch. The number of probing cycles reduces as fewer network telemetry items are to be collected (and links to be covered) in a faulty situation.

Collector and transmission overhead. Figure 16 and Figure 17 illustrate the collector and transmission overhead, respectively. By collector overhead (Figure 16), it means that the more probe packets are assigned to a given collector, the greater will be its workload. In both figures, we increase the number of INT collectors from 1 to 5. These metrics are

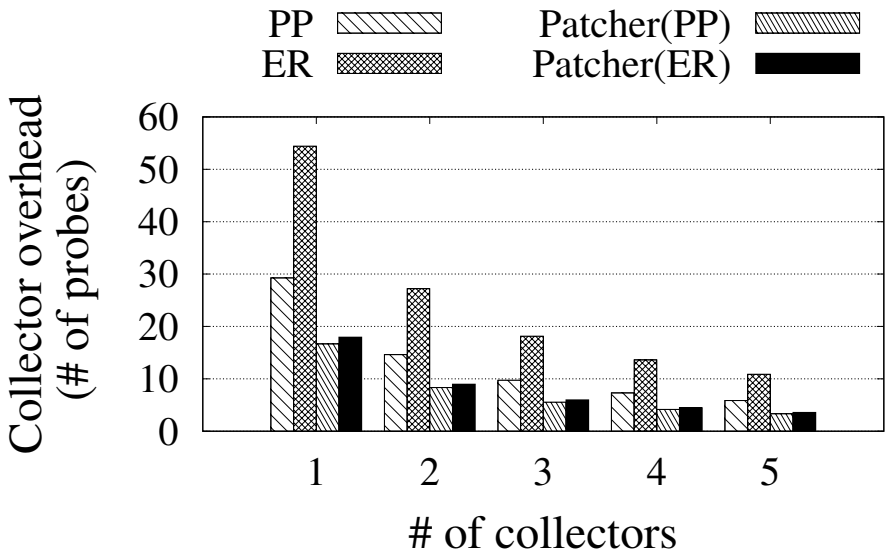


Figure 16 – Collector load.

calculated considering the optimal placement of INT collectors. That is, given a set of probing cycles, we place the available INT collectors in a given set of forwarding node D to minimize the value of these metrics. Figure 16 illustrates the average collector overhead: the amount of probing cycles assigned to each of them. The lower the number of INT collectors, the higher is the overhead. When there is a fault in the network, we analyze how this collector load is affected. Observe that *Patcher* reduces (up to 3x) this overhead as a consequence of reducing the number of probing cycles (illustrated in Figure 15). In turn, Figure 17 illustrates the average transmission overhead (i.e., the number of hops between probing cycles and placed INT collectors). Like the collector overhead, the transmission overhead is affected by the number of INT collectors. The more INT collectors, the lower is the transmission costs – as this increases the chance of having an INT collector closer to a given probing cycle. When a fault occurs, the transmission costs are maintained (or reduced up to 60.9%) in comparison to re-executing the whole solution.

Changes in existing solutions. Figure 18 and Figure 19 illustrate the average amount of changes required to implement a new solution on the event of a fault – regarding changes in links and telemetry items assignments, respectively. Observe that these two metrics are rather important in order to assess the recovery time taken by programmable network infrastructure in the case of a fault. Note, as well, that these changes can be instrumented directly by the control plane (e.g., by installing new rules in forwarding devices) or be encoded into INT probing packets to be handled directly by the data plane. In either case, a low number of changes are expected to reduce the recovery time or the resource usage of probing packets. Figure 18 illustrates the number of probing cycle links that have been changed from the initial solution to the new one (e.g., given by *Patcher* or re-executing the algorithms from scratch). Observe that, in general, *Patcher* reduces substantially the number of observed changes in existing solutions as it applies “patches”

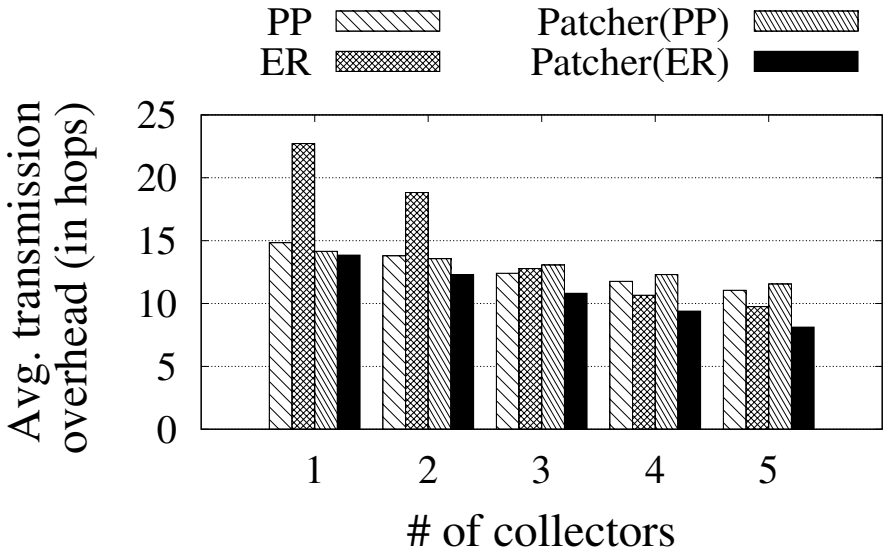


Figure 17 – Minimum distance to the closest collector.

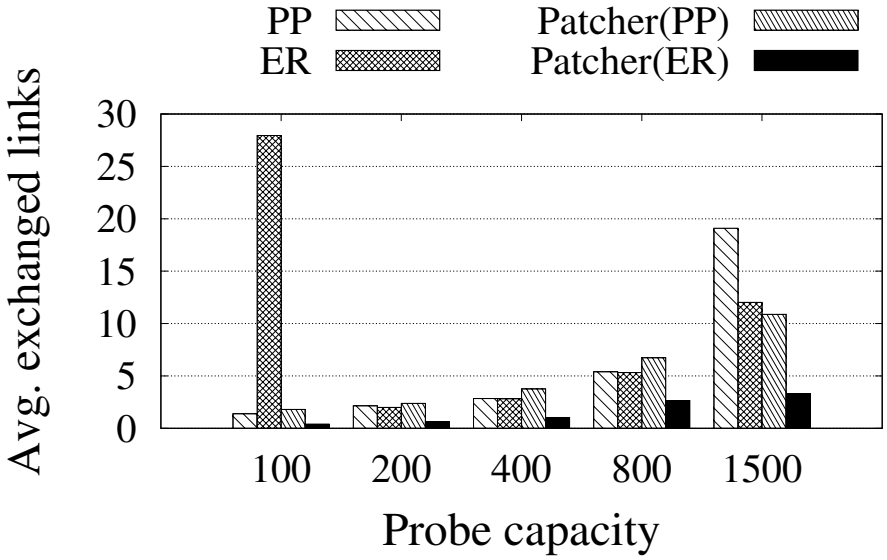


Figure 18 – Average difference of $|\mathcal{C}_p \cap \overline{\mathcal{C}_p}|$.

only on affected parts of probing cycles. *Patcher* can reduce up to 3.6x and 1.75x the link changes ($U(p) = 1500$) in comparison to `ER` and `PP`, respectively. In turn, Figure 19 shows the number of telemetry items data has been re-assigned from one probing cycle to another. It happens due to space constraints in existing cycles. Re-executing the whole leads to totally different item attributions. We need to avoid that in order to assess the recovery time taken by infrastructure in case of a fault. Eventually, a feasible solution encompasses the reallocation of existing telemetry items among available probing cycles when a fault occurs. As observed in the figure, the solutions produced by *Patcher* add a negligible amount of changes in comparison to `ER` (up to 28.5x) and `PP` (up to 4x), when probe capacity is equal to 1500 Bytes.

Runtime. Figures 20 and 21 depict the runtime for the trivial solution and *Patcher*,

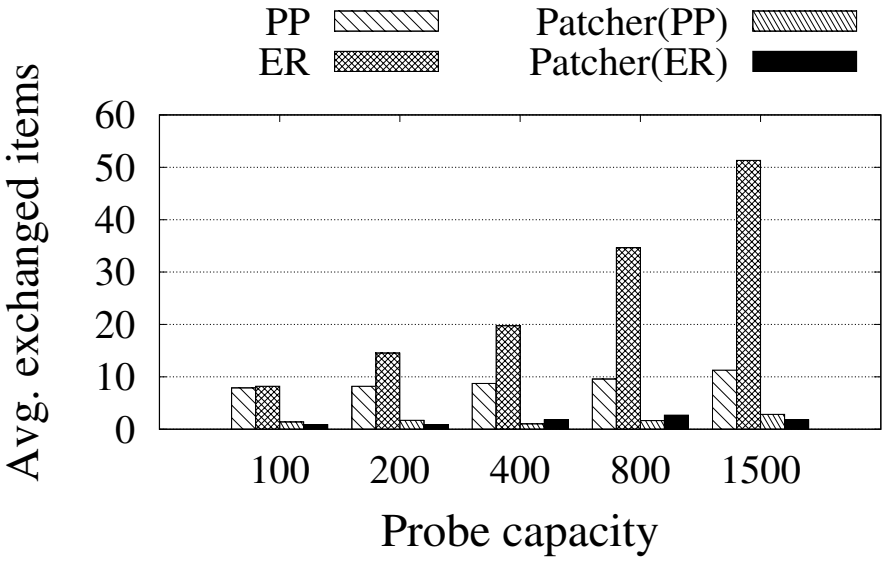


Figure 19 – Average difference of $|\mathcal{T}_p \cap \overline{\mathcal{T}_p}|$.

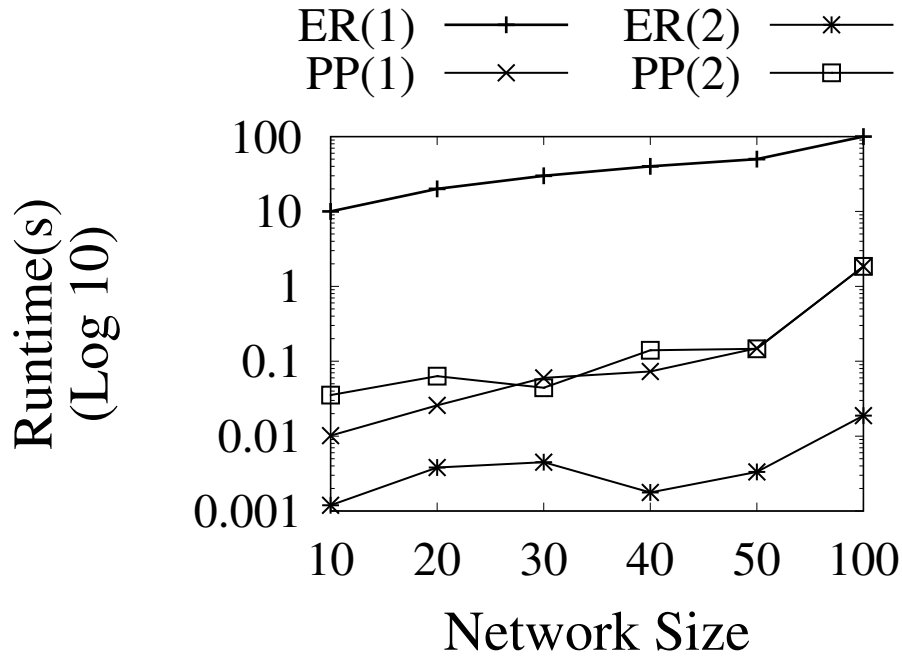


Figure 20 – Trivial solution runtime.

respectively for 1 and 2 faulty nodes. The trivial solution consists of running the heuristic again on top of faulty nodes. As it is possible to observe in Figure 20, the trivial solution can be as costly as 100 seconds (to a network size of 100 nodes). In contrast, when executing *Patcher* (Figure 21), we observe that the running time (even for large instances) is bounded by at most 5 seconds.

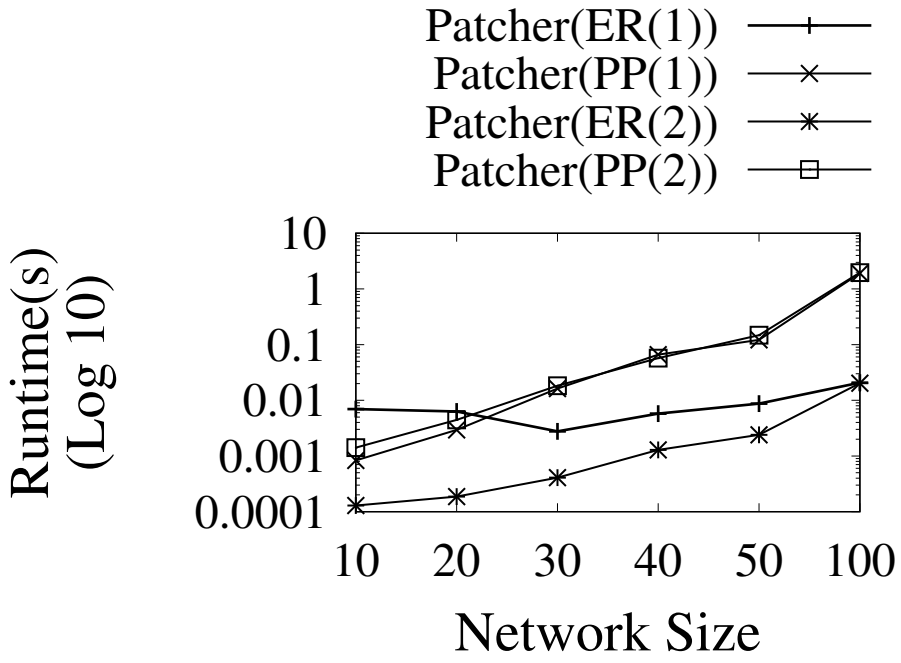


Figure 21 – Patcher runtime.

5 FINAL REMARKS

In this chapter, we describe our envisioned next research steps in order to finalize this work. We first provide an overview of the already accomplished results and then we describe all required steps toward the final goal, as well as the corresponding schedule of each activity.

In this work, we introduced two different optimization problems related to INT-based network monitoring orchestration, namely Probing Planning for In-band Network Telemetry (P^{2}INT) and Fault-tolerant Probing Planning for In-band Network Telemetry (FP^{2}INT). P^{2}INT is formalized employing a MILP model. It can be seen as a generalization of the Bin Packing problem and both problems (i.e., P^{2}INT and FP^{2}INT) can be seen as a generalization of the Capacitated Arc Routing Problem (CARP). Both models try to solve similar problems for programmable networks. P^{2}INT aims to orchestrate how probes are routed and generated across the network to cover all links while collecting all telemetry items available. For that, we introduced a scalable mathematical-based heuristic that iteratively chooses which variables would get fixed – i.e., remain static – and which variables would be optimized (varied), prioritizing cycles with great potential for optimization. In our results, our approach outperforms state-of-the-art heuristics (e.g., factor 6x with respect to probing cycles). However, it is still limited to (i) statistic solutions over time (i.e., probing cycles do not change); and (ii) fixed-throughput of probing packets (i.e., all probing cycles operated at the same packet rate). On the other hand, the proposed FP^{2}INT considers there is an initial feasible solution covering the network – such as the provided by P^{2}INT – and allows to reconstruct the initial solution to still cover the network and collect required telemetry information when fails occur in network devices (e.g., switches, routers). For that, *Patcher* iteratively iterates over existing probe cycles and "patches" it whenever we find a faulty node, besides trying to optimize the current solution and build more probe cycles prioritizing the collecting of remaining telemetry items if needed. As shown, our solution (i) reduces the number of probes by a factor of 5.5x, (ii) makes better use of network resources by keeping the collector and transmission overheads of probing cycles as low as possible, and (iii) requires up to 98% fewer changes (in comparison to baselines) to implement our fault-tolerant mechanism.

5.1 Achievements

The development of this work has led to the publication of the following peer-reviewed/journal papers:

1. Castro et al. Análise do Desempenho de Heurísticas na Coleta de Informações de Telemetria In-Band. In: 17^a Escola Regional de Redes de Computadores (ERRC 2019) (CASTRO et al., 2019)

2. Castro et al. Heurística construtiva para o problema de orquestração da coleta de dados de telemetria in-band. In: Anais da XX Escola Regional de Alto Desempenho da Região Sul (ERRC 2020) (CASTRO et al., 2020a)

3. Rumeningue et al. Orchestrating in-band data plane telemetry with machine learning. In: IEEE Communications Letters, 2019 (HOHEMBERGER et al., 2019)

4. Castro et al. Patcher: Towards fault-tolerant probing planning for in-band network telemetry. In: IEEE Latin-American Conference on Communications(LATINCOM) (CASTRO et al., 2020b)

5. Castro et al. Near-Optimal Probing Planning for In-band Network Telemetry. In: IEEE Communications Letters (Castro et al., 2021)

6. Viegas et al. The Actual Cost of Programmable SmartNICs: diving into the existing limits. In: International Conference on Advanced Information Networking and Applications (AINA 2021) (approved)

5.2 Future Work

In this section, we discuss what was not in the scope of our work and what we plan to do in future directions to mitigate such limitations.

In both presented versions of the mentioned problem of orchestrating INT probes across the network (i.e., P^{2}INT and FP^{2}INT) we do not consider temporal and space constraints required by monitoring applications as in a previous work (HOHEMBERGER et al., 2019). Temporal constraints dictate on what frequency telemetry items should be collected from the infrastructure, while space constraints determine telemetry information should be gathered from a certain device or subset of devices. These constraints are important to guarantee SLAs are being served by properly providing information to monitoring applications (e.g., DDoS defender, firewall). In future work, we will add these restrictions to the model, as part of the policy of creating probing cycles and test these changes considering a real environment – e.g., with SmartNICs.

Patcher (Chapter 4) has a polynomial behavior, while the fix-and-optimize approach (Chapter 3) is exponential in worst-scenario cases – i.e., several telemetry items and devices to be satisfied – which makes it very computationally costly for larger instances. Currently, our solution provides quality solutions close to the optimum only for midsize network instances (e.g., 100-200 nodes). It is important to mention that *Patcher* does not work when (i) there is no minimum path from the predecessor point to the failure to the next node at the point of failure and when (ii) a failure occurs at a point that connects the entire infrastructure (a bridge). However, no other solution covers the latter and it is unlikely that an actual infrastructure will be designed with a critical point of failure. In future work, alternatives to support hundreds/thousands of devices in a viable

time with minimal use of resources (e.g., CPU, memory) will be explored. Last, when a failed device happens to start working again, we could just utilize the initial solution from a heuristic such as ER. However, this may take a considerable amount of time to reconfigure all INT devices to configure the new routing policies and items' attributions. In future work, we aim to mitigate that problem to reduce the attribution overhead for devices that are back up and running.

BIBLIOGRAPHY

ADRICHEM, N. L. V.; DOERR, C.; KUIPERS, F. A. Opennetmon: Network monitoring in openflow software-defined networks. In: IEEE. **2014 IEEE Network Operations and Management Symposium (NOMS)**. [S.l.], 2014. p. 1–8. Cited in page 35.

ALBERT, R.; BARABÁSI, A.-L. Topology of evolving networks: Local events and universality. **Physical Review Letters**, American Physical Society, v. 85, p. 5234 – 5237, Dec 2000. Cited 2 times in the pages 47 and 58.

ALEXANDROV, E.; KAZYMOV, A.; PROKOSHIN, F. **BigData tools for the monitoring of the ATLAS EventIndex**. [S.l.], 2018. Cited in page 31.

ARAKAKI, R. K.; USBERTI, F. L. An efficiency-based path-scanning heuristic for the capacitated arc routing problem. **Computers & Operations Research**, Elsevier, v. 103, p. 288–295, 2019. Cited in page 38.

Arista. 2018. Disponível em: <https://www.arista.com/en/solutions/telemetry-analytics>. Cited in page 33.

BAS C. CASCONE, J. L. P. K. S. S. S. T. T. J. V. A. Y. T. A. **p4language**. [S.l.]: GitHub, 2020. <https://github.com/p4lang>. Cited in page 33.

BASAT, R. B. et al. Constant time updates in hierarchical heavy hitters. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 127–140. ISBN 978-1-4503-4653-5. Disponível em: <http://doi.acm.org/10.1145/3098822.3098832>. Cited in page 35.

BASAT, R. B. et al. Pint: Probabilistic in-band network telemetry. **arXiv e-prints**, p. arXiv–2007, 2020. Cited 2 times in the pages 37 and 38.

BELENGUER, J.-M. et al. Lower and upper bounds for the mixed capacitated arc routing problem. **Computers & Operations Research**, Elsevier, v. 33, n. 12, p. 3363–3383, 2006. Cited in page 58.

BERDE, P. et al. Onos: towards an open, distributed sdn os. In: **Proceedings of the third workshop on Hot topics in software defined networking**. [S.l.: s.n.], 2014. p. 1–6. Cited in page 28.

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 2 times in the pages 23 and 33.

CASADO, M. et al. Ethane: Taking control of the enterprise. **ACM SIGCOMM computer communication review**, ACM New York, NY, USA, v. 37, n. 4, p. 1–12, 2007. Cited in page 28.

CASADO, M. et al. Sane: A protection architecture for enterprise networks. In: **USENIX Security Symposium**. [S.l.: s.n.], 2006. v. 49, p. 50. Cited in page 28.

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 27.

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 in page 36.

CASTRO, A. de et al. Heurística construtiva para o problema de orquestração da coleta de dados de telemetria in-band. In: **Anais da XX Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2020. p. 33–36. ISSN 2595-4164. Disponível em: <https://sol.sbc.org.br/index.php/eradrs/article/view/10749>. Cited in page 66.

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 in page 66.

Castro, A. G. et al. Near-optimal probing planning for in-band network telemetry. **IEEE Communications Letters**, p. 1–1, 2021. Cited in page 66.

CASTRO, A. G. de et al. Análise do Desempenho de Heurísticas na Coleta de Informações de Telemetria In-Band. In: **17a Escola Regional de Redes de Computadores**. Alegrete-RS, Brasil: [s.n.], 2019. Disponível em: <http://errc.sbc.org.br/2019/papers/castro2019anlise.pdf>. Cited in page 65.

CHEN, X. et al. Fine-grained queue measurement in the data plane. In: **Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies**. New York, NY, USA: Association for Computing Machinery, 2019. (CoNEXT '19), p. 15–29. ISBN 9781450369985. Disponível em: <https://doi.org/10.1145/3359989.3365408>. Cited in page 39.

CHEN, Y.; HAO, J.-K.; GLOVER, F. A hybrid metaheuristic approach for the capacitated arc routing problem. **European Journal of Operational Research**, Elsevier, v. 253, n. 1, p. 25–39, 2016. Cited in page 38.

Cisco. 2018. Disponível em: <https://www.cisco.com/c/en/us/solutions/service-provider/cloud-scale-networking-solutions/model-driven-telemetry.html>. Cited in page 33.

CLAISE, B. et al. Cisco systems netflow services export version 9. RFC 3954, October, 2004. Cited in page 27.

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 36.

FILHO, R. I. T. da C. et al. Scalable qoe-aware path selection in sdn-based mobile networks. In: **IEEE INFOCOM 2018 - IEEE Conference on Computer Communications**. [S.l.: s.n.], 2018. p. 989–997. Cited in page 41.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN 0716710447. Cited in page 25.

GENG, Y. et al. {SIMON}: A simple and scalable method for sensing, inference and measurement in data center networks. In: **16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)**. [S.l.: s.n.], 2019. p. 549–564. Cited 5 times in the pages 23, 24, 37, 38, and 39.

GOLDEN, B. L.; DEARMON, J. S.; BAKER, E. K. Computational experiments with algorithms for a class of routing problems. **Computers & Operations Research**, Elsevier, v. 10, n. 1, p. 47–59, 1983. Cited in page 58.

GOLDEN, B. L.; WONG, R. T. Capacitated arc routing problems. **Networks**, v. 11, n. 3, p. 305–315, 1981. Cited in page 43.

GRAFANA. **Grafana: The open observability platform**. 2020. Disponível em: <https://grafana.com/>. Cited in page 31.

GUDE, N. et al. Nox: towards an operating system for networks. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 38, n. 3, p. 105–110, 2008. Cited in page 28.

GUPTA, A. et al. Sonata: Query-driven streaming network telemetry. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2018. p. 357–371. Cited in page 35.

HANDIGOL, N. et al. I know what your packet did last hop: Using packet histories to troubleshoot networks. In: **11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)**. Seattle, WA: USENIX Association, 2014. p. 71–85. ISBN 978-1-931971-09-6. Disponível em: <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol>. Cited in page 34.

HOHEMBERGER, R. et al. Orchestrating in-band data plane telemetry with machine learning. **IEEE Communications Letters**, v. 23, n. 12, p. 2247–2251, 2019. Cited 4 times in the pages 24, 25, 41, and 42.

HOHEMBERGER, R. et al. Orchestrating in-band data plane telemetry with machine learning. **IEEE Communications Letters**, IEEE, 2019. Cited 2 times in the pages 39 and 66.

HOHEMBERGER, R. et al. A heuristic approach for large-scale orchestration of the in-band data plane telemetry problem. In: BAROLLI, L. et al. (Ed.). **Advanced Information Networking and Applications**. Cham: Springer International Publishing, 2020. p. 381–392. ISBN 978-3-030-44041-1. Cited in page 53.

Huawei. 2018. Disponível em: <http://support.huawei.com/enterprise/en/doc/EDOC1000173015?section=j006>. Cited in page 33.

HYUN, J.; HONG, J. W.-K. Knowledge-defined networking using in-band network telemetry. In: IEEE. **2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)**. [S.l.], 2017. p. 54–57. Cited in page 36.

JEYAKUMAR, V. et al. Millions of little minions: Using packets for low latency network programming and visibility. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 44, n. 4, p. 3–14, 2014. Cited 3 times in the pages 23, 35, and 39.

Juniper. 2018. Disponível em: <https://www.juniper.net/documentation/en_US/junos/topics/concept/junos-telemetry-interface-oveview.html>. Cited in page 33.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: **ACM SIGCOMM**. [S.l.: s.n.], 2015. Cited in page 33.

Kreutz, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, v. 103, n. 1, p. 14–76, 2015. Cited in page 27.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: **Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks**. [S.l.: s.n.], 2010. p. 1–6. Cited in page 33.

LIN, Y. et al. Netview: Towards on-demand network-wide telemetry in the data center. **Computer Networks**, Elsevier, p. 107386, 2020. Cited in page 37.

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 6 times in the pages 23, 24, 25, 37, 39, and 53.

LUCKIE, M. J.; MCGREGOR, A. J.; BRAUN, H.-W. Towards improving packet probing techniques. In: **Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement**. [S.l.: s.n.], 2001. p. 145–150. Cited in page 32.

MA, L. et al. Inferring link metrics from end-to-end path measurements: Identifiability and monitor placement. **IEEE/ACM Transactions on Networking**, IEEE, v. 22, n. 4, p. 1351–1368, 2014. Cited in page 39.

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, 25, 37, 39, and 42.

MEDINA, A. et al. Brite: an approach to universal topology generation. In: **IEEE MASCOTS 2001**. [S.l.: s.n.], 2001. p. 346–353. ISSN 1526-7639. Cited 2 times in the pages 47 and 58.

MESTRES, A. et al. Knowledge-defined networking. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 47, n. 3, p. 2–10, set. 2017. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/3138808.3138810>. Cited in page 36.

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 24, 25, 37, 39, 46, 47, 53, and 58.

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 27.

PUTINA, A. et al. Telemetry-based stream-learning of bgp anomalies. In: **Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks**. New York, NY, USA: ACM, 2018. (Big-DAMA '18), p. 15–20. ISBN 978-1-4503-5904-7. Disponível em: <http://doi.acm.org/10.1145/3229607.3229611>. Cited in page 33.

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 in page 35.

SIVARAMAN, V. et al. Heavy-hitter detection entirely in the data plane. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2017. (SOSR '17), p. 164–176. ISBN 9781450349475. Disponível em: <https://doi.org/10.1145/3050220.3063772>. Cited in page 35.

TAMMANA, P.; AGARWAL, R.; LEE, M. Simplifying datacenter network debugging with pathdump. In: **12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)**. [S.l.: s.n.], 2016. p. 233–248. Cited 2 times in the pages 35 and 39.

TAMMANA, P.; AGARWAL, R.; LEE, M. Distributed network monitoring and debugging with switchpointer. In: **15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)**. [S.l.: s.n.], 2018. p. 453–456. Cited 2 times in the pages 36 and 39.

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 in page 44.

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 38.

TIRKOLAEE, E. B.; MAHDAVI, I.; ESFAHANI, M. M. S. A robust periodic capacitated arc routing problem for urban waste collection considering drivers and crew's working time. **Waste Management**, Elsevier, v. 76, p. 138–146, 2018. Cited in page 38.

TU, N. V. et al. Intcollector: A high-performance collector for in-band network telemetry. In: IEEE. **2018 14th International Conference on Network and Service Management (CNSM)**. [S.l.], 2018. p. 10–18. Cited in page 36.

Van Tu, N.; Hyun, J.; Hong, J. W. Towards onos-based sdn monitoring using in-band network telemetry. In: **2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)**. [S.l.: s.n.], 2017. p. 76–81. Cited 2 times in the pages 35 and 36.

WU J. STRASSNER, A. F. Q.; ZHANG., L. **Network Telemetry and Big Data Analysis**. 2016. Disponível em: <https://tools.ietf.org/html/draft-wu-t2trg-network-telemetry-00>. Cited in page 33.

YASEEN, N.; SONCHACK, J.; LIU, V. Synchronized network snapshots. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2018. p. 402–416. Cited in page 36.

YU, C. et al. Software-defined latency monitoring in data center networks. In: SPRINGER. **International Conference on Passive and Active Network Measurement**. [S.l.], 2015. p. 360–372. Cited in page 35.

ZHOU, Y. et al. Flow event telemetry on programmable data plane. In: **Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication**. [S.l.: s.n.], 2020. p. 76–89. Cited 2 times in the pages 37 and 38.

ZHU, Y. et al. Packet-level telemetry in large datacenter networks. In: **Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication**. [S.l.: s.n.], 2015. p. 479–491. Cited 2 times in the pages 35 and 39.

ZIMMERMANN, H. Osi reference model-the iso model of architecture for open systems interconnection. **IEEE Transactions on communications**, IEEE, v. 28, n. 4, p. 425–432, 1980. Cited in page 30.

ANNEX A – ORCHESTRATING IN-BAND DATA PLANE TELEMETRY WITH MACHINE LEARNING

Orchestrating In-Band Data Plane Telemetry with Machine Learning

Rumenigue Hohemberger, Ariel G. Castro, Francisco G. Vogt, Rodrigo B. Mansilha, Arthur F. Lorenzon, Fabio D. Rossi, Marcelo C. Luizelli

***Abstract*—In-band network telemetry (INT) is an emerging network monitoring paradigm. By collecting low-level telemetry items in real time, INT can substantially enhance network-wide visibility - allowing, for example, timely detection problems such as micro-burst. Recent studies have focused on *(i)* developing INT mechanisms to increase network-wide visibility; and *(ii)* to design new monitoring solutions. However, little has been done to coordinate the process of collecting telemetry items in this new paradigm. This is particularly challenging because depending on which network telemetry items are collected, it might degrade network-wide visibility in terms of consistency/freshness. In this letter, we theoretically formalize the In-band Network Telemetry Orchestration Plan Problem and propose a machine learning based orchestration model. Results show that our approach outperforms state-of-the-art heuristics by up a factor of 8x with respect to the number of network anomalies identified, for instance.**

I. Introduction

In-band network telemetry has recently emerged as a promising monitoring alternative to provide higher network-wide visibility to network operators [1]. This finer-grained data plane monitoring mechanism allows to cope with short-lived problems, such as network flow contention, micro-burst, and load imbalance – just to name a few [2], [3]. Yet, data plane telemetry is paramount to the success of real-time network applications with stringent responsiveness requirements, such as virtual/augmented reality [4] and self-driving cars [5]. Much of the progress in the field has been enabled by recent advances in programmable network devices and high-level domain-specific networking description and query languages [6].

By using programmable devices, in-band data plane telemetry allows to collect and encapsulate low-level telemetry information into production network traffic whenever possible. Packets contain header fields that are interpreted as telemetry instructions by network devices. These instructions instrument these devices to collect and write into the packet network states [1]. This telemetry information might encompass switch-internal states (*e.g.*, queue occupancy and switchID) and network performance metrics (*e.g.*, data plane processing time and latency), for instance. In the process of in-band telemetry, the collected information is carried into a packet along its routing path and, at some point in the network, it is extracted and reported to a monitoring application. Then, the monitoring applications process and eventually react to spurious networks events.

Recent approaches [1], [2], [3], [7], [8] have striven to deliver the best out of the in-band telemetry to improve network-wide visibility. PathDump [7] and SwitchPointer [3] combines in-network programmability and the available end-host resources to collect and monitor telemetry data in order to debug networks events. More recently, Liu et al.[1] and

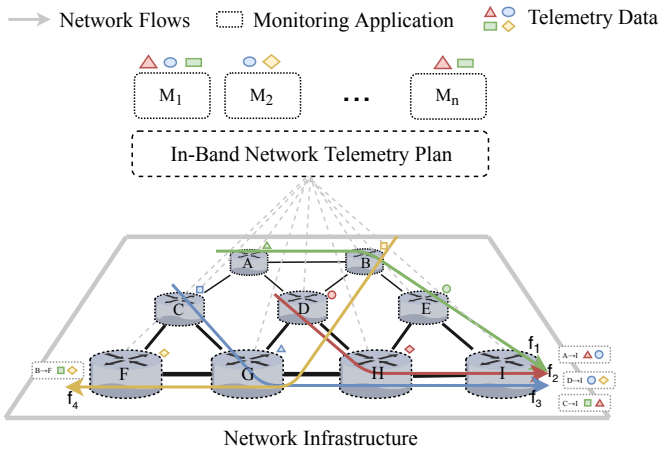


Fig. 1. Example of in-band network telemetry planning. The example illustrates a snapshot where four active network flows (f_1, f_2, f_3, f_4) collect telemetry data from forwarding devices. Telemetry data are then sent to monitoring applications according to their needs.

proposed NetVision, an attempt to provide network telemetry as a service, while Marques et al. [9] provided the first effort towards in-band network orchestration. Despite these efforts to make real-time in-band telemetry a reality in programmable networks, little has yet been done to dynamically coordinate how to collect network information in this new paradigm. This is particularly challenging mainly for two reasons. First, depending on which network telemetry items are collected, it might degrade network-wide visibility in terms of coverage, consistency, and freshness [9]. Second, depending on how network telemetry is collected, it might impact the network monitoring application's performance.

In this letter, we introduce the In-band Network Telemetry Orchestration Plan Problem as a machine learning aided optimization model. The main idea consists of dynamically (and wisely) guiding the in-band data acquisition by means of a learning mechanism. To tackle this problem, we theoretically formalize it as a Mixed Integer Linear Programming (MILP) model. The model can be seen as a generalization of the well-known Bin Packing problem [10] and, therefore, it is an NP-hard problem. Despite the scalability limitations on solving NP-hard problems, this exact formulation represents an optimal bound for future in-band network telemetry approximations. To the best of our knowledge, this is the first attempt to formulate this problem. Results show that the proposed model outperforms [9] by up a factor of 8x with respect to the number of network anomalies identified, increasing

the number of telemetry data collected, and maintaining the consistency and freshness of network visibility.

II. The In-band Network Telemetry Orchestration Plan Problem

A. Problem Overview

The in-band network telemetry consists of embedding telemetry information into production flow packets. Fig. 1 illustrates a given network infrastructure with four active network flows (namely, f_1, f_2, f_3, and f_4) being routed through a set of forwarding devices – ranging from A to I. Network flow f_1 can collect telemetry items from its forwarding devices $\{A, B, E, I\}$. On top of the network, there exists a set of specialized monitoring application $\{M_1, M_2, ..., M_n\}$. These applications are in charge of making inference about specific network problems or (mis)behaviors (*e.g.*, identify network congestion or malicious network attacks) and eventually react to them, demanding different pieces of information from the infrastructure. For instance, let us consider that application M_1 aims to predict and identify (transient) network congestion. It requires network flows to collect from forwarding devices the number of active network flows ($\triangle$), queue occupancy ($\bigcirc$), and processing time directly from data planes ($\square$). In turn, M_2 identifies SYN flood attacks. For that, it requires to constantly monitor the number of active network flows ($\triangle$), and the number of incomplete TCP handshakes ($\Diamond$). Observe that these monitoring applications are not limited to the given examples and may demand different subsets of telemetry items (coverage) to be continuously collected in a given rate (freshness) so that the inference accuracy can be met over time.

The In-band Network Telemetry Plan Problem consists of dynamically orchestrating the process of collecting network information to maximize network monitoring applications performance without, however, penalizing production network flows. The problem is far from being trivially solved. This is due to *(i)* monitoring applications might have completely different requirements – which implies that some telemetry items might be collected more frequently than others; and *(ii)* network packets have limited unused resources (in general, up to the MTU data link limit) – and, therefore, it is unfeasible to collect all items within the same time unit. To illustrate how challenging this problem is, consider the scenario from Fig. 1. Let us assume network flows can collect and carry a maximum fixed amount of telemetry items at a specific time unit (*e.g.*, two items at a time unit). There are a few alternatives to orchestrate the acquisition of INT items. First, a naïve solution consists of collecting all telemetry items in every single time unit. For building such a solution, we relax the assumption that network packets/frames are space-bounded. A second (and more realistic) strategy takes into account an upper-bound limit on the amount of collected telemetry items. Therefore, the number of telemetry items collected by network flows are rather constrained than the naïve strategy. In fact, it turns out that introducing this set of constraint makes the problem NP-hard (as it is a generalization of the well-studied Bin Packing problem [10]). In spite of having hard constraints on the number of items to be collected in a single time unit, a relaxed solution (with respect to time) might be built by round-robining the collection of telemetry items over time. Although this might represent a feasible (but relaxed) solution over time, it misses one fundamental aspect of network telemetry, namely to identify which telemetry items are in fact important to be collected. This information is rather important in order to ensure network-wide visibility of the network infrastructure accurately.

B. Model description and notation

The optimization model we propose to solve the orchestration problem defined above considers a physical network infrastructure $G = (D, L)$, a set of active network flows F, a set of telemetry items V, and a set of monitoring applications M. Set D in network G represents programmable forwarding devices $D = \{1, ..., |D|\}$, while set L links interconnecting pair of devices $(d_1, d_2) \in (D \times D)$. Each forwarding device $d \in D$ is able to embed a subset of items $V_d \subseteq V$ into packets of flow $f \in F$. Each telemetry item $v \in V$ has its size defined by function $S : V \to \mathbb{N}^+$.

Network flows F are used to collect real-time telemetry data from forwarding devices D. A flow $f \in F$ has two endpoints (*i.e.*, ingress and egress forwarding devices) and is routed through the network infrastructure G using a simple path $\mathcal{P}t$. We denote the path taken by flow f as function $\mathcal{P}t : F \to \{D_1 \times ... \times D_{|D|}\}$. Network flows F are encapsulated in a forwarding protocol (*e.g.*, NSH[1], IPv4). Therefore, the amount of available space to embed telemetry items in packets is bounded by a constant $K_f \in \mathbb{N}^+$. Observe that a single packet of flow $f \in F$ can collect at most $\sum_{\forall d \in \mathcal{P}t(f)} V_d$ telemetry items from the network infrastructure G at a given time frame. Unless K_f assumes a sufficient large value (*i.e.*, $K_f \geq \sum_{\forall d \in \mathcal{P}t(f)} |V_d|$), it is not possible to collect all items from all forwarding devices in the routed path $\mathcal{P}t(f)$.

Monitoring application $m \in M$ requires a subset of telemetry items $R_m \subseteq V$ to operate properly. These telemetry items might have spatial and temporal dependencies. We say two or more telemetry items have spatial dependency *iff* they must be collected from the same forwarding device $d \in D$. Spatial dependencies are represented by the set of sets $R^s_m \subseteq \mathscr{P}(R_m)$. In turn, we say that a set of telemetry items have temporal dependency *iff* they must be collected within a given deadline. Temporal dependencies are represented by the set of sets $R^t_m \subseteq R^s_m$, and the required deadline is expressed as a function $T^f : R^t_m \to \mathbb{N}^+$. The model keeps track of the last time unit an item $P \in R^t_m$ was collected by means of a function $H^f : R^t_m \to \mathbb{N}^+$. When $(\forall P \in R^t_m) : H^f(P) > T^f(P)$, item P is out of date (*i.e.*, the deadline has expired).

C. Naïve Orchestration Model

Given a network infrastructure G, a set of network flows F, a set of monitoring applications M, a set of telemetry items V and a constant K_f, the optimization problem seeks a feasible solution that maximizes the number of spatial and

[1]https://tools.ietf.org/html/rfc8300

temporal dependencies. The model output is denoted by a 3-tuple $\chi = \{Y, S^b, T^b\}$. Variables from $Y = \{\, y_{d,v,f}\,, \forall\, d \in D, v \in V, f \in F\}$ indicate that a forwarding device d embed telemetry item v into packets of network flow f. Variables from $S^b = \{\, s^b_{m,d,P}\,, \forall m \in M, d \in D, P \in R^s_m\}$ and $T^b = \{\, t^b_{m,P}\,, \forall m \in M, P \in R^t_m\}$ are used to keep track of spatial and temporal dependencies satisfied by the model. Next, we describe the MILP formulation for this orchestration problem.

$$\textbf{Maximize} \qquad \sum_{m \in M} \sum_{p \in P \in R^s_m} \sum_{d \in D} s^b_{m,d,p} + t^b_{m,p} \tag{1}$$

Subject to:

$$\sum_{d \in \mathcal{P}t(f)} \sum_{v \in V_d} y_{d,v,f} \cdot S(v) \leq K_f \qquad \forall f \in F \tag{2}$$

$$\sum_{f \in F} y_{d,v,f} \leq 1 \qquad \forall d \in D, v \in V_d, \tag{3}$$

$$s_{m,d,p} = \sum_{v \in P} \sum_{f \in F} y_{d,v,f} \qquad \forall m \in M, p \in P \in R^s_m, d \in D \tag{4}$$

$$t_{m,p} = \sum_{v \in P} \sum_{d \in D} \sum_{f \in F} y_{d,v,f} \qquad \forall p \in P \in R^t_m : H^f(p) > T^f(p) \tag{5}$$

$$s^b_{m,d,p} \leq \frac{s_{m,d,p}}{|P|} \qquad \forall m \in M, p \in P \in R^s_m, d \in D \tag{6}$$

$$t^b_{m,p} \cdot \leq \frac{t_{m,p}}{|P|} \qquad \forall m \in M, p \in P \in R^t_m \tag{7}$$

$$y_{d,v,f} \in \{0,1\} \qquad \forall d \in D, v \in V, f \in F \tag{8}$$

$$s_{m,d,p} \in \mathbb{N}^+ \qquad \forall m \in M, d \in D, p \in P \in R^s_m \tag{9}$$

$$s^b_{m,d,p} \in \{0,1\} \qquad \forall m \in M, d \in D, p \in P \in R^s_m \tag{10}$$

$$t_{m,p} \in \mathbb{N}^+ \qquad \forall m \in M, p \in P \in R^t_m \tag{11}$$

$$t^b_{m,p} \in \{0,1\} \qquad \forall m \in M, p \in P \in R^t_m \tag{12}$$

Constraint set (2) ensures that a network flow $f \in F$ does not exceed its capacity (*i.e.*, K_f). Constraint set (3) ensures that a single telemetry item is collected at most by a single network flow $f \in F$ in a given device d. Constraints sets (4) and (6) aim to account whether or not a spatial dependency is met. Given a spatial dependency, constraint set (4) count the number of telemetry items collected in a device d, while constraint set (6) verify whether or not the dependency is met (*i.e.*, checking if all items were collected). Similarly, constraints sets (5) and (7) count the number of temporal dependencies that are satisfied. Last, constraints sets (8)-(12) define the domains of output variables.

D. Machine Learning based Orchestration Model

The orchestration model proposed in the previous subsection is able to maximize the number of collected telemetry items. However, it still misses the fundamental orchestration question: *which telemetry items are in fact important to be collected?* Similar models in the literature (*e.g.* [11]) handle the importance of a given item by means of arbitrary weights, which needs to be manually fine-tuning from time to time, in a static fashion. To fill in this gap, we plug in our model a learning mechanism in order to dynamically (and wisely) instrument the collection of telemetry items from the network.

The importance of a telemetry item depends on the requirements of monitoring applications, which changes dynamically over time according to the network behavior. We assume that in a given time unit t, the proposed learning model knows a subset of telemetry items, represented by $V^t \subseteq \sum_{t-W}^{t} \sum_{m \in M} R^s_m$, where $W \in \mathbb{N}^+$ represents a given time window. An element $P \in V^t$ represents a $|P|$-dimensional space tuple $P = (v_1, v_2, \cdots v_{|P|})$, where each dimension represents a telemetry information collected from the infrastructure, *i.e.* $v_{|S|} \in \mathbb{R}^+$. Given a set V^t of telemetry data collected within the last W time units, we model the In-Band Network Telemetry Plan layer (as shown in Fig. 1) as multiple *online* clustering models. The idea consists of clustering network behaviors based on telemetry items that share the same dependencies. The orchestration model, therefore, can coordinate which telemetry items are important to be collected next (*e.g.*, telemetry items observing unusual behaviors). Each cluster keeps track of items having the same $|P|$ dimensions. We partition $V^t = \{V^{t,1}, V^{t,2}, ..., V^{t,\mathscr{P}(V^t)}\}$ so that each subset has items of the same $|P|$ dimensions. We then cluster subsets $V^{t,i} \in V^t$ into K_i exclusive partitions, *i.e.* $(\forall i \in \{1, ..., \mathscr{P}(V^t)\}) : V^{t,i}_1, V^{t,i}_2, ..., V^{t,i}_{K_i}$. The centroid of a subset $V^{t,i}_k$ is defined as $C(V^{t,i}_k)$. An element $P \in V^{t,i}$ is assigned to a cluster $V^{t,i}_k$ if the distance of v to $(V^{t,i}_k)$ is minimal. We omit the definition of the clustering problem (as an optimization problem) due to space constraints. However, it is important to mention that clustering problems are well-known NP-hard problems [12] – even for planar graphs.

Next, we define how the model keeps cluster consistency and freshness over time. In order to capture the network dynamics, we propose a fading function that estimates how long the model keeps telemetry data in it. Telemetry data measured with higher variance in a time frame W tends to be spanned through a lower number of time units than those with lower variance. The variance of a telemetry item (or a P-tuple) is defined according to its dispersion index. We denote the dispersion index as $I(v) = \frac{\sigma^2_v}{\mu}$. The number of time units a telemetry item is kept in V^t is defined according to a piecewise function $T(P) : V^t \rightarrow \mathbb{N}^+$, defined as follow:

$$T(P) = \begin{cases} W & I(P) \leq 1 \\ W \cdot (1 - \rho) & I(P) > 1 \end{cases} \tag{13}$$

Telemetry data points $P \in V^t$ that are under-dispersed (i.e, $I(P) \leq 1$) are kept in the clustering for at least W time units. In turn, over-dispersed data points (*i.e.*, $I(P) > 1$) tends to be in the cluster for fewer time units (W is decayed as with respect to $\rho \in [0, 1]$). Observe that this approach allows the model to capture transient events (*e.g.*, short-time queue occupancy) and also persist events the endure to longer time units (*e.g.*, physical disruption in a forwarding device). We denote by $V^+ \subseteq V^t$ and $V^- \subseteq V^t$ the subset of telemetry items that are over- and under-dispersed, respectively. Over-

dispersed items with spatial dependencies are represented by set $R_m^{+s} = \{P \in R_m^{+s} | P \in R_m^s \wedge (P \cap V^+ \neq \emptyset)\}$. Similarly, under-dispersed items with spatial dependencies by set $R_m^{-s} = \{P \in R_m^{-s} | P \in R_m^s \wedge (P \cap V^- \neq \emptyset)\}$. We are interested in optimizing how we evolve over time the information we know about the infrastructure G, maintaining $V^{t,i} \in V^t$ clusters over time. We assume most of the time, the value of telemetry points evolves smoothly. Abrupt changes happen to the network state, but with low probability. This assumption is realistic to in-band network telemetry. Our main goal is to reduce the acquisition of irrelevant telemetry data and, at the same time, maintaining the accuracy of acquired telemetry data. The machine learning based orchestration model replaces Equation (1) by Equation (14).

$$\textbf{Maximize} \quad \alpha \cdot \sum_{m \in M} \sum_{p \in P \in R_m^+} \sum_{d \in D} s_{m,d,p}^b + t_{m,p}^b + \beta \cdot \sum_{m \in M} \sum_{p \in P \in R_m^-} \sum_{d \in D} s_{m,d,p}^b + t_{m,p}^b \qquad (14)$$

The combined objective function tries to wisely maximize the number of collected over-dispersed and under-dispersed items from G. Observe that the objective function prioritizes the collection of over- and under-dispersed items according to parameters α and β. That is explained because, in regular network operation, the dispersion index tends to lower while there is not a suspicious event. On the event of a suspicious event, the index would tend to increase and therefore there will high the chances to be collected.

III. Evaluation

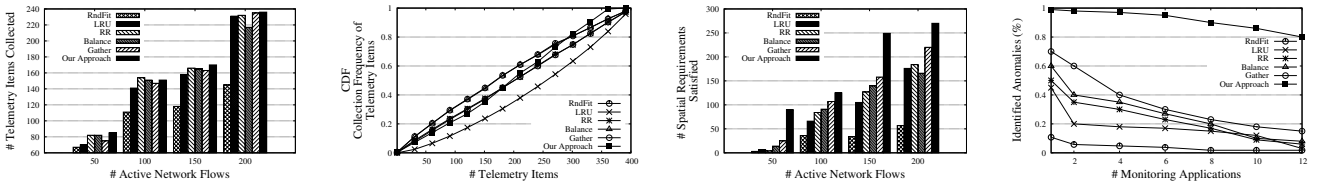
Setup. We ran the proposed model using IBM *CPLEX Optimization Studio* 12.9 to obtain optimum solutions and implemented the online version of K-Means [13] using Java language to dynamically ponderate the importance of telemetry items[2]. Experiments were performed on a machine with four Intel Xeon E5-2670 processors and 56 GB of RAM, using the Ubuntu Server 11.10. We considered different physical network instances that were generated with Brite [14], following the Barabasi-Albert model [15]. We used physical infrastructures consisting of 50 forwarding devices and, on average, 200 physical links. On top of each infrastructure instance, there was a set F of active flows. We varied the amount of network flows from 50 to 200. Each flow $f \in F$ interconnected two random endpoints in the infrastructure and, was routed using the shortest path algorithm. We consider IP-based network flows and, therefore, we vary uniformly the constant K_f from 10-30 Bytes. Further, we assume that forwarding devices have 8 possible telemetry items to export[3], varying $S(v)$ from 2 to 20 Bytes [16]. We consider a set of monitoring applications, ranging from 2 to 12. Each application requires at most 4 spatial dependencies, each having at least 2 items (*i.e.*, switch ID + monitored metric) and at most 4. We consider $\alpha = 1$, $\beta = 1$, $W = 5$, and $\rho = 0.5$. These parameters can be fine-adjusted to prioritizes the collection of spatial dependencies over temporal ones and to adjust the fading functions. Our experiment ran in 200 time units. At each time unit, our model and algorithms are run. Each experiment is repeated 30 times to ensure a confidence level of 95% or higher.

Baseline. We compare our proposed model against *(i)* two state-of-the-art orchestration heuristics proposed by [9], namely `Gather` and `Balance`, and *(ii)* three heuristics: Round-Robin (`RR`), Least-Recently Collected (`LRC`), Random-Fit (`RndFit`). The heuristic `RR` strives to assign telemetry items to network flows in a round-robin way. `LRC` prioritizes the collection of telemetry items that have not been collected recently. Finally, `RndFit` chooses randomly telemetry items and tries to assign to random network flow.

Results. We focus our analysis on two key aspects of network monitoring: *(i)* network-wide visibility (in terms of coverage), and *(ii)* network monitoring applications performance. For that, we evaluate network telemetry coverage (*i.e.*, the number of telemetry items collected), a number of spatial dependencies satisfied, and the ability to identify network anomalies. Fig. 2(a) illustrates the average amount of telemetry items collected over time for an increasing number of active network flows. We observe that *(i)* number of network flows directly impacts on the network visibility (*i.e.*, the more active network flows, the higher the network coverage) and *(ii)* the proposed orchestration model on average outperform evaluated heuristics by up a factor of 1.7x when comparing to `RndFit`. This behavior is due to the ability to optimally assign telemetry items to network flows. In contrast, evaluated heuristics make local decisions on which item assign (*e.g.*, least recently collected) to each flow, leading to sub-optimal solutions. On average, our proposed orchestration model covers 60% of telemetry items in each time unit (considering 400 telemetry items in total). Next, we evaluate the number of times (frequency) telemetry items are collected, illustrated in Fig. 2(b) by means of a CDF (Cumulative Distribution Function). Despite showing in Fig. 2(a) that the number of items collected by our model and by the baselines are similar, we observe in Fig 2(b) that each method prioritizes the collections of different subsets of telemetry items. This difference occurs due to *(i)* the heuristic assignment policy and *(ii)* the number of active network flows available to assign telemetry items. The latter point can be explained by the presence of highly interconnected forwarding devices in the network topologies – in which telemetry items are potentially collected more frequently than others. This fact is further exacerbated as active network flows are routed using shortest paths. Observe that our orchestration model, differently from evaluated heuristics, prioritizes the collection of a subset of telemetry items – on average, 25% of telemetry items are collected more intensely (more than 80% of the time units). In contrast, up to 40% of telemetry items are collected less frequently (less than 40% of the time units). Fig. 2(c) illustrates the average number of spatial dependencies satisfied over time. As observed, our model outperforms evaluated heuristics by up a factor of 2.5x (for 150 active network flows). The number of spatial dependencies satisfied is essential input information to

[2]Reproducible material available on https://github.com/mcluizelli/comml-int19

[3]In-band Network Telemetry: https://p4.org/assets/INT-current-spec.pdf



(a) Telemetry Items collected.

(b) Number of times telemetry items are collected (200 network flows).

(c) Spatial requirements satisfied.

(d) Anomaly events detected (200 network flows).

Fig. 2. In-band telemetry orchestration performance metrics.

learning models, which required coupled telemetry information – otherwise, telemetry data are useless for the learning process. Last, Fig. 2(d) illustrates the average number of network anomalies identified. For this experiment, we consider an increasing number of monitoring applications running in parallel and, therefore increasing the need for a higher number of spatial dependencies. Further, we assume that telemetry items values follow a normal distribution, defined by an average and standard deviation (*e.g.*, queue occupancy given by the average of 100 and deviation of 20). We then inject anomalies to these items (*i.e.*, values out from the expected range), lasting for 5 times units (on average) on (at most) 10% of forwarding devices. When the number of monitoring applications is relatively small (up to 2 monitoring applications), machine learning mechanism can identify up to 50-60% of network anomalies with the input provided by telemetry data collected by the evaluated approaches. However, as the number of monitoring applications increases (and, therefore, the number of spatial dependencies), the ability to identify network anomalies decreases considerably, reaching up to 5-10% of identified anomalies. The reason is correlated with the frequency with that telemetry items are collected from the network. Baseline algorithms strive to collect telemetry items in a fairly way (e.g., round-robin) – which lowers the overall frequency that telemetry items are collected and therefore higher the chances to miss the collection of anomalous events. In contrast, our model can cope with the increasing number of monitoring applications. The input provided by our model can feed machine learning mechanisms models in order to identify up to 97% of network anomalies (up to 4 monitoring applications). From 4-12 network monitoring applications, the accuracy of identifying network anomalies decreases by 20% (*i.e.*, 8x higher than the baselines).

we intend to investigate the design of dynamic and scalable heuristic/approximation algorithms, derive upper-bounds to the proposed model and to integrate to commercial offerings.

IV. Final Remarks

In this letter, we formalize the In-band Network Telemetry Orchestration Plan by means of a MILP model. The main idea consists of dynamically guiding the acquisition process of telemetry data using learning mechanisms. We showed that our proposed model can effectively collect the most important telemetry items and provide accurate network-wide visibility to monitoring applications. Our model outperforms state-of-the-art heuristics by a factor of 2.5x with respect to the number of spatial dependencies satisfied and by a factor of 8x when comparing the number anomalies identified. As future work,

References

[1] Z. Liu, J. Bi, Y. Zhou, Y. Wang, and Y. Lin, "Netvision: Towards network telemetry as a service," in *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, Sep. 2018, pp. 247–248.

[2] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "007: Democratically finding the cause of packet drops," in *15th USENIX NSDI 18*. Renton, WA: USENIX Association, 2018, pp. 419–435.

[3] P. Tammana, R. Agarwal, and M. Lee, "Distributed network monitoring and debugging with switchpointer," in *15th USENIX NSDI 18*, Renton, WA, 2018, pp. 453–456.

[4] R. I. T. da Costa Filho, M. C. Luizelli, M. T. Vega, J. van der Hooft, S. Petrangeli, T. Wauters, F. De Turck, and L. P. Gaspary, "Predicting the performance of virtual reality video streaming in mobile networks," in *ACM MMSys '18*. New York, NY, USA: ACM, 2018, pp. 270–283.

[5] G. Ananthanarayanan, P. Bahl, P. Bodk, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[6] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM 14*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[7] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *12th USENIX OSDI 16)*, Savannah, GA, 2016, pp. 233–248.

[8] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *ACM SIGCOMM 15*. New York, NY, USA: ACM, 2015, pp. 479–491.

[9] J. A. Marques, M. C. Luizelli, R. I. T. Da Costa, and L. P. Gaspary, "An optimization-based approach for efficient network monitoring using in-band network telemetry," *Journal of Internet Services and Applications*, vol. 10, no. 1, p. 16, Jun 2019.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[11] R. Cohen, L. Katzir, and D. Raz, "An efficient approximation for the generalized assignment problem," *Information Processing Letters*, vol. 100, no. 4, pp. 162 – 166, 2006.

[12] M. Mahajan, P. Nimbhorkar, and K. Varadarajan, "The planar k-means problem is np-hard," *Theoretical Computer Science*, vol. 442, pp. 13 – 21, 2012.

[13] E. Liberty, R. Sriharsha, and M. Sviridenko, "An algorithm for online k-means clustering," in *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 81–89.

[14] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: an approach to universal topology generation," in *Proceedings of the IEEE MASCOTS*, Aug 2001, pp. 346–353.

[15] R. Albert and A.-L. Barabási, "Topology of evolving networks: Local events and universality," *Physical Review Letters*, vol. 85, pp. 5234 – 5237, Dec 2000.

[16] T. Pan, E. Song, Z. Bian, X. Lin, X. Peng, J. Zhang, T. Huang, B. Liu, and Y. Liu, "Int-path: Towards optimal path planning for in-band network-wide telemetry," in *IEEE INFOCOM 19*, Apr 2019, pp. 1–9.

ANNEX B – PATCHER: TOWARDS FAULT-TOLERANT PROBING PLANNING FOR IN-BAND NETWORK TELEMETRY

Patcher: Towards Fault-Tolerant Probing Planning for In-band Network Telemetry

Ariel G. Castro*, Victor H. S. Lopes*, Francisco G. Vogt*, Fabio D. Rossi+,
Arthur F. Lorenzon*, Marcelo C. Luizelli*
*Federal University of Pampa (UNIPAMPA)
+Federal Institute Farroupilha (IFFAR)

***Abstract*—In-band Network Telemetry (INT) is a recent network monitoring approach. Despite the existence of a few initiatives to orchestrate the collection of in-band network statistics, little has yet been done to coordinate active INT probes to collect network information efficiently while considering the possibility of device failures (e.g., power failure, hardware failure). In this paper, we introduce *Patcher* – fault-tolerant probing planning for INT. It reconstructs probing cycles affected by failure nodes and optimizes them while ensuring all non-affected links/statistics are still being traversed/collected. Results show that *Patcher* reduces the number of probe cycles needed up to 5.5x compared to a state-of-the-art solution, while not increasing the INT collectors' load.**

I. Introduction

In-band Network Telemetry (INT) is a paramount network monitoring mechanism enabled by programmable network infrastructure [1]. INT provides the instrument and collects low-level (and per-packet) data plane statistics in (near) real-time, boosting the design of tailored monitoring applications. To do this, INT allows encapsulating switch-internal states (a.k.a. data plane telemetry) by programmable network devices (e.g., switches and SmartNICs) into network traffic packets (for example, queue occupancy, data plane processing time, or aggregate statistics).

INT data can be embedded into either active network flows or specially-crafted probing packets. These packets carry specific telemetry instructions interpreted by network devices, and setting forwarding devices to collect the required network states [2]. Therefore, at some predefined point in the network, the collected telemetry data is extracted and reported to an INT collector. Based on the above, INT has been applied to many networking use cases, including congestion control [3], path tracing [4], fast reroute [5] – to name a few applications. However, providing fault-tolerant probing cycles at the design is particularly challenging. In case a network link or forwarding device fails, all (or a substantial part) of the INT monitoring mechanism that relies on that device is compromised, affecting directly active monitoring applications.

Despite a few research efforts towards the orchestration of INT [1], [2], [6], [7], [8], [9], little has been done to provide fault-tolerant orchestration of INT mechanisms in programmable network infrastructures. Current strategies have relied on their strategies either on *Euler Circuits* [1], [2] or on actual routing paths [7], [8], [9] to instrument the forwarding of probes on the embedding of INT data into network packets. In this paper, we introduce *Patcher* – fault-tolerant probing planning for in-band network telemetry. In the event of faulty forwarding devices, *Patcher* efficiently rebuild and fix monitoring cycles by applying "*patches*" to ensure that all links are visited and the required INT data is collected correctly. *Patcher* is the first effort to provide fault-tolerant INT probing cycles. It is worth mentioning that solutions supplied by *Patcher* can be either used by the control plane (to re-actively fix monitoring cycles) or directly by the data plane (to proactively sets alternative INT routing strategies). To tackle this problem, we theoretically formalize the Fault-Tolerant Probing Planning for In-band Network Telemetry. The model can be seen as a generalization of the Capacitated Arc Routing problem [10] and, therefore, it is an NP-hard problem. To solve this problem efficiently, we introduce a heuristic that wisely finds a high-quality solution. To the best of our knowledge, this is the first attempt to formally define and solve this problem. Results show that *Patcher* outperforms the state-of-the-art solution [1] by a factor of 5.5x related to the number of probes, at the same time that makes better usage of available resources and decreases the transmission overhead to INT collectors.

The remainder of this paper is organized as follows. In Section 2, we discuss related work in the area of in-band network telemetry. In Section 3, we formalize the fault-tolerant probing planning for INT. In Section 4, we introduce our heuristic approach to solve it. In Section 5, we present and discuss the results of an evaluation of the heuristic. Last, in Section 6 we conclude the paper with final remarks and perspectives for future work.

II. Related Work

To the best of our knowledge, the in-band network telemetry plan problem has not been investigated before the inception of programmable data planes. We discuss the most prominent studies related to network telemetry.

A. In-band Network Telemetry Monitoring Mechanisms

Recent advances in forwarding devices have enabled to push telemetry information continuously (i.e., via streaming) to data collectors – known as Model-Driven Telemetry (MDT) [11]. In this context, Putina et al. [12] proposed a mechanism for real-time detection of BGP anomalies, relying on machine-learning techniques and MDT-based telemetry data streaming.

978-1-7281-8903-1/20/$31.00 ©2020 IEEE

Other studies have focused on the concept of in-band network telemetry (INT) [13]. Mazières et al. [14] introduced the seminal work on in-band telemetry. They proposed the concept of a tiny packet program (TPP). In turn, Everflow [15] extended INT concept by exploring the "match-and-mirror" functionality of commodity switches. Everflow uses the INT concept to filter packets that satisfy given patterns (i.e., *matching*) and send (*mirroring*) them to multiple data analyzers, which then can send "guided probes" to investigate potential faults.

Gupta et al. [16] designed SONATA, a high-level interface to express telemetry queries. Based on programmable data plane constraints, monitoring queries are partitioned and processed in multiple devices, ensuring, therefore, that monitoring queries and the packet forwarding still operate at line rate for high traffic volumes and rates. Omnimon [17] takes a step further to achieve resource efficiency and full accuracy telemetry for data center networks (DCN). It splits the execution among different entities (i.e., end-hosts, switches, controller) to monitor flows across the entire network. Similarly, Concerto [18] splits queries between switches to reduce the stream processor's load. The switches execute queries in a best-effort manner, i.e., the queries are executed only if available resources exist on the switches.

In turn, PathDump [19] is a mechanism designed to identify and debug anomalous behaviors in programmable network infrastructure. The approach is based on the route taken by network packets and on the subsequent analysis of it. For that, PathDump keeps track of the packet's route, employing INT instruction in the forwarding devices. SwitchPointer [20] extends PathDump by proposing to collect end-host telemetry information to enhance the debugging capabilities – in addition to in-network telemetry information. In turn, NetSeer [21] leverages programmable switches and NICs to allow operators to troubleshoot network problem anomalies. It selects packets that experience flow events, minimize false-positives (duplication of reported flows events), and aggregate sequential event (e.g., congestion) packets from flow into a single flow event to aid on the location of anomalies (e.g., packet drops) in the network. Other studies [22] have proposed to execute specific telemetry operations (e.g., heavy hitters identification) directly in the data plane. However, telemetry operations are limited by the available capabilities (e.g., memory) in forwarding devices.

B. Orchestration of In-band Network Telemetry

The INT orchestration problem has recently gained attention from academia. Marques et al. [7] and Liu et al. [2] were the first to introduce the problem. Marques et al. [7] propose two heuristic strategies for collecting telemetry data, namely, *concentrate* and *balance*. The proposed heuristics assign network flows to forwarding devices. Liu et al. [2] proposed NetVision, an attempt to provide an architectural design to offer network telemetry as a service. NetVision enhance network visibility by dynamically changing the routing policies applied to network flows. Hohemberger et al. [8] further improved previous solutions by designing a machine learning-based model that wisely choose and collect INT data based on its importance. Their model outperforms state-of-the-art solutions. Yet, Hohemberger et al. [9] introduced a meta-heuristic approach based on Iterated Local Search to scale the resolution of the orchestration problem [8].

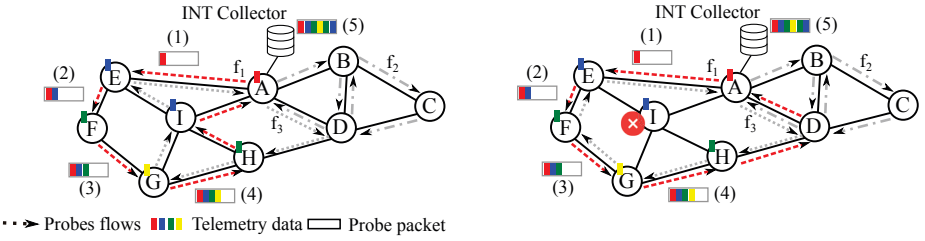
In turn, Pan et al. [1], and Geng et al. [6] have focused on performing network telemetry through active INT-based probing packets. These strategies have relied either on *Euler Circuits* [2], [1] or actual routing paths [6] to instrument the forwarding of probes. Pan et al. [1] propose *INT-path*, a network-wide telemetry system that embeds *source routing* into INT probes and develop an Euler trail-based algorithm to cover the whole network with non-overlapping INT paths. Geng et al. [6] proposes SIMON, a measurement system that reconstructs the network state by collecting key network state variables such as queuing times, link utilization, and queue composition. NICs retrieve data on an edge-based approach and a mesh of probes to reconstruct DCN networks by covering the paths on a per-packet or per-flow basis.

As can be observed, current research efforts related to the *in-band* telemetry are still restricted to mechanisms that mostly utilizes collected telemetry data for new monitoring solutions(e.g., [12], [14], [15], [16], [17], [18]). The studies introduced by [2] and [7] represent the first steps towards the orchestration of in-band network telemetry. This work is a first step in the direction of a fault-tolerant solution for the aforementioned problem. As will be shown later, our proposed approach is able to outperform state-of-the-art, coming up with feasible, high-quality solutions for larger scenarios.

III. Patcher: A Fault-Tolerant Probing Planning for In-band Network Telemetry

A. Problem Overview

The probing planning problem consists of designing probing cycles to cover a network infrastructure in terms of telemetry data and links – that is, collecting data plane telemetry statistics at near real-time using probing packets. By covering the network infrastructure, INT enables building and maintaining an updated network-wide state – which is crucial to large-scale networks. While previous studies have focused on building feasible solutions to the in-band probing planning problem (e.g., [1], [2], [9]), *Patcher* is the first effort to tackle fault-tolerance at design. *Patcher* focuses on building a feasible solution when (multiple) node failures occur. Suppose a given set of programmable devices fail. To keep the in-band network telemetry monitoring alive, the control plane (or the data plane) has to react and reorganize existing probing cycles in order to keep the network-wide visibility timely. *Patcher* is the first effort towards a feasible solution to this problem. Figure 1(b) illustrates the case when a single programmable device fails (i.e., node I). In this case, it affects the monitoring cycles performed by f_1 and f_2. *Patcher* focuses on efficiently rebuilding those cycles to ensure that all links are still being covered, while collecting the required INT data from the data planes. For that, *Patcher* tries to minimize the changes in the current solution by applying "patches" on affected cycles (and, therefore, the name *Patcher*). Figure 1(b) illustrates a



(a) Example of a feasible solution for the probing planning problem.

(b) Example of a feasible solution provided by *Patcher* when network device I is faulty.

Fig. 1. Scenarios with (rightmost) and without (leftmost) failure nodes

feasible solution that fixes probing cycle f_1 (affected by the failure), consisting of rerouting the probing cycle through nodes $G \rightarrow H \rightarrow D \rightarrow A$, instead of $G \rightarrow I \rightarrow A$ (where node I has failed).

Re-optimizing probing cycles is an NP-hard problem and, therefore, it is not trivially solved. The hardness of this problem is due to (i) existing space constraints on probing packets (in the figure, they support up to six telemetry data); and (ii) ensuring cycle connectivity without any sub-tour (i.e., nodes on a cycle are all connected to each other). A simple and naive solution consists of generating all probing cycles on the event of node failures. Although it represents a feasible solution, it impacts the operation of active (and unaffected) monitoring cycles as all probing cycles would need to be reprogrammed by the control plane. *Patcher*, in contrast, focuses only on affected cycles and, therefore, can be efficiently used by the control plane to instrument probing cycles on the event of failures, or directly by the data plane as a fast-failover mechanism.

B. Model description and notation

Input. The optimization model considers as input a physical network infrastructure $G = (D, L)$, a set of telemetry items V, a set of active probing cycles P, and a set of faulty nodes $D^* \subseteq D$. Set D in network G represents programmable devices $D = \{1, ..., |D|\}$, while set L consists of unidirectional links interconnecting pair of devices $(i, j) \in (D \times D)$. There exists a set of telemetry items V available in the network G. Each device $i \in D$ is able to embed a subset of items $V_i \subseteq V$ into a probing packet. Each telemetry item $v \in V$ has its size defined by function $S : V \rightarrow \mathbb{N}^+$. Telemetry items are collected by probing cycles $P = \{1, ..., |P|\}$. We denote the routing cycle followed by probing $p \in P$ as function $\mathcal{C} : P \rightarrow \{D_1 \times ... \times D_{|D|}\}$. Probing cycles $p \in P$ can collect telemetry items from forwarding devices $i \in \mathcal{C}(p)$. The capacity of probing packets (e.g., the MTU) is upper bounded by a given constant, defined as $U : P \rightarrow \mathbb{N}^+$. The subset of telemetry data collected by probing cycle $p \in P$ is represented by a set of pairs $(i, v) : i \in D, v \in V_i$ and is given by the function $\mathcal{T} : P \rightarrow \{(D \times V) \times (D \times V), ..., (D \times V)\}$. For simplicity, a pair $(i, v) \in \mathcal{T}$ is indexed as $_{[1]}$ and $_{[2]}$ to refer to the first and second element, respectively.

Constraints. Next, we describe the main feasibility constraints related to the problem. The problem is subject to (i) network telemetry coverage constraints; (ii) network link coverage constraints; (iii) cycle capacity; and (iv) cycle connectivity constraints.

(i) Network telemetry coverage: all network telemetry statistics available on data plane devices $V_i : (\forall i \in D)$ need to be collected by at least on active probing cycle P. For each network device $i \in D$, we keep track of collected telemetry items by existing probing cycles. Formally,

$$\Big|\Big(\bigcup_{\substack{(p \in P, (j,v) \in \mathcal{T}(p) \\ | \; i=j)}} (j, v)_{[2]}\Big) \bigcap V_i\Big| = \big|V_i\big| \qquad : (\forall i \in D) \quad (1)$$

(ii) Network link coverage: Network links L from the network G need to be covered by at least on probing cycle P. For each network link $(k, l) \in E$, we count the occurrences on existing probing cycles P (which should be greater than one). Formally,

$$\Big|\bigcup_{\substack{(p \in P, i=1 \\ | \; \mathcal{C}(p)_{[i]}=k \\ \wedge \mathcal{C}(p)_{[i+1]}=l)}}^{i \leq (|\mathcal{C}(p)|-1)} (\mathcal{C}(p)_{[i]}, \mathcal{C}(p)_{[i+1]})\Big| \geq 1 \qquad : (\forall (k, l) \in L) \quad (2)$$

(iii) Cycle capacity: Probing cycles are upper bounded by $U(p)$ to embed data plane statistics and accounting for used links and nodes. Usually, INT procedures utilize up to 1 byte to store device/link ID. We sum the cycle length (i.e., $|C(p)|$) with the telemetry usage to avoid creating unrealistic, lengthy cycles.

$$\Big(\sum_{(i,v) \in \mathcal{T}(p)} (j, v)_{[2]} \cdot S(v)\Big) + |C(p)| \leq U(p) \qquad : (\forall p \in P) \quad (3)$$

(iv) Cycle connectivity: Probing cycles need to be well constructed. A valid cycle $\mathcal{C} : P \rightarrow \{D_1 \times ... \times D_{|D|}\}$ is the one that starts and ends at the same device (Equation 4), while all devices $i \in \mathcal{C}(p)$ are pairwise strongly connected, i.e., any pair of devices in $\mathcal{C}(p)$ are reachable to each other. To describe this property, we recall a auxiliary function $\delta : (P \times D \times D) \rightarrow \{true, false\}$ that returns *true* in case there exists a path between node i and j in probing path P, i.e. $\mathcal{C}(p)_{[i]} \rightarrow ... \rightarrow \mathcal{C}(p)_{[j]}$, where $(\mathcal{C}(p)_{[i]}, \mathcal{C}(p)_{[i+1]}) \in L$. Otherwise, function δ returns *false*.

$$\mathcal{C}(p)_{[1]} = \mathcal{C}(p)_{[|\mathcal{C}(p)|]} \qquad : (\forall p \in P) \quad (4)$$

$$\delta(p,i,j) = \textit{true} \qquad : (\forall p \in P), \forall (i,j) \in (C(p) \times C(p)) \quad (5)$$

Given the feasibility constraints defined above, we assume there exists an assignment function $\mathcal{A} : (G, V) \rightarrow (\mathcal{C}, \mathcal{T})$ that, given a network infrastrucutre G, and a set of telemetry items V, it returns a feasible solution $(\mathcal{C}, \mathcal{T})$, with respect to constraints *(i)*, *(ii)*, *(iii)*, and *(iv)*.

Objective. Given a feasible solution $(\mathcal{C}, \mathcal{T})$ and a set of faulty nodes D^*, the optimization problem seeks a new assignment $\overline{\mathcal{A}} : (G - \{D^*\}, V) \rightarrow (\overline{\mathcal{C}}, \overline{\mathcal{T}})$ that minimizes the number of changes in the current solution $(\mathcal{C}, \mathcal{T})$. In other words, the solution of $\mathcal{A}$ and $\overline{\mathcal{A}}$ should be as similar as possible, despite the required changes with respect of faulty nodes D^* The objective function aims to minimize the changes made on current probing cycles by maximizing the intersection between the current and the new solution. Equation (6) describes the objective function. Observe that parameters α and β are used for weighting the importance of probing cycle structures (i.e., α) and telemetry items assignments (i.e., β).

$$\textbf{Maximize} \quad \alpha \cdot \sum_{p \in P} |\mathcal{C}_p \cap \overline{\mathcal{C}_p}| + \beta \cdot \sum_{p \in P} |\mathcal{T}_p \cap \overline{\mathcal{T}_p}| \qquad (6)$$

IV. Proposed Heuristic Approach

To tackle the above problem efficiently and provide a quality-wise solution, we propose a heuristic procedure that only rebuilds parts of probing cycles affected by the faulty nodes. Next, we overview the ideas behind our proposed heuristic, and then we discuss the pseudo-code and its complexity analysis.

The heuristic procedure aims to maintain all network links covered while collecting all telemetry items from available devices. The main idea consists of reconstructing only affected cycles by faulty nodes D^*. Algorithm 1 summarizes our proposed approach. Our heuristic receives as input a feasible solution $(\mathcal{C}, \mathcal{T})$ and the set of faulty nodes D^*. In lines 1-5, we "patch" all affected cycles by directly connecting the predecessor and the successor of a given faulty node $i \in D^*$. We perform this new interconnection using the shortest path. As we do not control the length of this new interconnection, it could be that the applied "patch" violates the probe's capacity. In this case, we exclude telemetry items from the affect cycles until it does not (lines 6-11). Then, we iterate over the unsatisfied items U_{dv} (i.e., those removed from probing cycles) and look for a probing $p \in P$ that is able to collect it again (lines 12-19). If the existing probing paths P could not collect all the remaining items, we then create new probing cycles for that purpose (lines 20-29). We sort telemetry items U_{dv} with respect to forwarding devices D so as to ensure that we collect items in a given order. Finally, we apply a local search procedure (line 30) that further optimizes our incumbent solution. The local search procedure strives to minimize the number of probing cycles by removing cycles in which forwarding devices have already been covered by other probes. We iterate over each pair of probing cycles, checking whether or not it is possible to delete. Our proposed approach has a worst-case time complexity of $\mathcal{O}(D \cdot V \cdot P)$, considering the most consuming code section (lines 12-18).

Algorithm 1 Overview of the patcher procedure.

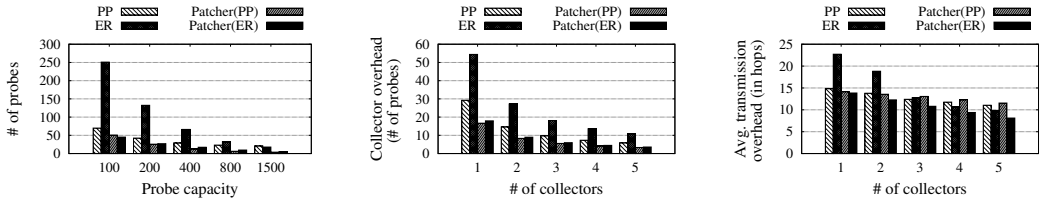
Input: $(\mathcal{C}, \mathcal{T})$: initial solution; D^*: subset of faulty nodes.
1: **for** all devices $i \in \mathcal{C}(p) : (\forall p \in P)$ **do**
2: **if** $i \in D^*$ **then**
3: $\mathcal{C}(p) \leftarrow \mathcal{C}(p) - i \cup \text{ShortestPath}(\mathcal{C}(p)_{[i-1]}, \mathcal{C}(p)_{[i+1]})$
4: **end if**
5: **end for**
6: **for** all cycles $p \in P$ such that Equation (3) is not satisfied **do**
7: **while** capacity $U(p)$ constraint is not satisfied **do**
8: exclude a collected item, until it does
9: Save item on U_{dv}
10: **end while**
11: **end for**
12: **for** all unsatisfied items $(d, v) \in U_{dv}$ **do**
13: **for** all $p \in P$ the cycles **do**
14: **if** cycle p has residual capacity $U(p) - S(v) \geq 0$ **then**
15: $U_{dv} \leftarrow U_{dv} - (d, v)$; $U(p) \leftarrow U(p) - S(v)$
16: $\mathcal{T}(p) \leftarrow \mathcal{T}(p) \cup (d, v)$
17: **end if**
18: **end for**
19: **end for**
20: Sort $(d, v) \in U_{dv}$ regarding device d.
21: **while** $U_{dv} \neq \varnothing$ **do**
22: $P \leftarrow$ new probe containing a cycle $C(p_{new})$ which prioritizes unsatisfied items U_{dv}
23: **while** $(d, v) \in U_{dv}$ **do**
24: **if** cycle p has residual capacity $U(p) - S(v) \geq 0$ **then**
25: $U_{dv} \leftarrow U_{dv} - (d, v)$; $U(p) \leftarrow U(p) - S(v)$
26: $\mathcal{T}(p) \leftarrow \mathcal{T}(p) \cup (d, v)$
27: **end if**
28: **end while**
29: **end while**
30: Apply local search
31: **return** new solution $(\mathcal{C}, \mathcal{T})$

V. Evaluation

Setup. All experiments were performed on a machine with AMD Threadripper 2920X processor and 80 GB of RAM, using the Ubuntu 16.04 operating system. We considered physical network instances that were generated with Brite [23], following the Barabasi-Albert model [24]. We used physical network infrastructures varying from 10 to 100 forwarding devices. We vary the amount of available space to embed telemetry items in probing packets (i.e. $U(p)$) from 100 to 1500 Bytes. Further, we assume that forwarding devices have 8 possible telemetry items to export[1], varying its size $S(v)$ uniformly from 2 to 20 Bytes [1]. Yet, we consider that there exist a single faulty node (i..e, $|D^*| = 1$) in the network infrastructure G. In each execution, we vary this set to cover all devices $i \in D$, ensuring that all devices fail individually. We consider $\alpha = 1$ and $\beta = 1$. As future work, we left the evaluation of higher values for $|D^*|$, as well the fine-tuning of α and β.

We compare the results obtained by *Patcher* against (i) the `Edge Randomization (ER)` [25], and (ii) the recent state-of-the-art work proposed by Pan et al. [1], namely `PathPlanning (PP)`. `Edge Randomization` is a well-known heuristic procedure used to build feasible solutions to the arc routing problem. We modify such a strategy to the problem in hand as follows. We randomly select a starting

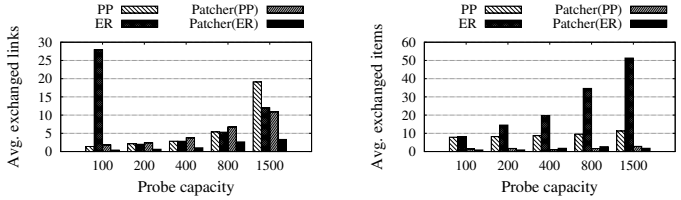
[1]In-band Network Telemetry: https://p4.org/assets/INT-current-spec.pdf



(a) Number of probing cycles for an increasing network size.

(b) Collector load.

(c) Minimum distance to the closest collector.



(d) Average difference of $|\mathcal{C}_p \cap \overline{\mathcal{C}_p}|$.

(e) Average difference of $|\mathcal{T}_p \cap \overline{\mathcal{T}_p}|$.

Fig. 2. In-band probing cycles performance metrics.

device $d \in D$ to start the probing cycles p. While the capacity $U(p)$ is not violated, we collect all telemetry items V_i of the current node i and randomly select a next forwarding device to further expand the current probing cycle. At the point that the capacity $U(p)$ is reached, we return to the origin forwarding device (the procedure ensures that there is enough capacity to make the way back). Our initial solution to the problem (Section III) is given according to the above algorithms (i.e., `ER` and `PP`). On the event of a failure in devices D^*, we apply *Patcher* on top of existing solutions – mentioned as *Patcher*(`ER`) or *Patcher*(`PP`). Alternatively, we re-execute algorithms `ER` and `PP` from scratch (considering faulty nodes D^*) as a baseline comparison.

Metrics. We focus our evaluation on five metrics: (i) number of probing cycles, (ii) INT collector overhead (i.e., the number of probing cycles assigned to a given INT collector), (iii) transmission overhead (i.e., the distance between the probing cycles and the closest INT collector), (iv) the number of changes in current solution regarding links (i.e., the difference between sets $\mathcal{C}_p$ and $\overline{\mathcal{C}_p}$), and (v) the number of changes in current solution regarding telemetry data (i.e., the difference between sets $\mathcal{T}_p$ and $\overline{\mathcal{T}_p}$).

Results. *Number of probing cycles*. Figure 2(a) illustrates the average amount of probing cycles after a given faulty node, varying probing cycle from 100 to 1500. Note that in all graphs, we show the average considering all faulty nodes D^*. *Patcher* can substantially reduce the number of probing cycles after an observed fault in comparison to `ER` (up to 82%) and `PP` (up to 27.3%). This is mostly due to reorganizing existing probing cycles, instead of rebuilding a new solution from scratch. The number of probing cycles is reduced as fewer network telemetry items are to be collected (and links to be covered) in a faulty situation.

Collector and transmission overheads. Figure 2(b) and Figure 2(c) illustrate the collector and transmission overhead, respectively. In both figures, we increase the number of INT collectors from 1 to 5. These metrics are calculated considering the optimal placement of INT collectors. That is, given a set of probing cycles, we place the available INT collectors in a given set of forwarding node D to minimize the value of these metrics. Figure 2(b) illustrates the average collector overhead: the amount of probing cycles assigned to each of them. The lower the number of INT collectors, the higher is the overhead. When there is a fault in the network, we analyze how this collector load is affected. Observe that *Patcher* reduces (up to 3x) this overhead as a consequence of reducing the number of probing cycles (illustrated in Figure 2(a)). In turn, Figure 2(c) illustrates the average transmission overhead (i.e., the number of hops between probing cycles and placed INT collectors). Similarly to the collector overhead, the transmission overhead is affected by the number of INT collectors. The more INT collectors, the lower is the transmission costs – as this increases the chance of having an INT collector closer to a given probing cycle. When a fault occurs, the transmission costs are maintained (or reduced up to 60.9%) in comparison to re-executing the whole solution.

Changes in existing solutions. Figure 2(d) and Figure 2(e) illustrate the average amount of changes required to implement a new solution on the event of a fault – regarding changes in links and telemetry items assignments. Observe that these two metrics are rather important in order to assess the recovery time taken by programmable network infrastructure in the case of a fault. Note, as well, that these changes can be instrumented directly by the control plane (e.g., by installing new rules in forwarding devices) or be encoded into INT probing packets to be handled directly by the data plane. In either case, a low number of changes are expected in order to reduce the recovery time or the resource usage of probing packets. Figure 2(d) illustrates the number of probing cycle links that have been changed from the initial solution to the new one (e.g., given by *Patcher* or re-executing the algorithms from scratch). Observe that, in general, *Patcher* reduces substantially the number of observed changes in existing solutions as it applies "patches" only on affected parts

of probing cycles. *Patcher* can reduce up to 27 (98.5%) and 8 (43%) link changes in comparison to ER and PP, respectively. In turn, Figure 2(e) shows the number of telemetry items data has been reassigned from one probing cycle to another. This happens due to space constraints in existing cycles. Eventually, a feasible solution encompasses the reallocation of existing telemetry items among available probing cycles when a fault occurs. As observed in the figure, the solutions produced by *Patcher* add a negligible amount of changes in comparison to ER (up to 96.5%) and PP (up to 74.8%).

VI. Final Remarks

In this paper, we proposed *Patcher* fault-tolerant probing planning for INT. To the best of our knowledge, *Patcher* is the first mechanism to consider network failures and create dynamic solutions over time. As shown, our solution *(i)* reduces the number of probes by a factor of 5.5x, *(ii)* makes better use of network resources by keeping the collector and transmission overheads of probing cycles as low as possible, and (iii) requires up to 98% fewer changes (in comparison to baselines) to implement our fault-tolerant mechanism. As future work, we intend to evaluate the impact of multiple simultaneous failures on *Patcher*'s efficiency.

Acknowledgements

This work was partially funded by National Council for Scientific and Technological Development (CNPq 2018/427814), Foundation for Research of the State of Sao Paulo (FAPESP 2018/23092-1), and Foundation for Research of the State of Rio Grande do Sul (19/2551-0001224-1, 19/2551-0001266-7).

References

[1] T. Pan, E. Song, Z. Bian, X. Lin, X. Peng, J. Zhang, T. Huang, B. Liu, and Y. Liu, "Int-path: Towards optimal path planning for in-band network-wide telemetry," in *IEEE INFOCOM*, Apr 2019, pp. 1–9.

[2] Z. Liu, J. Bi, Y. Zhou, Y. Wang, and Y. Lin, "Netvision: Towards network telemetry as a service," in *IEEE ICNP*, Sep. 2018, pp. 247–248.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, p. 503–514, Aug. 2014. [Online]. Available: https://doi.org/10.1145/2740070.2626316

[4] R. L. Carter and M. E. Crovella, "Server selection using dynamic path characterization in wide-area networks," in *Proceedings of the INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution*, ser. INFOCOM '97. USA: IEEE Computer Society, 1997, p. 1014.

[5] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid, "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*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–14. [Online]. Available: https://doi.org/10.1145/3359989.3365410

[6] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "SIMON: A simple and scalable method for sensing, inference and measurement in data center networks," in *USENIX NSDI 19*. Boston, MA: USENIX Association, Feb. 2019, pp. 549–564.

[8] R. Hohemberger, A. G. Castro, F. G. Vogt, R. B. Mansilha, A. F. Lorenzon, F. D. Rossi, and M. C. Luizelli, "Orchestrating in-band data plane telemetry with machine learning," *IEEE Communications Letters*, vol. 23, no. 12, pp. 2247–2251, 2019.

[7] J. A. Marques, M. C. Luizelli, R. I. T. Da Costa, and L. P. Gaspary, "An optimization-based approach for efficient network monitoring using in-band network telemetry," *Journal of Internet Services and Applications*, no. 1, p. 16, Jun 2019.

[9] R. Hohemberger, A. F. Lorenzon, F. D. Rossi, and M. C. Luizelli, "A heuristic approach for large-scale orchestration of the in-band data plane telemetry problem," in *Advanced Information Networking and Applications*, L. Barolli, F. Amato, F. Moscato, T. Enokido, and M. Takizawa, Eds. Cham: Springer International Publishing, 2020, pp. 381–392.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[11] A. F. Q. Wu, J. Strassner and L. Zhang. (2016, Mar.) Network telemetry and big data analysis. [Online]. Available: https://tools.ietf.org/html/draft-wu-t2trg-network-telemetry-00

[12] A. Putina, D. Rossi, A. Bifet, S. Barth, D. Pletcher, C. Precup, and P. Nivaggioli, "Telemetry-based stream-learning of bgp anomalies," in *Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, ser. Big-DAMA '18. New York, NY, USA: ACM, 2018, pp. 15–20. [Online]. Available: http://doi.acm.org/10.1145/3229607.3229611

[13] m. . j. y. . . u. . h. The P4.org Applications Working Group, title = In-band Network Telemetry (INT) Dataplane Specification.

[14] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626292

[15] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," in *Proceedings of the 2015 ACM Conference on SIGCOMM*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 479–491. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787483

[16] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proceedings of the 2018 ACM Conference on SIGCOMM*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018.

[17] Q. Huang, H. Sun, P. P. Lee, W. Bai, F. Zhu, and Y. Bao, "Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy," in *Proceedings of the ACM SIGCOMM*, 2020, pp. 404–421.

[18] Y. Li, K. Gao, X. Jin, and W. Xu, "Concerto: cooperative network-wide telemetry with controllable error rate," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020, pp. 114–121.

[19] P. Tammana, R. Agarwal, and M. Lee, "Simplifying datacenter network debugging with pathdump," in *12th USENIX OSDI 16)*, Savannah, GA, 2016, pp. 233–248.

[20] ——, "Distributed network monitoring and debugging with switchpointer," in *15th USENIX NSDI 18*, Renton, WA, 2018, pp. 453–456.

[21] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi *et al.*, "Flow event telemetry on programmable data plane," in *Proceedings of the ACM SIGCOMM*, 2020, pp. 76–89.

[22] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 127–140. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098832

[23] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: an approach to universal topology generation," in *IEEE MASCOTS 2001*, Aug 2001, pp. 346–353.

[24] R. Albert and A.-L. Barabási, "Topology of evolving networks: Local events and universality," *Physical Review Letters*, vol. 85, pp. 5234 – 5237, Dec 2000.

[25] J.-M. Belenguer, E. Benavent, P. Lacomme, and C. Prins, "Lower and upper bounds for the mixed capacitated arc routing problem," *Computers & Operations Research*, vol. 33, no. 12, pp. 3363–3383, 2006.

ANNEX C – NEAR-OPTIMAL PROBING PLANNING FOR IN-BAND NETWORK TELEMETRY

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2021.3053485, IEEE Communications Letters

Near-Optimal Probing Planning for In-Band Network Telemetry

Ariel G. Castro, Arthur F. Lorenzon, Fábio D. Rossi, Roberto I. T. da Costa Filho, Fernando M. V. Ramos, Christian E. Rothenberg, Marcelo C. Luizelli

***Abstract*—In-band Network Telemetry (INT) is gaining traction as an advanced network monitoring approach. Despite a few recent initiatives to orchestrate the collection of in-band network statistics, state-of-the-art approaches fall short when it comes to efficiently collect telemetry items while subjected to real-world constraints. In this letter, we propose Probe Planning for In-Band Network Telemetry (P^{2}INT) to coordinate how probing packets are generated and routed to ensure that all links are covered so that the required in-band network telemetry data is collected. We theoretically formalize the problem as a Integer Linear Programming model and propose an efficient mathematical programming-based heuristic to solve it. Our results show that P^{2}INT outperforms the closest contender by a factor of up to 6x concerning the number of probing cycles generated.**

***Index Terms*—In-band Network Telemetry; INT; Math-heuristic; MILP; data plane programability; P4.**

I. Introduction

In-band Network Telemetry (INT) has recently emerged as a promising near real-time network monitoring to improve network visibility [1], [2], [3]. Due to the rich spectrum of benefits behind INT, there is increasing attention from the networking ecosystem fostered by the rapid adoption of programmable data planes and domain-specific networking description languages (e.g., P4 [4]). In short, INT consists of instrumenting the collection of low-level network statistics directly from the data plane. In the classic hop-by-hop INT (a.k.a INT-MD (eMbed Data)[1]), an INT source node embeds instructions into production network packets. Then, INT transit nodes embed metadata while an INT sink node strips the instruction out of the packet and sends the accumulated telemetry data to an INT collector.

In this work, we consider using probe packets to instruct network devices to collect telemetry data. Figure 1 illustrates the entire INT process. In the first step, probing packets are generated aiming at instrumenting the collection of telemetry data along a given path. For example, the red flow (i.e. f_1) – that is routed through the devices A, E, F, G, H, and I – carries instructions to collect telemetry data from devices A to H. In the second step, the collected telemetry data is extracted and reported to an INT collector.

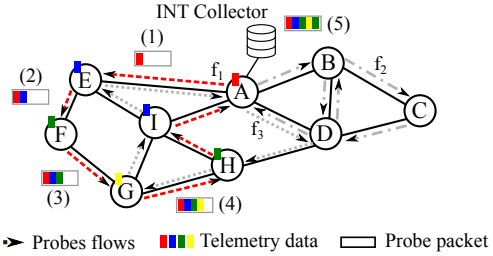


Fig. 1. Example of a solution for the probing planning problem, illustrating a snapshot where probing packets (f_1, f_2, f_3) collect telemetry data from selected network devices.

Recently, investigations have made the first efforts towards the orchestration of INT data collection to improve network-wide visibility. Liu et al. [2], Pan et al. [3], Bhamare et al.[5], and Geng et al. [6] have focused on performing network telemetry through active INT-based probing packets. These strategies have relied either on *Euler Circuits* [2], [3] or on actual routing paths [5], [6] to instrument the forwarding of probes. In turn, Marques et al. [7] and Hohemberger et al. [8] have focused on the embedding of INT data into production network packets. Marques et al. [7] designed heuristic approaches to orchestrate how network flow packets collect network telemetry data, while Hohemberger et al. [8] designed a machine learning-based model to wisely choose and collect INT data based on its importance. Further, others studies [9] have focused on the design of operational INT mechanisms and on reducing the amount of INT messages being reported.

Despite current efforts towards near real-time in-band network telemetry, the coordination of INT probing packets to collect network information efficiently is still full of gaps and challenges. The first attempts [2], [3], [5] to tackle this problem contributed with initial steps but suffer from (i) uncoordinated probing packet generation, and (ii) neglected capacity constraints. By using uncoordinated probes, there is an increasing transmission overhead from active probes to the INT collectors (as discussed later on). Furthermore, relaxing capacity constraints of probing packets simplifies the problem – but unrealistic from an operational point of view[2]. In this letter, we introduce P^{2}INT – Probe Planning for In-Band Network Telemetry – to coordinate how probing packets are generated and routed in order to ensure that all links are visited and all required INT data is collected. Although existing solutions have either focused on INT probes to monitor link

This research was partially supported by National Council for Scientific and Technological Development (CNPq) (grant 427814/2018-9), São Paulo Research Foundation (FAPESP) (grant 2018/23092-1), Rio Grande do Sul Research Foundation (FAPERGS) (grants 19/2551-0001266-7,20/2551-000483-0, 19/2551-0001224-1) and by the Portuguese national funds through FCT via UIDB/50021/2020 and PTDC/CCI-INF/30340/2017 (uPVN) projects.

Ariel G. Castro, Arthur F. Lorenzon, and Marcelo C. Luizelli are with the Federal University of Pampa, Brazil.

Fabio D. Rossi is with the Federal Institute Farroupilha, Brazil.

Roberto I. T. da Costa Filho is with Instituto Federal de Educação, Ciência e Tecnologia Sul-Rio-Grandense, Brazil.

Fernando M. V. Ramos is with the University of Lisbon, Portugal.

Christian E. Rothenberg is with the University of Campinas, Brazil.

[1]INT specification: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf

[2]For instance, in the Netronome SmartNIC architecture, the ingress timestamp has 64 bits, while the ingress port 16 bits.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2021.3053485, IEEE Communications Letters

connectivity (e.g. [2], [3]) or focused on the collection itself of INT data (e.g. [7], [8]), they still miss how to jointly optimize the way to collect telemetry data and cover network links. To tackle this problem, we theoretically formalize P^{2}INT as a Integer Linear Programming model. The model consists of a generalization of two well-known optimization problems – namely, Capacitated Arc Routing problem and Bin Packing problem [10] and, therefore, it is an NP-hard problem. We introduce a novel mathematical programming-based heuristic that wisely guides the MILP model to find a high-quality solution. Results show that P^{2}INT outperforms the state-of-the-art solution [3] by a factor of 6x related to the number of probes, at the same time that makes better usage of available resources (up to 3x) and decreases the transmission overhead to INT collectors (up to 2x).

II. P^{2}INT: Probing Planning for In-band Network Telemetry

A. Problem Overview

The P^{2}INT problem consists of defining optimized probing cycles to cover a given network infrastructure, i.e., in terms of telemetry data and network connectivity. It is noteworthy that the complete network coverage, both in terms of links and nodes, enables the assessment of end-to-end metrics based on different composition rules (e.g., multiplicative, additive, and concave) [11]. This approach is crucial for operating in large-scale networks such as the 5G device-to-device ecosystem, where path-based measurements are prohibitive due to the massive number of available paths. The P^{2}INT problem is not trivially solved. First, probing packets are space-bounded (i.e., w.r.t. bytes), and therefore it is infeasible (in most cases) to collect all network telemetry data with a single packet. Second, routing a probing packet is challenging. Probes need to be routed in such a manner that telemetry data requirements are met, while avoiding extra overheads on production network traffic (e.g., excessive generation of probing cycles). Figure 1 illustrates a network infrastructure with nine programmable forwarding devices (ranging from A to I), each having exactly one equal-sized telemetry data (represented by colored rectangles). These telemetry data represent data planes' internal states (e.g., queue occupancy or processing time), which are used by specialized monitoring applications [8] (e.g., DDoS detection). In the example, probing packets are limited to collect at most five telemetry data. There exists a set of active probing cycles (i.e., f_1, f_2, and f_3) which are responsible for continuously (i) collecting telemetry data and (ii) checking network connectivity. Probing cycles f_1, f_2, and f_3 are routed and instrumented to collect a given subset of telemetry data. For instance, probing cycle f_1 collect telemetry data from forwarding devices A to H, while probing cycle f_3 from devices D and I. Observe that all network links are covered by at least one probing cycle.

B. Model description and notation

The proposed optimization model considers a physical network infrastructure $G = (D, L)$ and a set of telemetry items V. Set D in network G represents programmable forwarding devices $D = \{1, ..., |D|\}$, while set L consists of unidirectional links interconnecting pair of devices $(i, j) \in (D \times D)$. Similarly to the recent literature [7], [8], we consider that there exists a set of telemetry items V available. Each forwarding device $i \in D$ is able to embed a subset of items $V_i \subseteq V$ into a probing packet. Each telemetry item $v \in V$ has its size defined by function $S : V \rightarrow \mathbb{N}^+$.

We consider there is at most $|P|$ probing cycles (i.e., $P = \{1, 2, ..., |P|\}$) to collect telemetry items from forwarding devices D. Packets in a probing cycle are encapsulated in a forwarding protocol, and therefore the amount of available space to embed telemetry items in packets is bounded by a constant, defined by function $U : P \rightarrow \mathbb{N}^+$ (e.g., $U(p)$ lower or equal to the MTU data link). The larger is this set P, the higher is the amount of decision variables in the model – and so the search space. A worst-case upper-bound for $|P|$ can be estimated as $|P| = |V| \cdot |D| \cdot |L|$. Probing cycles P are routed within the network infrastructure G – i.e., the packet is generated in a given source device, is routed through a subset of devices, and returns to its origin. We denote the cycle taken by the probing $p \in P$ as function $\mathcal{C} : P \rightarrow \{D_1 \times ... \times D_{|D|}\}$. Probing cycles $p \in P$ can collect telemetry items from forwarding devices $i \in \mathcal{C}(p)$. The set of telemetry items collected by probing cycle $p \in P$ is represented by pairs $(i, v) : i \in D, v \in V_i$ and is given by the function $\mathcal{T} : P \rightarrow \{D \times V\}$. A feasible cycle satisfy the upper-bound $U(p)$, that is $\sum_{i \in \mathcal{C}(p)} \sum_{v \in V_i : (i,v) \in \mathcal{T}(p)} S(v) \leq U(p)$. Observe that a given cycle $p \in P$ can visit a forwarding device $i \in \mathcal{C}(p)$ and not necessarily collect the set of telemetry items associated. Yet, our model does not restrict a cycle $p \in P$ to collect any telemetry item[3]. We denote the origin (starting/ending device) of each cycle $p \in P$ as function $O : P \rightarrow D$. Therefore, our model is generic to consider single- and multi-source probing cycle scenarios (i.e., cycles might start at different INT sources).

Given the problem input, the optimization problem seeks a feasible solution that minimizes the number of probing cycles, while visiting all network links and collecting the required telemetry data. The model output is denoted by a 3-tuple $\chi = \{Z, X, Y\}$. Variables from $Z = \{ z_{p,v,i}, \forall p \in P, v \in V, i \in D\}$ indicate that a forwarding device i embed telemetry item v into a probing packet from cycle p. Variables from $X = \{ x_{p,i,j}, \forall p \in P, (i, j) \in L\}$ indicates that network link $(i, j) \in L$ is used to route probing cycle $p \in P$. Last, variable $Y = \{ y_p, \forall p \in P\}$ is used to keep track of probing cycles used by the solution. Next, we describe the ILP formulation.

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

Subject to:

$$\sum_{p \in P} z_{p,v,i} = 1 \qquad \forall i \in D, v \in V_i \tag{2}$$

$$z_{p,v,i} \leq \sum_{j \in D} x_{p,j,i} \qquad \forall p \in P, i \in D, v \in V_i \tag{3}$$

$$z_{p,v,i} + x_{p,i,j} \leq 2 \cdot y_p \qquad \forall p \in P, (i, j) \in L, v \in V_i \tag{4}$$

[3]Telemetry items might be only available to specific queues inside the data plane.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2021.3053485, IEEE Communications Letters

$$\sum_{j \in D} x_{p,i,j} - \sum_{j \in D} x_{p,j,i} = 0 \qquad \forall p \in P, i \in D \quad (5)$$

$$\sum_{p \in P} x_{p,i,j} + x_{p,j,i} \geq 1 \qquad \forall (i,j) \in L \quad (6)$$

$$\sum_{i \in D} \sum_{v \in V_i} z_{p,v,i} \cdot S(v) + \sum_{i \in D} \sum_{j \in D} x_{p,i,j} \leq U(p) \qquad \forall p \in P \quad (7)$$

$$\sum_{i \in S} \sum_{j \in S} x_{p,i,j} \leq |S| - 1 \qquad \forall p \in P, S \subseteq \{D - O_p\}, |S| \geq 2 \quad (8)$$

$$z_{p,v,i} \in \{0,1\} \qquad \forall p \in P, v \in V_i, i \in D \quad (9)$$

$$y_p \in \{0,1\} \qquad \forall p \in P \quad (10)$$

$$x_{p,i,j} \geq 0 \qquad \forall p \in P, v \in V_i, i \in D \quad (11)$$

Constraint set (2) ensures that generated probing cycles collect the required network telemetry data. Constraint set (3) ensures that if telemetry item v is collected from forwarding device i, then there should have a probe being routed through i. Constraint set (4) accounts for the number of probing cycles in use. In short, it sets a cycle p as active whenever variables $z_{p,v,i} = 1$ or $x_{p,i,v} = 1$. Constraint set (5) ensures flow conservation on probing cycles. In other words, they generate probing cycles without ramification or self-loops. In turn, constraint set (6) guarantees a probing cycle covers at least one link direction. Constraint set (7) ensures that the available capacity is not violated either by the telemetry items collected or by the network links being covered. Observe that $\sum_{i \in D} \sum_{j \in D} x_{p,i,j}$ limits the probing cycle length. Constraint set (8) is the well-known sub-tour elimination constraints, ensuring that generated cycles are strongly connected [12]. Last, constraint sets (9)–(11) define the domains of output variables. It is worth mentioning that the complexity of the proposed model comes from (i) capacitated probe packets, (ii) non-uniform size of telemetry data, and (iii) cycle definition.

III. A Math-Heuristic Approach to P^{2}INT

To tackle the P^{2}INT complexity and come up with near-optimum solutions, we introduce a mathematical programming-based heuristic. To minimize the number of probing cycles in the solution χ, our strategy optimizes a few probing cycles at once, to merge them by reallocating telemetry data to other cycles. The idea consists of iteratively choosing a subset of probing cycles to be optimized (i.e., their variables Z, X, Y are freely changed), while the others remain fixed. Algorithm 1 presents the proposed approach. We first compute a feasible solution χ to the P^{2}INT problem (line 1). Then, we iteratively select a subset of k probing cycles (lines 5-10), and enumerate the list of variables $x_{p,i,j} \in \mathcal{D} \subseteq X$ related to them (line 11); variables listed in $\mathcal{D}$ will be subject to optimization, while others will remain unchanged (line 12).

We take advantage of meta-heuristic VNS (Variable Neighborhood Search) to systematically iterate over subsets of probing cycles. Further, we prioritize subsets with higher potential for improvement – i.e., probing cycles that might be merged (discussed in Subsection III-C). For each subset of probing cycles, we submit its set of variables $\mathcal{D}$ along with χ to a mathematical programming solver. The goal is to obtain a set of values to those variables listed in $\mathcal{D}$, so that a better solution is found. In case there is no improvement, we rollback

Algorithm 1 Overview of the fix-and-optimize heuristic.

Input: T_{global}: global time limit, T_{local}: time limit for each solver run, $\mathcal{K}_{init}, \mathcal{K}_{end}$: initial/final neighborhood size, $\mathcal{K}_{inc}$: increment for neighborhood size, *NoImprov*$_{max}$: max. rounds without improvement
Output: χ: best solution found to the optimization model

1: $\chi \leftarrow$ initial feasible solution
2: **if** a feasible solution does not exist **then** fail **else**
3: $k \leftarrow \mathcal{K}_{init}$
4: **while** T_{global} is not exceeded **and** $k \leq \mathcal{K}_{end}$ **do**
5: $\mathcal{N}_k \leftarrow$ current neighborhood, i.e. tuples of k probing cycles
6: $\mathcal{N}_{k,shr} \leftarrow$ tuples from $\mathcal{N}_k$, whose cycles share devices
7: $\mathcal{N}_{k,any} \leftarrow \mathcal{N}_k \setminus \mathcal{N}_{k,shr}$
8: *NoImprov* $\leftarrow 0$
9: **while** $\{\mathcal{N}_{k,shr}, \mathcal{N}_{k,any}\} \neq \emptyset$ **and** *NoImprov* $\leq$ *NoImprov*$_{max}$ **do**
10: $\mathcal{T} \leftarrow$ next unvisited neighbor (w.r.t Equation 16)
11: $\mathcal{D} \leftarrow$ list of variables $x_{p,i,j}$ from cycles in neighbor $\mathcal{T}$
12: $\chi' \leftarrow$ solution χ optimized by the solver, under time limit T_{local}, and making variables not in $\mathcal{D}$ as fixed
13: **if** χ' is a better solution than χ **then**
14: update χ to reflect solution χ';
15: $k \leftarrow \mathcal{K}_{init}$;
16: **break**
17: **else**
18: *NoImprov* $\leftarrow$ *NoImprov* + 1
19: **end if**
20: **end while**
21: **if** no improvement was made **then** $k \leftarrow k + \mathcal{N}_{inc}$ **end if**
22: **end while**
23: **return** χ
24: **end if**

and pick the probing cycle subset that follows. We run this process iteratively until a better solution is found. Once it happens, we replace the incumbent solution with χ' (line 14), and restart the process (i.e., $k \leftarrow \mathcal{K}_{init}$). This loop continues until we have explored the most promising combinations of available cycles, or T_{global} execution time is exceeded.

A. Obtaining an initial solution

The first step of our algorithm (line 1) is generating a feasible solution χ. The solution is one that satisfies all constraints, though not necessarily a high-quality one, in terms of used probing cycles. There are several ways to generate feasible solutions to P^{2}INT. We propose two approaches. The first consists of adapting the `Edge Randomization` (`ER`) heuristic – an approach widely applied in Arc Routing Problems. `ER` starts a probing cycle from a random unvisited network link. Then, the algorithm randomly chooses an adjacent forwarding device and collect as many network telemetry items as possible. While the probing capacity is not depleted, the algorithm keeps repeating this procedure. Once it happens, the probing cycle returns to its origin using the shortest-path approach. The procedure is repeated until all network telemetry items are collected and all network links are visited.

The second approach is based on recent state-of-the-art work `PathPlanning` proposed by Pan et al. [3].Their proposal does not consider probe capacity, nor data plane telemetry items. We adapt their DFS-like algorithm to consider both requirements in the best effort approach. Our adaptation of the proposed DFS-like strategy only moves on to a next network link *iff* there is enough capacity on the current probe to collect telemetry data and to return to its origin.

B. Neighborhood selection and prioritization

We carefully choose the subset of probing cycles $\mathcal{D} \in X$ that will be optimized. We explore the search space using

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2021.3053485, IEEE Communications Letters

VNS. In a nutshell, VNS organizes the search space in k-neighborhoods. Each neighborhood is determined as a function of the incumbent solution (χ), and a neighborhood size k. We build a neighborhood as a combination of any k probing cycles. Formally, we define a k-neighborhood as a set composed of k-tuples $\mathcal{N}_k = \{\, p \,|\, p \subseteq P \wedge y_p = 1 \,\}$. The number of neighbors in a k-neighborhood is given by the binomial coefficient $\binom{\sum_{p=1}^{P} y_p}{k}$. We focus only on active probing cycles to build our neighborhood, since other variables in the model are easily inferred once the probing cycles are defined.

The time required by the solver to optimize a solution χ and a subset $\mathcal{D} \subseteq X$ is often small. However, processing every candidate subset $\mathcal{D}$ from the entire k-neighborhood is impractical. Therefore, we prioritize those neighbors that might lead to a better solution. We prioritize tuples in the k-neighborhood set $\mathcal{N}_k$ according to two observations: *(i)* it is more probable to merge probing cycles if they are not over-committed (i.e., the higher the residual capacity, the better); and *(ii)* it is easier to merge cycles that are close to each other. We define a tuple priority, as a function of its residual capacity. The residual capacity of a tuple $\mathcal{T} \in \mathcal{N}_k$ is given by $r : \mathcal{T} \rightarrow \mathbb{R}^+$, according to Equation 16.

$$r(\mathcal{T}) = \sum_{p \in \mathcal{T}} \left(U(p) - \Big(\sum_{i \in D} \sum_{v \in V_i} z_{p,v,i} \cdot S(v) + \sum_{i \in D} \sum_{j \in D} x_{p,i,j} \Big) \right) \quad (16)$$

We break down a k-neighborhood set into two distinct sets. The first one is formed by tuples whose probing cycles sharing forwarding devices ($\mathcal{N}_{k,shr}$) – i.e., $\cap_{p \in \mathcal{T}} \neq \emptyset$. The second set is formed by remaining tuples in $\mathcal{N}_k$ ($\mathcal{N}_{k,any} = \mathcal{N}_k \setminus \mathcal{N}_{k,shr}$), *i.e.* those tuples whose cycles do not share any forwarding device. We first process the tuples of $\mathcal{N}_{k,shr}$. Then, we process the remainder ones ($\mathcal{N}_{any}$). Last, our approach takes as input $\textit{NoImprov}_{max}$. It indicates the maximum number of iterations without improvement that is allowed over a given neighborhood. We stop processing the current neighborhood once *NoImprov* exceeds $\textit{NoImprov}_{max}$ (line 9).

IV. Evaluation

A. Setup

The proposed model was ran using IBM *CPLEX Optimization Studio* 12.9 to obtain optimum solutions, while the proposed heuristic approach was implemented using Java language. Experiments were performed on a machine with AMD Threadripper 2920X processor and 80 GB of RAM, using the Ubuntu 16.04 operating system. We considered different physical network instances that were generated with Brite [13], following the Barabasi-Albert model [14]. We used physical network infrastructures varying from 10 to 200 forwarding devices. We vary the amount of available space to embed telemetry items in probing packets (i.e. $U(p)$) from 100 to 1500 Bytes. Further, we assume that forwarding devices have from 2 to 8 possible telemetry items to export, varying its size $S(v)$ uniformly from 2 to 20 Bytes [3]. P^{2}INT considers the following parameters $T_{global} = 6h$, $T_{local} = 600s$, $\mathcal{K}_{init} = 2$, $\mathcal{K}_{end} = 4$, $\mathcal{K}_{inc} = 1$, and $\textit{NoImprov}_{max} = 15$. The fine-tuning and sensitive analysis of these parameters is out of the scope of this work. For example, P^{2}INT can trade solution quality for resolution time by adjusting T_{global} and T_{local}. Using the t-test method, we found that 30 runs of each experiment is enough to achieve a confidence level 95% or higher.

Baseline. We compare P^{2}INT against (i) the optimal solution (`OPT`), (ii) the `Edge Randomization` (`ER`), and (iii) the recent state-of-the-art work `PathPlanning` (`PP`) [3].

Reproducibility. Our implementation is publicly available in order to encourage full reproducibility of our experiments[4] and foster the design of new solutions.

B. Results

We analyze the quality of the proposed approach by evaluating: (i) the number of probing cycles generated; (ii) the resource usage of probing cycles; (iii) the data transmission overhead to INT collectors; (iv) the INT collector usage; and (v) the network link coverage.

Probing cycles. Figure 2(a) illustrates the amount of probing cycles generated for an increasing size of network infrastructures (from 10 to 200). P^{2}INT comes up with quality-wise solutions compared to the optimal and the state-of-the-art approach `PP`. Our solution is able to approach the optimal value for small network infrastructures (up to 20 nodes)[5] At the same time, the `PP` produces solutions with up to 2x the number of probes considering small networks. For medium- to large-scale networks, P^{2}INT produces (on average) solutions with 2.2x and 3.70x fewer cycles compared to `PP` and `ER`, respectively. This behavior is explained by the ability of P^{2}INT to jointly route probing packets and collect items.

Probe scalability. Figure 2(b) depicts the impact of probing packet capacity with respect to the number of generated cycles. For the purpose of this evaluation, we show the results of a 50 node network infrastructure[6]. As the probe capacity increases, we observe a sharp reduction in the number of probing cycles – as there is more room to accommodate network telemetry data. When comparing P^{2}INT to its contenders, we observe that it is able to generate solutions with up to 5.5x and 4.6x fewer cycles than `PP` and `ER`, respectively – e.g., to $U(p) = 1500$.

Probe capacity. Figure 2(c) illustrates the average probing capacity usage by generated cycles. Observe that P^{2}INT can utilize up to 3x more available capacity than `PP` (e.g., $U(p) = 1500$). On average, P^{2}INT utilizes 70% of available resources, while the other strategies (`PP` and `ER`) 48% and 50%, respectively. By using available resources efficiently, P^{2}INT produces solutions with minimum overheads.

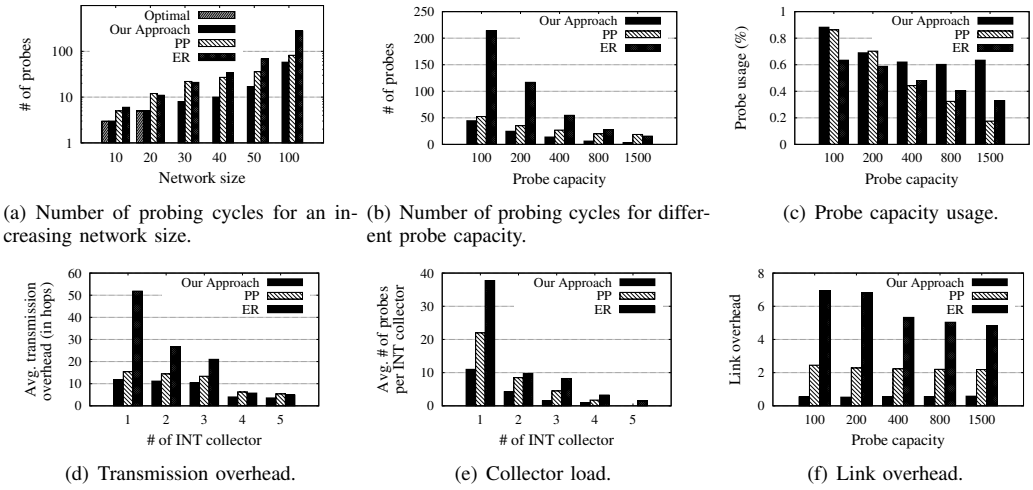
Network efficiency. Probing cycles might be computed in a way that they are not routed through an INT collector. In this case, at some point in the generated cycle, the collected INT data needs to be sent to a given monitoring sink. For the purpose of this evaluation, we consider that there are up to five INT collectors placed optimally according to the requirements of each solution. In other words, the INT collectors were placed connected to a forwarding device in a way that the

[4]Available implementation of our simulation: https://anonymous.4open.science/r/de4b2622-2b86-457b-82a0-fe2ad10004d8/

[5]Optimal solutions for large-scale ($|D| > 20$) networks are unfeasible due to NP-hardness. For small instances, the computing time surpasses 24h.

[6]The results for other network infrastructures follow the same behavior.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2021.3053485, IEEE Communications Letters



(a) Number of probing cycles for an increasing network size.

(b) Number of probing cycles for different probe capacity.

(c) Probe capacity usage.

(d) Transmission overhead.

(e) Collector load.

(f) Link overhead.

Fig. 2. In-band probing cycles performance metrics.

distance (in hops) to probing cycles is minimized. Figure 2(d) illustrates the transmission cost (in hops) to the closest INT collector. We opt to illustrate this cost in hops as the volume of transmitted data depends on the probe capacity/usage and the frequency that probe packets are generated. Observe that the more INT sinks are available in the infrastructure, the lower is the transmission overhead. Also, note that P^{2}INT keeps this transmission cost as low as possible even when there exists just one INT collector. On average, `PP` and `ER` generate solutions with 1.38x and 2.26x higher transmissions overheads than P^{2}INT, respectively.

Collector load. Figure 2(e) shows the collector load as the number of probing cycles assigned to each INT collector. Considering two INT collectors, solutions can cover 83% (our approach), 82% (`PP`) and 78% (`ER`) of all probing cycles. Note that the observed controller load can be substantially reduced if a data plane filtering mechanism is considered (e.g., [9]).

Network coverage. Figure 2(f) depicts the network link coverage as the average probing cycles per network link. Higher values indicate that network links are being over-covered (i.e., multiple times), representing a waste of resources. P^{2}INT, on average, keeps this value close to one, while the other strategies produce solutions with network links begin covered by up to seven probing cycles (i.e., 7x more than the necessary).

V. Final Remarks

In this letter, we formalized the Probing Planning for In-band Network Telemetry (P^{2}INT) employing a MILP model and introduced a scalable mathematical-based heuristic to solve it. While our approach outperforms state-of-the-art heuristics (e.g., factor 6 w.r.t probing cycles), it is still limited to (i) static solutions over time (i.e., probing cycles do not change); (ii) fixed-throughput of probing packets (i.e., all probing cycles operates at the same packet rate); and (iii) agnostic to network services (i.e., probing cycles are not aware of running functions or services). Addressing these limitations from the theoretical and operational point of view (e.g., control and data plane integration) is part of our future work.

References

[1] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," *ACM SIGCOMM CCR*, vol. 44, no. 4, pp. 3–14, 2014.

[2] Z. Liu, J. Bi, Y. Zhou, Y. Wang, and Y. Lin, "Netvision: Towards network telemetry as a service," in *IEEE ICNP*, Sep. 2018, pp. 247–248.

[3] T. Pan, E. Song, Z. Bian, X. Lin, X. Peng, J. Zhang, T. Huang, B. Liu, and Y. Liu, "Int-path: Towards optimal path planning for in-band network-wide telemetry," in *IEEE INFOCOM*, Apr 2019, pp. 1–9.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM 14*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[5] D. Bhamare, A. Kassler, J. Vestin, M. A. Khoshkholghi, and J. Taheri, "Intopt: In-band network telemetry optimization for nfv service chain monitoring," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–7.

[6] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "SIMON: A simple and scalable method for sensing, inference and measurement in data center networks," in *USENIX NSDI 19*. Boston, MA: USENIX Association, Feb. 2019, pp. 549–564.

[7] J. Marques, M. Luizelli, R. Da Costa, and P. Gaspary, "An optimization-based approach for efficient network monitoring using in-band network telemetry," *Journal of Internet Services and Applications*, no. 1, p. 16, Jun 2019.

[8] R. Hohemberger, A. G. Castro, F. G. Vogt, R. B. Mansilha, A. F. Lorenzon, F. D. Rossi, and M. C. Luizelli, "Orchestrating in-band data plane telemetry with machine learning," *IEEE Communications Letters*, vol. 23, no. 12, pp. 2247–2251, 2019.

[9] J. Vestin, A. Kassler, D. Bhamare, K. Grinnemo, J. Andersson, and G. Pongracz, "Programmable event detection for in-band network telemetry," in *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*, 2019, pp. 1–6.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. NY, USA: W. H. Freeman & Co., 1979.

[11] R. C. Filho, W. Lautenschläger, N. Kagami, M. Luizelli, V. Roesler, and L. Gaspary, "Scalable qoe-aware path selection in sdn-based mobile networks," in *IEEE INFOCOM*, April 2018, pp. 989–997.

[12] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Journal of the Operations Research Society of America*, vol. 2, no. 4, pp. 393–410, 1954.

[13] A. Medina, A. Lakhina, I. Matta, and J. Byers, "Brite: an approach to universal topology generation," in *IEEE MASCOTS 2001*, 2001.

[14] R. Albert and A.-L. Barabási, "Topology of evolving networks: Local events and universality," *Physical Review Letters*, vol. 85, pp. 5234 – 5237, Dec 2000.

ANNEX D – THE ACTUAL COST OF PROGRAMMABLE SMARTNICS: DIVING INTO THE EXISTING LIMITS

The Actual Cost of Programmable SmartNICs: diving into the existing limits

Pablo B. Viegas, Ariel G. de Castro, Arthur F. Lorenzon, Fábio D. Rossi, Marcelo C. Luizelli

Abstract

Programmable Data Planes is a novel paradigm that enables efficient offloading of network applications. An important enabler for this paradigm is the current available SmartNICs, which should satisfy rigid network requirements such as high throughput and low latency. Despite recent research in this field, not much attention was given to understand the costs and limitations of offloading network applications into SmartNIC devices. Existing offloading approaches either neglect the existing limitations of SmartNICs or assume that as an ongoing cost – leading, therefore, to sub-optimal offloading approaches. In this work, we conduct a comprehensive evaluation of SmartNICs in order to quantify existing performance limitations. We provide insights on network performance metrics such as throughput and packet latency while considering different key building blocks of complex P4 programs (e.g., registers, cryptography functions, or packet recirculation). Results show that line-rate throughput can degrade up to 8x, while latency can increase as much as 80x when performing memory-intensive operations in the data plane.

1 Introduction

Programmable Data Planes (PDP) is a mainstream technology that has been recently redesigning the networking domain. PDP allows to (re)define the behavior of network devices (e.g., programmable routers and SmartNICs),

Pablo B. Viegas, Ariel G. de Castro, Marcelo C. Luizelli, Arthur F. Lorenzon
Federal University of Pampa (UNIPAMPA)
Alegrete, Brazil, e-mail: {pabloviegas.aluno,arielcastro.aluno,marceloluizelli,arthurlorenzon}@unipampa.edu.br

Fabio D. Rossi
Federal Institute Farroupilha (IFFAR)
Alegrete, Brazil, e-mail: fabio.rossi@iffarroupilha.edu.br

allowing to deliver specialized packet processing mechanisms [2]. Recent advances in data plane programmability have enabled offloading typical control plane applications to the data plane (e.g., machine learning algorithms [19], routing [12], or network monitoring [4, 8, 9]). On shifting the operation of these applications to the data plane, it brings the benefit to process every single packet and react to network conditions in the order of nanoseconds, with minimum control plane intervention. Despite that, data plane operation might become complex – and the complexity comes at a price: *lower throughput and higher latency.*

Current SmartNIC architectures (e.g., [11]) do not limit the number of operations performed by the data plane in a single pipeline stage. For example, a PDP application could trigger the read and write of an unbounded number of registers in a given stage of the packet processing pipeline. Yet, the same application could recirculate the ingress packet multiple times in order to mimic a loop-based mechanism. These are straightforward examples of simple operations commonly used by more complex PDP applications (e.g., in-network clustering [19]). Therefore, understanding the current performance limitations of existing SmartNICs is paramount to the design of efficient PDP applications.

A recent study [7] has made the first effort to understand the existing limitations of SmartNICs. Harkous et al. [7] have focused on evaluating the performance of general PDP metrics such as parsers, control blocks, and header modifications in P4 programs. Despite this effort, no study has yet thoroughly evaluated key building blocks of complex P4 programs (e.g., registers, cryptography functions, or packet recirculation) – which are essential for most recent P4 applications. To fill in this gap, we perform an extensive performance evaluation of SmartNICs to understand and quantify PDP application primitives and existing limitations. We focus our evaluation on measuring the performance in terms of latency and throughput for a variety of packet sizes (from 64B to 1500B) when (i) operating multiple registers of different sizes/widths (e.g., used to implement bloom filter alike structures [1]), (ii) matching on multiple tables in the ingress and egress pipeline (e.g., used to implement machine-learning algorithms [19]), (iii) performing packet recirculation (e.g., used to implement IoT data desegregation [17]), and (iv) using cryptography functions and arithmetic operations. Results show that network throughput can degrade up to 8X, while latency can increase as much as 80X when performing memory-intensive operations in the data plane. The main contributions of this paper can be summarized as:

- an in-depth performance evaluation of SmartNICs;
- a discussion of current limitations in SmartNIC architectures; and
- an open-source code of all experiments in order to foster reproducibility.

The remainder of this paper is organized as follows. In Section 2, we describe the SmartNIC architecture used in this work. In Section 3, we overview the recent literature regarding PDP applications and performance evaluation. In Section 4, we present and discuss the obtained results and, in Section 5, we conclude this paper with final remarks.

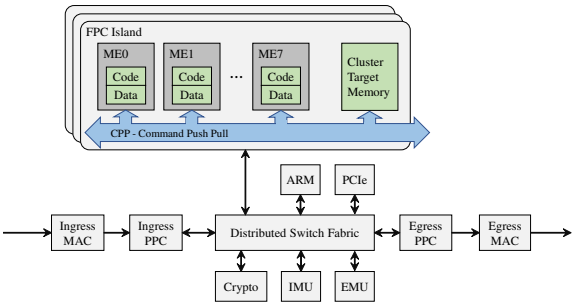


Fig. 1 An overview of the Netronome SmartNIC architecture.

2 SmartNIC Architecture

The hardware techniques used to deliver high-speed network packet processing require context switching on the order of nanoseconds and very high degrees of processing parallelism to scale the performance of P4-based programs to Gb/s of throughput and beyond [11]. This is particularly challenging given the ever-increasing complexity of offloaded codes to network adapters. Current programmable NICs (also referred to as SmartNICs) rely their architectures on multi-threaded, multicore flow processor units to cope with this increasing and stringent demand. Next, we focus our discussion on the general architectural details of the Netronome SmartNIC architecture [11] – which is used later in our experiments.

The SmartNIC Netronome NFP4000 architecture is illustrated in Figure 1 and organizes its flow processing cores (FPC) in multiple islands. FPC is a 32-bit machine, and therefore, all of its internal registers and local memory are formed from 32-bit words. Each FPC contains eight Micro Engines (MEs) – a separate processor with its own instruction store (*code*) and local memory (*data*). Because of that, every ME can run in parallel with all other MEs. Each ME has 8 threads that can be used for co-operative multithreading, in the way that at most one thread is executing code from the same program at any given moment. Hence, each FPC runs at most 8 parallel threads at 1.2Ghz (one thread per ME). FPCs follow a Harvard Architecture, and therefore code and data occupy different memories. Usually, 4K bytes are shared between all 8 threads for data and private memory of 8K instructions for the coding store.

In each FPC, local memory is composed of a set of 32-bit registers, shared between all 8 threads. These registers are divided into: (*i*) general-purpose registers (256 32-bits registers) – used by default to store any register of up to 32-bits size; (*ii*) transfer registers (512 32-bits registers) – used for copying register over the interconnection bus (e.g., from or to other FPCs or memories); (*iii*) next-neighbor registers (128 32-bits registers) – used mainly to communicate with neighboring FPCs; and last (*iv*) local memory (1024 32-bit registers) – which is a little bit slower than general register (3 cycles access,

instead of just 1 cycle). When there is a need for more memory than available space in local FPC registers, variables are automatically and statically allocated to other in-chip memory hierarchies.

There are four other kinds of memory which are available to FPCs: (*i*) Cluster Local Scratch (CLS) (20-50 cycles); (*ii*) Cluster Target Memory (CTM) (50-100 cycles); (*iii*) Internal Memory (IMEM) (120-250 cycles); and (*iv*) External Memory (EMEM) (150-590 cycles). In summary, the local memory register is used for data that is used in every packet. The CLS is used for data, which is needed for most packets and small shared tables. The CTM is used for packet headers and coordination between other sub-systems. Then, IMEM is used for packet bodies and medium-sized shared tables. Finally, EMEM is used for large shared tables.

As packets are received from the network, an FPC thread picks up the packet from the *Distributed Switch Fabric* and processes it (i.e., on-path SmartNIC [6]). Additional threads are allocated to new packets as they arrive. For instance, the SmartNIC NFP-4000 supports up to 60 FPCs, each supporting up to 8 threads. Then, the device is able to process up to 480 packets simultaneously.

3 Related Work

In this section, we discuss the most recent efforts towards P4 SmartNIC offloading and the performance evaluation of them in PDP.

DAIET [15] is a network system that performs in-network data aggregation. It uses Machine Learning (ML) to judiciously decide which partition of the application (e.g., MapReduce) is deployed into PDP to reduce network traffic while maintaining the correctness of the overall computation. FairNIC [6] utilizes SmartNICs to decrease CPU host utilization while providing performance isolation in a multi-tenant environment. It provides an abstraction that allows network applications to access NIC resources. In turn, Clara [13] provides performance clarity for SmartNIC offloading. It analyzes network functions (NFs) and predicts their performance when ported to SmartNIC targets. It uses a logical SmartNIC model to capture SmartNIC architecture. Then, an intermediate representation identifies the code blocks and maps them onto the logical model while optimizing for a performance objective. Finally, it outputs the performance profile for the original NF input on a particular workload. Similarly, SmartChain [18] minimizes the redundant packet transmission by analyzing service function chaining (SFC) forwarding paths and reducing the packet transmissions between the SmartNIC and the host CPU. In the same direction, iPipe [10] allows to offload distributed applications onto SmartNICs. At its core, a scheduler combines first-come & first-serve strategy with deficient round-robin to tolerate applications with variable execution costs.

In addition to the efforts mentioned above towards enabling efficient offloading of network applications to SmartNICs, there are also recent initia-

tives to offload ML algorithms to PDPs. In this context, N2Net [16] and Sanvito et al.[14] represent the first steps toward in-network inference. N2Net [16] is a compiler that generates a P4 program configuration for an RMT-like switch pipeline [3] based on a binary neural network (BNN) model description while Sanvito et al.[14] introduce BaNaNa SPLIT, a system capable of offloading BNNs from CPUs to SmartNICs through a quantization process that transforms the NN model into a format that can be appropriately executed on PDPs. More recently, Xiong et al. [19] propose to deploy trained ML classification algorithms into PDPs. The proposed approach relies on multiple match-action tables and, therefore, is portable between different PDP targets.

As one can observe, most of the existing efforts are still restricted to offloading mechanisms to PDP. However, to the best of our knowledge, these studies do not take into account the current limitations of existing SmartNICs on the offloading process. Therefore, these solutions might either lead to infeasible solutions (e.g., using more resources than available) or suffer high penalties on the expected performance. One noticeable exception is the recent study conducted by Harkous et al. [7]. They evaluate different P4 programs and their impact on the packet processing latency. They gradually increase the complexity of a SmartNIC pipeline (i.e., including parser, control blocks, and deparser) to identify the most influential variables for predicting packet latency. Despite this effort, the work conducted by [7] is still full of gaps considering key building blocks of complex P4 programs (e.g., registers, cryptography functions, packet recirculation, or multiple tables) – which are essential in P4 applications (especially ML applications). In this work, we take a step further into thoroughly understanding the performance of SmartNICs and their existing limitations.

4 Deployment Evaluation

In this section, we evaluate the performance of Netronome SmartNIC for P4 programs concerning their properties and achieved throughput and latency. We start describing our environment setup and performance metrics, followed by the discussion of results.

4.1 Setup

Our environment setup consists of two high-end servers. Each server has an Intel Xeon 4214R processor with 32 GB RAM. One server is our Device Under Test (DUT) – i.e., the server in which P4 programs are loaded – and the other is our traffic generator. Both servers have a Netronome SmartNIC Agilio CX 10 Gbit/s network device with two network interfaces, which are

physically connected. We use MoonGen[5] as our DPDK[1] traffic generator. We instruct MoonGen using the Netronome Packet Generator[2]. In our experiments, we send IPv4 packets at line rate (i.e., 10Gbit/s) with random source and destination prefixes. For our evaluation, we varied the packet size from 64B to 1500B. All experiments were run at least 30 times to ensure a confidence level higher than 90%.

4.2 Metrics

In our evaluation, we aim to measure the performance of P4 programs with respect to the achieved throughput and latency, and identify existing hardware limitations. We evaluate the impact on those metrics regarding (*i*) the number of operations on registers, (*ii*) the number of access to match+action tables, (*iii*) the number of packet recirculation, (*iv*) the number of applied cryptography functions, and (*v*) the number of performed arithmetic operations. To evaluate these metrics, we automatically generate P4 codes with the properties to be analyzed. All P4 codes have at least one match+action table used to perform IP forwarding between physical ports. After generating the P4 source codes, we compiled them using the Netronome compiler and loaded the generated firmware into the physical SmartNIC. Then, we pump network traffic with MoonGen and collect the obtained throughput and latency. To measure data plane latency (i.e., the amount of time a packet stays on PDP), all P4 programs have at least one register which keeps that information (i.e., the difference between the ingress and egress timestamps). During our experiments, we read that register data using the Netronome CLI. The obtained throughput (measured in packets per second) is obtained directly from MoonGen. In order to foster reproducibility, our experimental codes are public available[3]. It is important to note that other SmartNICs and compilers can be easily adapted to our experimental codes.

4.3 Results

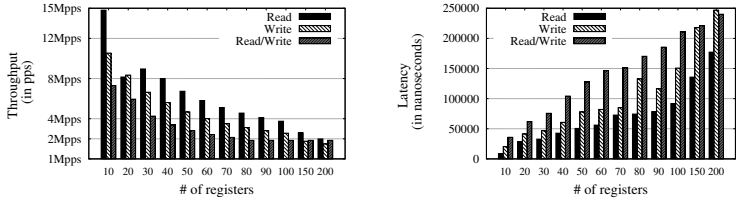
4.3.1 The cost of reading and writing at multiple registers

We start by analyzing the cost of performing multiple register operations in the P4 pipeline. Register operations are one of the main building blocks of recent P4 applications (e.g., [1, 17, 19]). In the experiments, we varied the number of register operations performed sequentially by the P4 program from 10 to 200 registers while varying the register width from 32- to 512-bit words.

[1] `https://www.dpdk.org/`

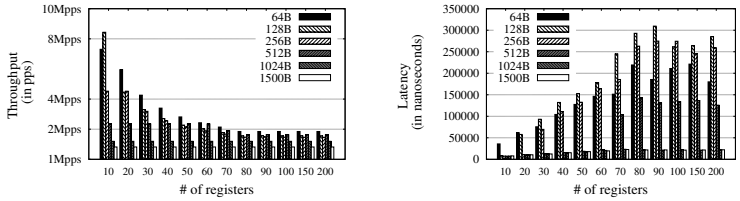
[2] `https://github.com/emmericp/MoonGen/tree/master/examples/netronome-packetgen`

[3] `https://github.com/mcluizelli/performanceSmartNIC`



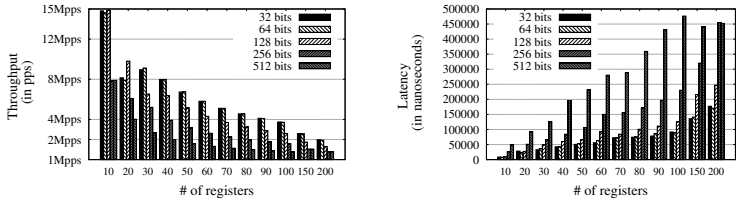
(a) Measured throughput for different register operations (packet size 64B).

(b) Measured latency for different register operations (packet size 64B).



(c) Measured throughput for different packet sizes.

(d) Measured latency for different packet sizes.



(e) Measured throughput for different register width.

(f) Measured latency for different register width.

Fig. 2 Measured throughput and latency for register operations.

We consider that registers are placed in the ingress pipeline and are either read, write, or read & write. Our goal is to quantify the impact on throughput and latency, as well as to quantify the existing limitation of the current architecture. Figure 2 illustrates the measured throughput (in packets per second) and latency (in nanoseconds).

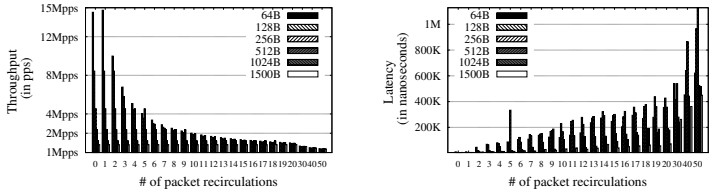
Packet intense line-rate network throughput. Figures 2(a) and 2(b) depict the measured throughput and latency, respectively, for small packets (64 Bytes) and register width of 32 bits. As the number of registers increases (and consequently the number of P4 instructions), there is a sharp performance degradation on observed throughput and latency. The line rate for 10 Gbit/s (i.e., 14.88 million packets per second – Mpps) can be sustained for reading operations to only 10 registers (Figure 2(a)). Even with 10 registers, the bandwidth degradation for writing operations is 30% (and it reaches 50% for read & write) – as they demand more micro engine cycles to be

performed. After, we observe a linear decrease of up to 87% (i.e., 2 Mpps) considering 200 registers. In Figure 2(b), the latency increases linearly as the number of operations performed by each processed packet in the pipeline. For a small number of registers (i.e., 10), there is an acceptable overhead of 8650ns (reading), 20296ns (writing), and 35839ns (both operations). However, this overhead can be as high as 0.12 and 0.24 milliseconds for 50 and 200 registers, respectively.

Line rate network throughput for different packet sizes. Figures 2(c) and 2(d) depict the measured throughput and latency, respectively, for different packet sizes (from 64B to 1500B). We fixed the register width to 32 bits and the register operation to read & write (as it is the most resource-consuming one). Observe that the larger the packet size (and, consequently, the less the number of packets per second to achieve 10Gbit/s), the more register operations can be sustained at line rate. For instance, to 512B-size packets, the line rate throughput can be sustained up to 60 registers being read and write (2Mppps). For 1024B-size packets and higher, there is no throughput degradation – even for 200 registers. Although there is no throughput degradation, Figure 2(d) illustrates that per-packet latency increases substantially. For instance, for 1500B-size packet, the latency increases up to 3X (from 7983ns to 22172ns), while for small packet sizes (and, consequently, higher packet throughput), this latency overhead can be as high as 0.25 millisecond per packet (i.e., a 34X increase).

Line rate network throughput for different register width. Figure 2(e) and Figure 2(f) illustrate the throughput and the latency for a varying width of registers. We fixed the packet size to 64B and the operation to read – as the goal is to quantify the performance degradation with respect to the achieved line rate. As one can observe, the line-rate operation is kept for a register width of up to 128 bits (and 10 registers). Larger register width demands more cycles to fetch the data from memory. As discussed in Section II, Netronome SmartNIC follows a 32-bit architecture. Therefore, any register width wider than that requires extra cycles to be fetched. In addition to that, it is important to mention the memory hierarchy. The more register is needed, the more external memories are used – which directly affects the time to fetch data.

Current limitations. *Is there any limitation on operating registers in a SmartNIC?* In our experiments, we were able to define at most six arrays of 32-bit registers, each having 130M positions. This limitation is due to the amount of memory available on the board (see Section II). Although we could instantiate such a large number of registers, we could not access all of them in a single pipeline pass because micro engines have a limited code space to store the instruction set (i.e., at most 8K instruction). Differently from traditional languages, P4 does not have go-to primitives, and therefore all instructions are defined at compiling time. For that reason, we were able to



(a) Measured throughput. (b) Measured latency.

Fig. 3 Analysing the effect of packet recirculation in the P4 pipeline.

operate at most 200 registers in a single P4 program (considering that our P4 program has also other instructions to perform the forwarding).

4.3.2 Impact of packet recirculation

Next, we evaluate the impact of performing packet recirculation in the P4 pipeline. As previously mentioned, the P4 language does not support iteration-based structures, and therefore, packet recirculation has been used as a way to circumvent such limitations. In short, packet recirculation consists of sending a packet back to the ingress pipeline after processing it (and therefore, mimic a loop-based structure). In the experiments, we varied the number of packet recirculation made (from 0 to 50) in each packet while varying the packet size (from 64B to 1500B). We consider that our P4 program is just forwarding network traffic from physical interfaces. Figure 3 illustrates the measured throughput and latency.

Figure 3(a) illustrates the throughput behavior as the number of packet recirculations increases (it decreases in a super-linear manner). For small-packet network traffic, we observe that fewer packet recirculations are sustained at line rate (e.g., three packet recirculations for 128B size packets). In contrast, for larger packets (e.g., 1024B), it can maintain line rate even for up to two dozen packet recirculations. As the packets are recirculated, more packets are being pushed into the data plane – which enqueue them, and eventually dropped occurs (reducing throughput). In turn, Figure 3(b) depicts the observed latency. As one can observe, there is a non-negligible increase in latency as packets recirculate. For instance, even for large packets (e.g., 1500B) – in which we observe little or no throughput degradation, the per-packet latency doubles on performing just 3 packet recirculations (from 5610ns to 10688ns). This overhead is even sharper as the number of recirculations increases. For example, there is a 6x latency increase (32762ns) for doing 10 recirculations, and an 80x latency increase (450062ns) for 50 recirculations. As one can observe, this overhead is even greater for small packets.

Current limitations. In our experiments, we observe that the Netronome architecture does not allow for recirculating custom-made metadata struc-

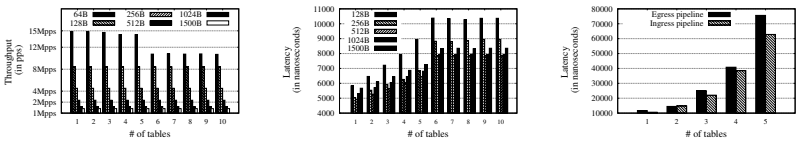
ture. This limits substantially the ability to write complex P4 programs – especially the ones using packet recirculation to circumvent the lack of iteration-based structure.

4.3.3 Impact of using multiple tables

We evaluate the impact of using multiple match-action tables. Unlike traditional forwarding devices that use match+action tables exclusively for routing (i.e., to look up network addresses), the P4 language has opened up new possibilities for this construction type. For instance, Xiong et al. [19] have used multiple tables to implement data plane clustering approaches (e.g., k-means with a table per cluster). Similar to the work conducted by Harkous et al. [7], we also observe that the performance of P4 programs is not affected by the size of tables – as a hash-based data structure implements them. Usually, large match+action tables are already placed on larger and slow memories (e.g., DRAM). Here, instead, we aim at analyzing the impact of using multiple match+action tables at different stages of the pipeline. In the experiments, we varied the number of existing tables in our P4 programs (from 1 to 10), and we ensure that every packet is always matched sequentially on all tables. An action is invoked to read a single 32-bit data from the table and store it in a metadata structure on a packet matching. We varied the packet size (from 64B to 1500B) and the number of tables per pipeline (either on the ingress or egress pipeline). Figure 4 illustrates the measured throughput and latency.

Figure 4(a) illustrates the measured throughput for an increasing number of match+action tables. As observed, the throughput for 64B packets (most packet intensive network traffic) is almost negligible for up to 5 match+action tables (i.e., it keeps the line rate). However, we observe an abrupt decay after 5 tables, followed by a constant throughput behavior (up to 10). In the Netronome architecture, a P4 program can only have 5 tables in each pipeline (ingress/egress). Therefore, tables $1-5$ are located in the ingress pipeline, while $6-10$ in the egress. As all the memory is statically allocated for a P4 program, the Netronome compiler tends to use faster, closer available memory to micro engines to allocate ingress tables. Even when defining tables only in the egress pipeline, the compiler tends not to use faster memory. We empirically show this behavior in Figure 4(c). We incrementally place 5 tables either in the ingress or egress pipeline. We observe that the ingress pipeline is always faster to use available tables (w.r.t. latency) – even in the cases where no tables are used in the ingress. Last, Figure 4(b) illustrates the per-packet latency for an increasing number of tables (both in the ingress/egress pipeline). On average, there in an increase of 40-50% in the latency in the ingress pipeline (between 1 and 5 tables).

Current limitations. As mentioned, the Netronome architecture poses a limit on the number of match+action tables one can use in a P4 program (i.e., at most 5 in each pipeline). That limits the applicability of more complex



(a) Measured throughput. (b) Measured latency. (c) Measured latency in the egress/ingress pipeline.

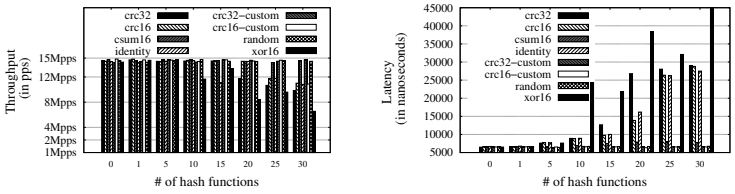
Fig. 4 Analysing the effect of using multiple tables on the P4 pipeline.

algorithms (e.g., [7]) in SmartNICs. Further, we also observe performance differences w.r.t. to the incurred latency (i.e., tables placed in the ingress behave 20% faster).

4.3.4 Impact of using cryptography functions and arithmetic operations

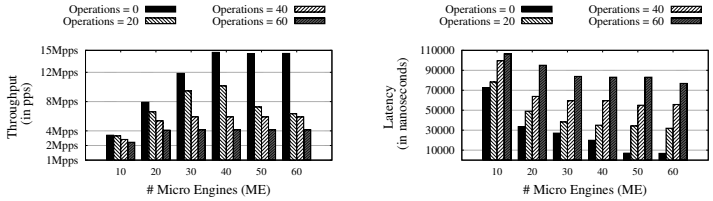
Next, we evaluate the impact of using cryptography functions in our P4 programs. Bloom filters or hash-based data structures widely use them (e.g., to identify heavy-hitters [1]). Cryptography functions are target-dependent (i.e., the implementation depends on the hardware), and in the Netronome architecture, they are implemented by specific micro engines (named Crypto in Figure 1). Netronome implements eight hash functions: (*i*) `crc32`, (*ii*) `crc16`, (*iii*) `identity`, (*iv*) `csum16`, (*v*) `crc32custom`, (*vi*) `crc16custom`, (*vii*) `random`, and (*viii*) `xor16`. In the experiments, we analyse the impact of applying consecutive calls to these cryptography functions (from 0 to 30) in each packet being processed. For the purpose of this experiment, we keep the packet size in 64B and consider that our P4 program just forward network traffic between physical interfaces. Figure 5 illustrates the measured throughput and latency.

Figures 5(a) and 5(b) illustrate the throughput and latency, respectively. As one can observe, only three out of the eight cryptography functions (`crc32-custom`, `random`, and `csum16`) do not lead to performance degradation w.r.t. throughput and latency when increasing the number of calls to them. The remaining ones (i.e., `crc32`, `crc16`, `identity`, `crc16-custom`, and `xor16`) lead to some performance degradation from applying 10 cryptography functions (e.g., 23% of throughput degradation for applying `xor16`). This overhead is even higher for 30 cryptography functions (up to 55% overhead for cryptography function `xor16`). In turn, Figure 5(b) illustrates the incurred per-packet latency. For up to 10 cryptography functions, the per-packet latency remains acceptable (i.e., below 10000ns). However, on applying higher number of cryptography functions (e.g., from 15-20 and on), the latency cost grows exponentially. For instance, the `xor16` function reaches up to 7X higher latency (applying 30 functions) in comparison to the simple forwarding (the case of 0 cryptography functions). We further evaluate the impact of using



(a) Measured throughput. (b) Measured latency.

Fig. 5 Analysing the effect of applying multiple hash functions on the P4 pipeline.



(a) Measured throughput. (b) Measured latency.

Fig. 6 Analysing the effect of using multiple Micro Engines (CPU cores) to process network traffic.

arithmetic operations in our P4 programs (i.e, $+$, $-$, $*$, $/$, $\%$, and $<<$ (bit-shifting)). In the experiments, we analyse the impact of applying consecutive arithmetic operations (varying from 10 to 10000 operations) for intensive packet processing (i.e., 64B packets). We also consider that our P4 program just forwarding network traffic between physical interfaces. In this experiment, we do not observe any statistically significant latency or throughput degradation.

Current limitations. Netronome architecture poses a few limitations restricting the design of more complex P4 programs to its SmartNICs. For instance, multiplication and division are only performed over an integer. Further, the architecture restricts multiplication and division operations of fixed-size operands (i.e., at most 32 bits operands). This limits, for instance, the implementation of precise fixed-point representation for real numbers. Another limitation is the bit shifting operation. The current architecture requires bit-shifting to be static-compiled with predefined values – limiting its applicability. Last, the number of arithmetic operations is limited to 10000 operations per pipeline – related to the number of instructions a micro engine can store.

4.3.5 Analysing used cores and energy consumption.

Last, we evaluate the impact of varying the number of micro engines used by the SmartNIC (from 10 to 60 ME). The goal is to verify whether or not it affects the obtained performance. We keep the packet size at 64B for this experiment and consider that our P4 program forwards network traffic between physical interfaces. We varied the number of reading operations (from 0 to 60) in existing 32-bit registers to stress out the hardware. Figure 6(a) illustrates the measured throughput. As expected, the more ME is available, the more throughput is achieved (in general). However, we can observe that the SmartNIC does not need all ME working in parallel for the evaluated workload. For instance, for 0 read operations, the line rate throughput is achieved using 40 ME. When considering the case of 60 read operations, we observe that more than 20 ME does not affect the performance. Yet, we also observe that allocating a higher number of ME is not always the best strategy. In some cases (e.g., for 20 read operations), the performance is worsened by increasing the number of MEs from 40 to 50-60. Figure 6(b) depicts the measured latency. As one can observe, there is always a latency reduction when increasing the available ME (even when there is no improvement in the throughput). Finally, we evaluated the energy consumption. In our experiments, the energy consumption varied 0.2 Watt between using 10 ME and 60 ME.

5 Final Remarks

In this paper, we performed an extensive performance evaluation of SmartNICs to understand and quantify existing limitations. We focus our evaluation on measuring the performance in terms of latency and throughput for a plethora of packet memory-intensive scenarios. We showed that the line-rate throughput is bounded by (i) the number of register operations (up to 10 operations), (ii) the number of multiple match+action tables user in the pipeline (up to 5), (iii) the number of cryptography operations (up to 10). As future work, we intend to build an analytical model that can accurately estimate the performance of P4 applications executing on SmartNICs.

Acknowledgements

This work was partially funded by National Council for Scientific and Technological Development (CNPq) (grant 427814/2018-9), São Paulo Research Foundation (FAPESP) (grant 2018/23092-1), Rio Grande do Sul Research Foundation (FAPERGS) (grants 19/2551-0001266-7,20/2551-000483-0, 19/2551-0001224-1)

References

1. Ben Basat, R., Einziger, G., Friedman, R., Luizelli, M.C., Waisbard, E.: Constant time updates in hierarchical heavy hitters. In: Proceedings of the ACM SIGCOMM. p. 127–140. SIGCOMM '17, ACM, New York, NY, USA (2017)
2. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., Walker, D.: P4: Programming protocol-independent packet processors. ACM SIGCOMM 14 **44**(3), 87–95 (Jul 2014)
3. Bosshart, P., Gibb, G., Kim, H.S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., Horowitz, M.: Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. ACM SIGCOMM CCR **43**(4), 99–110 (2013)
4. Castro, A.G., Lorenzon, A.F., Rossi, F.D., Da Costa Filho, R.I.T., Ramos, F.M.V., Rothenberg, C.E., Luizelli, M.C.: Near-optimal probing planning for in-band network telemetry. IEEE Communications Letters pp. 1–1 (2021)
5. Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., Carle, G.: Moongen: A scriptable high-speed packet generator. In: Proceedings of the ACM IMC. p. 275–287. IMC '15, ACM, New York, NY, USA (2015)
6. Grant, S., Yelam, A., Bland, M., Snoeren, A.C.: Smartnic performance isolation with fairnic: Programmable networking for the cloud. In: Proceedings of the ACM SIGCOMM. pp. 681–693 (2020)
7. Harkous, H., Jarschel, M., He, M., Priest, R., Kellerer, W.: Towards understanding the performance of p4 programmable hardware. In: ACM/IEEE Symposium on Architectures for Networking and Communications Systems. pp. 1–6. IEEE (2019)
8. Hohemberger, R., Castro, A.G., Vogt, F.G., Mansilha, R.B., Lorenzon, A.F., Rossi, F.D., Luizelli, M.C.: Orchestrating in-band data plane telemetry with machine learning. IEEE Communications Letters **23**(12), 2247–2251 (Dec 2019)
9. Hohemberger, R., Lorenzon, A.F., Rossi, F.D., Luizelli, M.C.: A heuristic approach for large-scale orchestration of the in-band data plane telemetry problem. In: Barolli, L., Amato, F., Moscato, F., Enokido, T., Takizawa, M. (eds.) Advanced Information Networking and Applications. pp. 381–392. Springer International Publishing (2020)
10. Liu, M., Cui, T., Schuh, H., Krishnamurthy, A., Peter, S., Gupta, K.: Offloading distributed applications onto smartnics using ipipe. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 318–333 (2019)
11. Netronome: Internet (2020), `https://www.netronome.com/static/app/img/products/silicon-solutions/WP_NFP4000_TOO.pdf`
12. Pizzutti, M., Schaeffer-Filho, A.E.: Adaptive multipath routing based on hybrid data and control plane operation. In: IEEE INFOCOM. pp. 730–738 (2019)
13. Qiu, Y., Kang, Q., Liu, M., Chen, A.: Clara: Performance clarity for smartnic offloading. In: Proceedings of the ACM Hot Topics in Networks. pp. 16–22 (2020)
14. Sanvito, D., Siracusano, G., Bifulco, R.: Can the network be the ai accelerator? In: Proceedings of the Workshop on In-Network Computing. pp. 20–25 (2018)
15. Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M., Kalnis, P.: In-network computation is a dumb idea whose time has come. In: Proceedings of the 16th ACM Workshop on Hot Topics in Networks. pp. 150–156 (2017)
16. Siracusano, G., Bifulco, R.: In-network neural networks. arXiv preprint arXiv:1801.05731 (2018)
17. Wang, S.Y., Wu, C.M., Lin, Y.B., Huang, C.C.: High-speed data-plane packet aggregation and disaggregation by p4 switches. Journal of Network and Computer Applications **142**, 98 – 110 (2019)
18. Wang, S., Meng, Z., Sun, C., Wang, M., Xu, M., Bi, J., Yang, T., Huang, Q., Hu, H.: Smartchain: Enabling high-performance service chain partition between smartnic and cpu. In: IEEE International Conference on Communications. pp. 1–7. IEEE (2020)
19. Xiong, Z., Zilberman, N.: Do switches dream of machine learning? toward in-network classification. In: Proceedings of the 18th ACM Workshop on Hot Topics in Networks. pp. 25–33 (2019)

INDEX

AMD, 47
API, 30
ASIC, 30

BGP, 33

DCN, 34
DDoS, 39, 66
DFS, 25
DPDK, 29

FP^{2}INT, 11, 13, 26, 53, 65, 66
FPGA, 29, 30

HPCC, 38

IBM, 47
IETF, 23
INT, 11, 13, 23–26, 31–33, 35–38, 53, 55, 59, 60, 67
ISP, 23, 34

KDN, 36

MDT, 33
MIB, 31
MILP, 25, 65
MTU, 33, 42, 54

NIC, 30
NP, 26
NPU, 29
NSH, 33

OID, 31
ONOS, 35, 36

P^{2}INT, 11, 13, 25, 26, 41, 44, 46–51, 65, 66
P4, 23, 29, 30, 33, 36
POF, 29
PTP, 36

QoE, 23

RAM, 47
RTT, 27

SDN, 36
SLA, 23, 27, 66
SNMP, 23, 27, 30, 31

TCP, 32, 33

UDP, 31, 32

VNS, 26