UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JONATAS ADILSON MARQUES

Advancing Network Monitoring and Operation with In-band Network Telemetry and Data Plane Programmability

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Luciano Paschoal Gaspary

Porto Alegre
May 2022

CIP — CATALOGING-IN-PUBLICATION

Marques, Jonatas Adilson

Advancing Network Monitoring and Operation with In-band Network Telemetry and Data Plane Programmability / Jonatas Adilson Marques. – Porto Alegre: PPGC da UFRGS, 2022.

156 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Luciano Paschoal Gaspary.

1. Network Monitoring. 2. Software-Defined Networking. 3. Data Plane Programmability. 4. P4. 5. In-band Network Telemetry. I. Gaspary, Luciano Paschoal. II. Título.

ABSTRACT

Modern communication networks operate under high expectations on performance and resilience (e.g., latency, bandwidth, availability) mainly due to the continuous proliferation of non-elastic highly-distributed applications. In this context, closely monitoring the state, behavior, and performance of networking devices and their traffic as well as quickly troubleshooting problems as they arise is essential for the operation of network infrastructures. Unfortunately, existing tools and techniques fall short at providing the required level of detail, enabling quick reactions, and keeping monitoring overhead from affecting the network operation. Data Plane Programmability (DPP) along with In-band Network Telemetry (INT), backed by the recent advances in Software-Defined Networking, emerge in this context as promising platforms to meet these monitoring demands. INT enables unprecedented monitoring accuracy and precision, but may lead to performance degradation if applied indiscriminately to all packet flows in a network. One alternative to avoid this issue is to orchestrate telemetry tasks and use only a portion of traffic to monitor the network via INT. The general problem consists, then, in assigning subsets of traffic to carry out INT and provide full monitoring coverage while minimizing the overhead. To achieve this goal, as a first step in this thesis, we introduce and formalize the In-band Network Telemetry Orchestration (INTO) problem, prove that it is NP-Complete, and propose polynomial computing time heuristics to solve it. In our evaluation using real wide-area network topologies, we observe that the heuristics produce solutions close to optimal to any network in under one second. We also observe that networks can be covered assigning a linear number of flows in relation to the number of device interfaces and, finally, that it is possible to minimize telemetry load to one interface per flow for most networks. Continuing our work, we investigate DPP capabilities further and design INTSIGHT, a system for highly accurate and fine-grained detection and diagnosis of SLO violations. The main contribution of INTSIGHT is, building upon in-band telemetry, introducing path-wise computation of network metrics and selective generation of reports. We show the effectiveness of INTSIGHT by way of two use cases. Our evaluation using real networks also shows that INTSIGHT generates up to two orders of magnitude less monitoring traffic than state-of-the-art approaches. Furthermore, its processing and memory requirements are low and therefore compatible with currently existing programmable platforms. As a final step in this thesis, we shift our focus to quick reaction and propose FELIX, a system for failure recovery that reroutes around failures at data-plane timescales

while still using the shortest available paths. Our evaluation shows that our approach can recover from failures up to four orders of magnitude faster than existing SDN approaches while making sensible use of data-plane resources. Finally, with the design of FELIX, we introduce the Strategy-Tactic paradigm to enable data-plane timescale reactions with control-plane decisions based on a global understanding of the network to general network operation tasks. We argue the generality of this paradigm by discussing the main challenges involved in modeling a promising use case.

Keywords: Network Monitoring. Software-Defined Networking. Data Plane Programmability. P4. In-band Network Telemetry.

Avançando o Monitoramento e Operação de Redes com Telemetry In-band e Programabilidade do Plano de Dados

RESUMO

As redes de comunicação modernas operam sob altas expectativas de desempenho e resiliência (por exemplo, latência, largura de banda, disponibilidade),isto principalmente devido à contínua proliferação de aplicações não elásticas altamente distribuídas. Nesse contexto, monitorar de perto o estado, o comportamento e o desempenho dos dispositivos de rede e seus tráfegos, bem como solucionar rapidamente os problemas à medida que estes surgem, são essenciais para a operação das infraestruturas de rede. Infelizmente, as ferramentas e técnicas existentes são limitados no nível de detalhes oferecido, na rapidez de suas reações e na capacidade de manter a sobrecarga de monitoramento baixa o suficiente para não afetar a operação da rede. A Programabilidade do Plano de Dados (do inglês *Data Plane Programmability* – DPP) juntamente com a Telemetria de Redes no modo *In-band* (*In-band Network Telemetry* – INT), respaldadas pelos recentes avanços em Software-Defined Networking, surgem neste contexto como plataformas promissoras para atender a essas demandas de monitoramento. A INT permite alcançar níveis de precisão e granularidade de monitoramento sem precedentes, mas pode levar à degradação do desempenho significante se aplicada indiscriminadamente a todos os pacotes e fluxos em uma rede. Uma alternativa para evitar esse problema é orquestrar tarefas de telemetria e usar apenas uma parte do tráfego para monitorar a rede via INT. O problema geral consiste, então, em atribuir subconjuntos de tráfego para realizar INT e fornecer cobertura total de monitoramento, minimizando o overhead. Para atingir este objetivo, como primeiro passo nesta tese, apresentamos e formalizamos o problema *In-band Network Telemetry Orchestration* (INTO), provamos que ele é NP-Completo e propomos heurísticas polinomiais em tempo de computação para resolvê-lo. Em nossa avaliação usando topologias de redes de larga escala reais, observamos que as heurísticas produzem soluções próximas ao ótimo para qualquer rede em menos de um segundo. Observamos também que as redes podem ser cobertas atribuindo um número linear de fluxos em relação ao número de interfaces dos dispositivos e, por fim, que é possível minimizar a carga de telemetria para uma interface por fluxo para a maioria das redes. Continuando nosso trabalho, investigamos ainda mais os recursos disponíveis na DPP e projetamos o INTSIGHT, um sistema para detecção e diagnóstico altamente precisos de violações de SLO. A principal

contribuição do INTSIGHT é, com base na telemetria *in-band*, introduzir o cálculo de métricas de rede ao longo do caminho dos pacotes e a exportação seletiva de informações para o plano de controle. Mostramos a eficácia do INTSIGHT por meio de dois casos de uso. Nossa avaliação usando redes reais também mostra que INTSIGHT gera até duas ordens de magnitude menos tráfego de monitoramento do que abordagens do estado da arte. Além disso, seus requisitos de processamento e memória são baixos e, portanto, compatíveis com as plataformas programáveis existentes. Como etapa final desta tese, mudamos nosso foco para a reação rápida e propomos o FELIX, um sistema para recuperação de falhas que redireciona o tráfego afetado em escalas de tempo de plano de dados enquanto ainda usa os caminhos mais curtos dentre os disponíveis. Nossa avaliação mostra que nossa abordagem pode se recuperar de falhas até quatro ordens de magnitude mais rapidamente do que as abordagens SDN existentes, ao mesmo tempo em que faz o uso sensato dos recursos do plano de dados. Finalmente, com o projeto de FELIX, introduzimos o paradigma Estratégia-Tática para tarefas gerais de operação de rede que busca permitir reações na escala de tempo de plano de dados com decisões no plano de controle baseadas em uma compreensão global da rede. A generalidade desse paradigma é discutida considerando os principais desafios envolvidos na modelagem de um caso de uso promissor.

Palavras-chave: Monitoramento de Redes, Redes Definidas por Software, Programabilidade do Plano de Dados, P4, In-band Network Telemetry.

LIST OF ABBREVIATIONS AND ACRONYMS

ASIC	Application-Specific Integrated Circuit
BGP	Border Gateway Protocol
BH	Balance Heuristic
BMP	Balance Mathematical Program
BPP	Bin Packing Problem
CDF	Cumulative Distribution Function
CH	Concentrate Heuristic
CMP	Concentrate Mathematical Program
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DARPA	Defense Advanced Research Projects Agency
DCN	Data Center Network
DDoS	Distributed Denial-of-Service
DPP	Data Plane Programmability
DSL	Domain-Specific Language
ECMP	Equal-Cost Multi-Path Routing
FA	Full Assignment
FIFR	Failure-Inferencing Fast Reroute
ForCES	Forwarding and Control Element Separation
FPGA	Field-Programmable Gate Array
IETF	Internet Engineering Task Force
INT	In-band Network Telemetry
INTO	In-band Network Telemetry Orchestration
IP	Internet Protocol

IPFIX	IP Flow Information Export
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ITZ	Internet Topology Zoo
IXP	Internet Exchange Point
LAG	Link Aggregation Groups
LUT	Lookup Table
MAC	Media Access Control
MSP	Multiprocessor Scheduling Problem
MTU	Maximum Transmission Unit
NP	-Complete
NPU	Network Processing Unit
NSH	Network Service Header
NVMe	Non-Volatile Memory Host Controller Interface Specification
OAM	Operations, Administration and Management
OF	OpenFlow
OM	Opportunistic Merging
ONF	Open Networking Foundation
OS	Operating systems
OSPF	Open Shortest Path First
OWAMP	One-Way Active Measurement Protocol
RAM	Random-Access Memory
RCP	Routing Control Platform
RMT	Reconfigurable Match-Action Tables
SDN	Software-Defined Networking
SLF	Switch-local failover

SLO	Service-Level Objective
SNMP	Simple Network Management Protocol
SRAM	Static Random-Access Memory
SSD	Solid State Disk
ST	Strategy-Tactic
TBFF	Tagging-Based Fast Failover
TCAM	Ternary Content-Addressable Memory
TCP	Transmission Control Protocol
TPP	Tiny Packet Programs
TWAMP	Two-Way Active Measurement Protocol
UDP	User Datagram Protocol
VoIP	Voice over Internet Protocol
WAN	Wide-Area Network

LIST OF FIGURES

Figure 1.1 Overview of this thesis with its main foundation blocks and contributions. . 20

Figure 2.1 Traditional SDN architecture. 28
Figure 2.2 SDN architecture extended with programmable data planes and P4. 29
Figure 2.3 P4 code sections, and mapping to the abstract forwarding model (Adapted from Kim and Lee (KIM; LEE, 2016)). 30
Figure 2.4 P4 header declaration examples. 30
Figure 2.5 Architecture of the INT framework for in-band network telemetry (Adapted from Thomas and Laupkhov (THOMAS; LAUPKHOV, 2016)). 32

Figure 3.1 Example of full assignment. 35
Figure 3.2 Concentrated assignment example. 39
Figure 3.3 Example BPP reduction to INTO Concentrate. A BPP instance with $|A| = 3$ items, bin capacity $C = 4$, and number of bins $n = 2$ is reduced to an INTO Concentrate instance with 3 forwarding devices and 2 flows with telemetry capacity equal to 4 items. 41
Figure 3.4 Balanced assignment example. 43
Figure 3.5 Example MSP reduction to INTO Balance. An MSP instance – with $|J| = 3$ jobs, number of processors $m = 2$, and deadline $n = 6$ – is reduced to an INTO Concentrate instance – with 3 forwarding devices and 2 flows with telemetry capacity equal to 6. 45
Figure 3.6 Processing times for the mathematical programming models. 51
Figure 3.7 Processing times for the heuristic algorithms. 52
Figure 3.8 Interface Coverage. 54
Figure 3.9 Mean and confidence interval for flow packet load when the mean telemetry capacity is 35 items. 56
Figure 3.10 Evaluation of flow load as telemetry capacity varies. 56
Figure 3.11 Flow usage as a function of the number of interfaces in a network. 58
Figure 3.12 Mean and confidence interval for information correlation when the mean telemetry capacity is 35 items. 58
Figure 3.13 Evaluation of information correlation as the telemetry capacity of flows varies. 59
Figure 3.14 Mean and confidence interval for information freshness with mean telemetry capacity equal to 35 items. 60
Figure 3.15 Information freshness as a function of the network diameter. 61

Figure 4.1 Overview of INTSIGHT's main procedures in the data and control planes. . . 69
Figure 4.2 Mapping a traffic flow to a register array index (`FlowID`) for metadata storage and analysis. 75
Figure 4.3 Example network for path tracing method. 77
Figure 4.4 Examples field values for pinpointing contentions and suspects. 80
Figure 4.5 End-to-End Delay Case Study. A delay-sensitive flow A–H has its SLO violated due to bursty flow E–G. 86
Figure 4.6 Bandwidth Case Study. A bandwidth-sensitive flow A–H suffers performance degradation from recurrent natural increases in demand for other traffic sharing link $N_3 \rightarrow N_5$. 87
Figure 4.7 Report rate (packets per second). 89
Figure 4.8 Header space usage (bytes). 91

Figure 4.9 Comparison of monitoring techniques considering generated telemetry traffic as a function of network resource usage. 92

Figure 5.1 Failure recovery delay factors for SDN with OpenFlow approaches. 100
Figure 5.2 Limitations to existing failover mechanisms and failure-inferencing approaches in the data plane. 101
Figure 5.3 Example of FELIX's approach to safeguard against and reroute around an example link failure. 103
Figure 5.4 Steps to reroute around a shared-risk failure. 110
Figure 5.5 Downtime with varying detection delay values (entry installation delay fixed at 1 ms). 116
Figure 5.6 Downtime with varying forwarding entry installation delay values (detection delay fixed at 10 ms). 118
Figure 5.7 Downtime factor cost breakdown. 119
Figure 5.8 Main components of the Strategy-Tactic paradigm. 126
Figure 5.9 Steps to reroute around a shared-risk failure. 130

Figure A.1 Sumário da tese com suas principais contribuições e fundamentos. 152

LIST OF TABLES

Table 3.1 Summary of symbols. 36
Table 3.2 Summary of the evaluation results. 62

Table 4.1 Metadata fields maintained by IntSight at ingress devices, telemetry headers, and/or egress devices for each flow. Path-wise fields are marked with an asterisk (*). 71
Table 4.2 Metadata for the network topologies considered on the efficiency evaluation. 84
Table 4.3 Device memory usage (Mb). 90

Table 5.1 Summary of the conceptual comparison of existing approaches for failure recovery and FELIX. 102
Table 5.2 Summary of the networks used for evaluation. 115
Table 5.3 Summary of the scalability results. 121
Table 5.4 Summary of the results for the opportunistic merging memory-minimization procedure 125

CONTENTS

1 INTRODUCTION 17
1.1 Problem Statement 18
1.2 Hypothesis, Research Questions, and Contributions 19
1.3 Organization 23

2 BACKGROUND 25
2.1 Network Programmability 25
2.2 Programmable Data Planes and P4 28
2.3 In-Band Network Telemetry 31

3 ORCHESTRATING IN-BAND NETWORK TELEMETRY 33
3.1 Motivation 33
3.2 INTO 35
3.3 Problem Variations 38
3.3.1 INTO Concentrate 39
3.3.2 INTO Balance 42
3.4 Heuristic Algorithms to Solve the INTO Problems 46
3.4.1 Concentrate Heuristic Algorithm 46
3.4.2 Balance Heuristic Algorithm 48
3.5 Evaluation 49
3.5.1 Mathematical Programming Models and Heuristic Algorithms 50
3.5.2 Comparison of the INTO problem variations 54
3.6 Related Work 62
3.7 Chapter Summary 64

4 DIAGNOSING SLO VIOLATIONS WITH IN-BAND NETWORK TELEMETRY 67
4.1 Motivation 67
4.2 INTSIGHT 68
4.2.1 INTSIGHT Data Plane 69
4.2.2 INTSIGHT Control Plane 73
4.3 Design and Implementation 74
4.3.1 Correlating Events in Time 74
4.3.2 Storing Traffic Metadata Persistently 75
4.3.3 Tracing Packet Paths 76
4.3.4 Encoding Paths 78
4.3.5 Pinpointing Contentions and Suspects 80
4.3.6 Monitoring SLO Compliance 81
4.4 Evaluation 82
4.4.1 Experimental Setup 82
4.4.2 Functional Evaluation 84
4.4.3 Performance Evaluation 88
4.5 Related Work 92
4.6 Additional Remarks 94
4.7 Chapter Summary 98

5 RESPONDING TO NETWORK FAILURES AT DATA-PLANE SPEEDS 99
5.1 Motivation 99

5.2 FELIX 103
5.3 Design and Implementation 105
5.3.1 Forwarding Packets in the Data Plane 106
5.3.2 Handling Local Failures 107
5.3.3 Coordinating Network-Wide Rerouting 109
5.3.4 Planning for Failure Scenarios 111
5.4 Evaluation 114
5.4.1 Experimental Setup 114
5.4.2 Performance 115
5.4.3 Scalability 120
5.5 Related Work 122
5.6 Additional Remarks 123
5.7 On the Generality of the Strategy-Tactic Paradigm 125
5.8 Chapter Summary 132

6 FINAL CONSIDERATIONS 133
6.1 Conclusions 133
6.2 Directions for Future Research 135
6.3 Achievements 137

REFERENCES 139

APPENDIX A — SUMMARY IN PORTUGUESE 149
A.1 Definição do Problema 150
A.2 Hipótese, Questões de Pesquisa e Contribuições 152

1 INTRODUCTION

Current networks operate with high expectations on performance (e.g., latency, bandwidth, availability), especially with the emergence and proliferation of new applications (e.g., algorithmic trading, telesurgery, and virtual reality video streaming) with architectures based on many interconnected components spread across multiple end-points (BALAKRISHNAN, 2021). These applications and their users demand strict requirements, which to be met require defining clear goals for network performance, the so-called service-level objectives (SLOs), and troubleshooting problems that may prevent achieving such goals. Unfortunately, there is a myriad of problems that may impact the correct and efficient operation of a network, ranging from traffic congestions all the way to hardware failures. In this context, monitoring the state, behavior, and performance of networking devices and their traffic is essential for the operation of today's network infrastructures. Nevertheless, network monitoring is an inherently hard task, sometimes compared to searching for a needle in a haystack (ZHU et al., 2015; GUPTA et al., 2018)[1].

Existing tools and techniques are not engineered to monitor networks and help troubleshoot their problems with the required level of detail and accuracy. For example, and regarding the collection of metadata and statistics, traditional passive monitoring tools (e.g., SNMP (CASE et al., 1989) and NetFlow/IPFIX (CISCO, 2005; CLAISE; TRAMMELL; AITKEN, 2013)) operate at coarse timescales (dozens of seconds and up) and, thus, lack adequate granularity to detect events such as short-lived traffic bursts (e.g., microbursts) that may be critical to modern applications. Active measurement techniques (e.g., ping, traceroute, OWAMP (SHALUNOV et al., 2006), and TWAMP (HEDAYAT et al., 2008)) also do not provide sufficient time resolution; additionally, there is no guarantee that the network will route and prioritize probes in the same way as production packets. The consistent move towards heterogeneity in the treatment of traffic (JEYAKUMAR et al., 2014; HONG et al., 2013), multi-path routing (JAIN et al., 2013; KUMAR et al., 2015), and flowlet load balancing (ALIZADEH et al., 2014; HE et al., 2015) exacerbates this limitation. As a second example, and regarding troubleshooting SLO violations, approaches based on packet mirroring (e.g., NetSight (HANDIGOL et al., 2014), Planck (RASLEY et al., 2014), Everflow (ZHU et al., 2015), and Stroboscope (TILMANS et

[1] (ZHU et al., 2015): "[Network troubleshooting] is not only akin to searching in the proverbial haystack for needles, but for specific needles of arbitrary size, shape and color." and (GUPTA et al., 2018): "[...] telemetry queries often require finding 'needles in a haystack' where the fraction of total traffic or flows that satisfies these queries is tiny."

al., 2018)) can help give visibility into the network to understand how packets are being processed and forwarded by devices. These approaches find their main challenge in keeping the monitoring overhead (i.e., required bandwidth and processing) under reasonable levels while still collecting fine grained data (TILMANS et al., 2018). Packet sampling, their common method for addressing this challenge, inherently leads these techniques to miss important events; deciding what and when to sample is hard.

Another challenge in meeting SLOs in modern times is the common dependency and delay in communication between the mechanisms that detect problems and the ones that find the solution to these problems. Consider, for example, the case of equipment failures and their impact on network availability. Existing solutions depend on some type of computation in control plane at the time of failure and subsequent reconfiguration of forwarding tables. Computing the new forwarding entries for devices or, in broader terms, the solution to the problem can take considerable time. As the delay in reacting to failures leads to a significant number of packet drops, in the general case, the delay caused by the use of long control loops to solve problems can lead to substantial performance and financial loss. Ideally, the data plane should be able to react immediately at the time of failure. We note that the limitations and drawbacks presented by the existing monitoring tools and techniques result from the low level of flexibility in defining how packets are to be processed by the data plane in traditional networks. Even with Software-Defined Networking (SDN) and OpenFlow (MCKEOWN et al., 2008), there is only support for standardized protocols; there is no freedom to define customized protocols and procedures.

1.1 Problem Statement

As a result to the presented scenario, the networking community has sought for more flexibility in the data plane, which recently culminated in the proposal of data plane programmability (DPP) (BOSSHART et al., 2013; CHOLE et al., 2017; OZDAG, 2012). DPP reshapes the SDN landscape by enabling network operators to reprogram forwarding devices in-field to deploy novel networking protocols, customize the network behavior, and consequently develop and support innovative services and applications. Protocols and packet processing procedures, in this new paradigm, are defined via domain-specific languages – e.g., P4 (BOSSHART et al., 2014) – with support for abstractions to specify customized protocol headers, parsing logic, and match-action tables, for example. One interesting concept that gained traction with the introduction of programmable data

planes is In-band Network Telemetry (INT) (JEYAKUMAR et al., 2014; KIM et al., 2015; THOMAS; LAUPKHOV, 2016). Within this concept, forwarding devices are programmed to annotate production packets with metadata regarding their state, behavior, and performance (such as port utilization, matched forwarding entries, and queuing delays). The annotated information is accumulated in a packet along its path and, at some point in the network, extracted and reported to analyzer servers. These servers piece together the received information (as needed) to build an accurate and global view of the network, as observed by its traffic.

INT-based techniques have shown to produce monitoring data with an unprecedented level of accuracy and fine granularity (KIM et al., 2015; ZHANG et al., 2017). That is because instead of relying on active probes, which may be subject to forwarding and routing behaviors different from those of the traffic of interest, the production packets themselves can be used to probe the network. Moreover, metadata collection can be made precisely during the instants when individual packets of interest are being processed at a device. As a consequence, INT makes it possible to detect and pinpoint network events that were previously imperceptible, such as microsecond congestions.

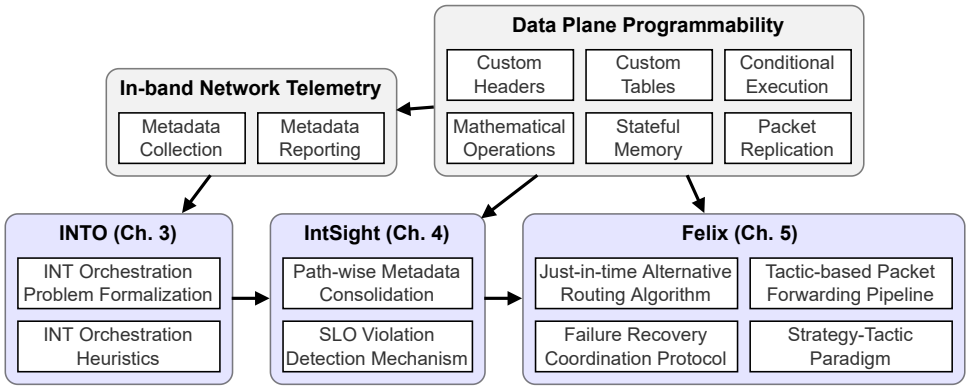
Although DPP brings greater flexibility to the development of monitoring mechanisms, to operate at line rate on high-speed links, data plane programs are constrained to a small time budget (dozens of nanoseconds) and a limited memory space (e.g., hundreds of megabits of SRAM and dozens of megabits of TCAM) (BOSSHART et al., 2013). Regarding INT, since it involves modifying production packets traversing the network, the amount of metadata that may be collected by a packet is limited by its original size and the network maximum transmission unit (MTU). Additionally, some of the performed actions may increase network load and impact performance. For example, embedding telemetry data into packets increases the load on network links, and generating report packets increases the load on forwarding devices and control channels. We argue that these constraints and factors need to be carefully considered in order to enable the full potential of data plane programmability to the discipline of network monitoring.

1.2 Hypothesis, Research Questions, and Contributions

This thesis seeks to bridge the gap to materializing the new opportunities for network monitoring and operation brought up by Data Plane Programmability (DPP) and In-band Network Telemetry (INT). Our hypothesis is that INT along with DPP can be

successfully applied to monitor and operate networks with per-packet granularity as well as practicable overheads. We make eight valuable contributions that answer important questions about effectively and efficiently managing networks in this new paradigm[2]. Figure 1.1 summarizes our work, showing its main foundation blocks originated from the emergence of Data Plane Programmability and In-band Network Telemetry as well as its main contributions. These contributions are positioned as a result of three research questions that we describe next.

Figure 1.1: Overview of this thesis with its main foundation blocks and contributions.



Question 1. *How can In-band Network Telemetry collection actions be orchestrated across devices in a network to maximize measurement quality while minimizing network and traffic overheads?*

Our initial investigation into In-band Network Telemetry showed that when such concept is applied unsystematically to collect metadata from each one of the devices visited by each of the traffic packets traversing the network, substantial burden is placed on the traffic, network resources, and analysis servers. The overhead imposed by this burden can arrive at the point of halting the operation of the network or its effective monitoring.

In Chapter 3, as our **first contribution**, we formalize what we call the In-band Network Telemetry Orchestration (INTO) problem by means of Integer Linear Programming. Our ultimate goal with INTO is to minimize monitoring overheads while still obtaining high-quality data. We formalize two variations of the INTO problem as mathematical programming models, each of them focusing on optimizing the usage of a specific resource: the packet processing capacity of devices and the bandwidth of links, respectively. We also prove that both variations of the orchestration problem are NP-Complete. Through an extensive evaluation using real network topologies, we confirme that generating optimal

[2]In this thesis, we generally focus on greenfield deployments, that is, scenarios where all forwarding devices in the network are P4-programmable. Nevertheless, where appropriate, we discuss opportunities for brownfield deployment and the effects of legacy devices on the task at hand.

solutions takes prohibitive amounts of time.

As the **second contribution**, also in Chapter 3, we address the scalability limitation of the mathematical programming models by designing heuristic algorithms. These algorithms are capable of computing high-quality solutions in polynomial computing time to the two variations of the INTO problem. Through our evaluation, we observe that the proposed heuristics are able to generate close to optimal solutions for all of the considered topologies under a second. We also evaluate the quality and costs associated with the proposed heuristics under different aspects and compare their results to identify what types of networks each heuristic is better suited to monitor.

In the following part of this thesis, we continue this study on INT by bringing DPP into focus and answering the following intriguing question.

Question 2. *Given the flexibility in packet processing provided by Data Plane Programmability, can monitoring data be pre-processed or consolidated by forwarding devices before being reported to the control plane to further reduce overhead and with no loss (or even improvement) to measurement quality?*

One of the observations from our study of INT orchestration is that programmable data planes have capabilities that could allow for monitoring actions beyond raw metadata collection. Starting from this observation, in Chapter 4, we present INTSIGHT, a system that fully exploits the capabilities of network programmability (BOSSHART et al., 2013; BOSSHART et al., 2014) to monitor SLOs related to end-to-end delay and bandwidth guarantees. In a nutshell, INTSIGHT discretizes time into sub-second, fixed-length slices called epochs. Network and flow-of-interest traffic are monitored on a per-packet basis by the data plane to track their state, behavior, and performance (e.g., routing, contention, delay, packet drops, and provided bandwidth). Each production packet is instrumented to carry essential information (in a telemetry header), which has its values systematically updated (through in-network computation) as the packet moves towards the destination. Egress forwarding devices consolidate this information temporarily in memory. At the end of each epoch, the consolidated information kept for each flow of interest enables detecting and diagnosing SLO violation events, their causes, victims, and culprits. As a **third contribution** of this thesis, thus, we design and implement efficient data plane procedures for gradually computing path-wise metadata such as paths, contention points, end-to-end delays, and provided bandwidth. In our evaluation, we demonstrate the benefits (regarding functionality, performance, and resource footprint) of path-wise in-band network telemetry compared to state-of-the-art approaches, considering six representative

network topologies.

The positive answer to the second question – i.e., the success in pre-processing and consolidating monitoring data directly in the data plane – motivates our last question.

Question 3. *Can part of the analysis and reaction logic (traditionally placed in the control plane) be offloaded to the data plane to enable detecting and reacting to network problems in shorter timescales?*

We describe cases in which we find the answer to this question to be positive. First, as briefly mentioned, the consolidated information stored in forwarding devices under INTSIGHT enables detecting and diagnosing SLO violation events. During the design of INTSIGHT we observed that the way information is consolidated would enable the forwarding devices themselves to detect epochs in which SLO violations are present. As a result, our **fourth contribution** is an in-network, distributed, path-aware mechanism for monitoring network traffic capable of fine-grained and timely detection of SLO violations and other problems impacting performance (e.g., microbursts). In Chapter 4, when presenting INTSIGHT, we describe how, at the end of each epoch, deviations from what is expected from the network (e.g., SLO violations) are detected by egress forwarding devices, which triggers the generation of reports. Control plane servers receive and analyze the generated reports to identify the culprit traffic disrupting network operation. In our evaluations, this approach to telemetry data reporting has shown to make judicious utilization of control plane bandwidth and analysis server resources without significant loss in accuracy and detail.

Following our work on INTSIGHT, in Chapter 5, we present another step taken towards answering Question 3. We shift our focus to network equipment failures and investigate ways for more efficient and resilient rerouting to meet availability SLOs that arise in the context of SDN with programmable data planes. We propose FELIX, a novel system that proactively computes forwarding tactics for the normal network state as well as failure scenarios in the control plane and programs these tactics on data plane devices along with a lightweight coordination protocol to immediately react to failures. In a sense, the control plane acts as a strategist devising recovery tactics to handle failures, while the data plane carries out these tactics accordingly when needed. This approach to the problem eliminates the need to wait for the control plane to compute and install new entries upon a failure while also enabling the use of the best alternate paths to bypass failures in general topologies. We make four main contributions with FELIX. As the **fifth contribution** of this thesis, we devise a packet processing pipeline with customized match-action

tables that forwards packets according to the current network state. The **sixth contribution** is the design of a lightweight protocol running on the fast path of switches to enable failure recovery coordination entirely in the data plane. The **seventh contribution** is the development of algorithms that compute and install alternate forwarding entries just in time to handle possibly-imminent failures. Through an extensive evaluation of FELIX, we find that it considerably reduces reaction times while making sensible use of data plane in-device memory. The **eighth**, and final, **contribution** is the proposal of the Strategy-Tactic paradigm. This paradigm abstracts away FELIX's elements to form an overall architecture that can be instantiated to perform other network operation tasks. We exemplify the generality of this paradigm by combining lessons learned from both FELIX and INTSIGHT to sketch INTREACT, a system for SLO- and contention-aware rerouting at data-plane timescales.

1.3 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we present the fundamental concepts related to this thesis, which revolves around network programmability, data plane programmability with the P4 language, and In-band Network Telemetry. In Chapter 3, we formally define the In-band Network Telemetry Orchestration (INTO) problem by introducing two mathematical programming models. We also introduce heuristic algorithms to both variations of the INTO problem to scale the ability of solving these problems to the order of hundreds of forwarding devices and thousands of packet flows. In Chapter 4, we introduce INTSIGHT, our SLO monitoring system that makes use of data plane programmability constructs to detect violations in the data plane and reduce analysis overheads. In Chapter 5, we present FELIX, our system for enabling networks to react and reroute around network failures in data plane time scales, substantially shortening downtime. In Chapter 6, we draw conclusions and directions for future research.

2 BACKGROUND

Despite the increased research interest on network programmability in the past few years, some of its concepts have been evolving over the last two or three decades. This chapter provides a background on past and present research efforts that explored and continue to explore network programmability[1]. In Section 2.1, we briefly revisit how the concept of programmable networks evolved since its inception until nowadays. In Section 2.2, we describe programmable data planes and the P4 language for programming packet forwarding devices. In Section 2.3, we introduce the concept of In-band Network Telemetry.

2.1 Network Programmability

The first research efforts in the direction of programmable networks emerged from discussions about the future directions of networking in the early to mid-1990s (FEAMSTER; REXFORD; ZEGURA, 2014). As the Internet began to support a wider range of applications, interconnected systems within it also expanded, both in number and size. The networking research community, led by the Defense Advanced Research Projects Agency (DARPA), identified several problems with network technologies at that time (CALVERT et al., 1998). Examples included difficulties in (*i*) integrating new technologies and standards into shared network infrastructures and (*ii*) introducing new protocols and services into existing architectural models.

Active networking was proposed as a solution to these issues (TENNENHOUSE et al., 1997). It envisioned that programming interfaces would provide abstractions of individual network node's resources (e.g., processing, packet queuing), just like programming languages expose commodity server resources. Programs would then be developed using these abstractions to define how data packets should be processed. There were two main approaches to realize that vision:

- Programmable switch model: switch programs that parse and process packets would

[1]This chapter is based on the following publication:

- Weverton Luis da Costa Cordeiro, Jonatas Adilson Marques, Luciano Paschoal Gaspary. **Data Plane Programmability Beyond OpenFlow: Opportunities and Challenges for Network and Service Operations and Management.** Journal of Network and Services Management (2017) (CORDEIRO; MARQUES; GASPARY, 2017).

be installed via out-of-band mechanisms (e.g., Mobile Agents (WHITE, 1993; GRAY, 1996), Script MIB (SCHOENWAELDER; QUITTEK, 2001)).

- Capsule-based system: flow forwarding is customized by higher level applications through capsules (special packets that direct themselves using custom forwarding routines), sent via programmable networking devices (active nodes) (WETHERALL; GUTTAG; TENNENHOUSE, 1998; WETHERALL, 2002).

This effort to enable network programmability proceeded until around the late 1990s to early 2000s, but did not gain widespread adoption (FEAMSTER; REXFORD; ZEGURA, 2014). That could be attributed mainly to (*i*) the lack of a clear path for deployment and (*ii*) the insufficient support for fully decoupling network intelligence from forwarding devices. Nonetheless, active networking offered some intellectual contributions and insights to network programming research. For example, projects like Alien (ALEXANDER et al., 1997), ANTS (WETHERALL; GUTTAG; TENNENHOUSE, 1998; WETHERALL, 2002), SwitchWare (ALEXANDER et al., 1998) and others (TENNENHOUSE et al., 1997; CALVERT et al., 1998; BRUNNER; STADLER, 1999; SMITH et al., 1999; QUITTEK; BRUNNER, 2003) were the first to put forward the notion that the ability to program networks is key to allowing networking innovation. Additionally, active networking research uncovered the challenges (e.g., providing security) and downsides (e.g., potential processing overload) involved in executing arbitrary code at packet forwarding nodes inside the network.

In the early to mid-2000s, as networks continued to grow in size and traffic volume, control and management challenges became more complex and demanding. To further complicate matters, existing infrastructures had a large base of deployed devices and protocols (often legacy ones). Such scenario hampered adoption of innovative approaches, because of the reluctance to experiment them with production traffic and at scale. With this problem in mind, the research community started a long-term effort to build large scale facilities (like GENI and FIRE) for experimenting novel approaches and architectures (FEAMSTER; REXFORD; ZEGURA, 2014).

In parallel, some investigations (YANG et al., 2004; FEAMSTER et al., 2004; LAKSHMAN et al., 2004) evidenced the limitations (related, for example, with scalability and reliability) of mechanisms and protocols employed to assist with control plane tasks. They argued that these limitations where inherent to the distributed way (e.g., decentralized decision process, information flooding) in which those mechanisms and protocols operated. Given this perspective, two complementary lines of research where

explored: (*i*) standardization of interfaces between the control and forwarding planes – for example, Linux Netlink (SALIM et al., 2003), Forwarding and Control Element Separation (ForCES) (YANG et al., 2004) – and (*ii*) logical centralization of network control – for example, Routing Control Platform (RCP) (FEAMSTER et al., 2004) and SoftRouter (LAKSHMAN et al., 2004). An open and standardized interface between the components that implement the functionality of those planes would eliminate the tight coupling of their designs, enabling a higher degree of innovation in both planes. In its turn, a logically centralized control would provide controller applications with a complete network view, enabling the simplification and improvement of their decision process (e.g., choosing a route for a packet).

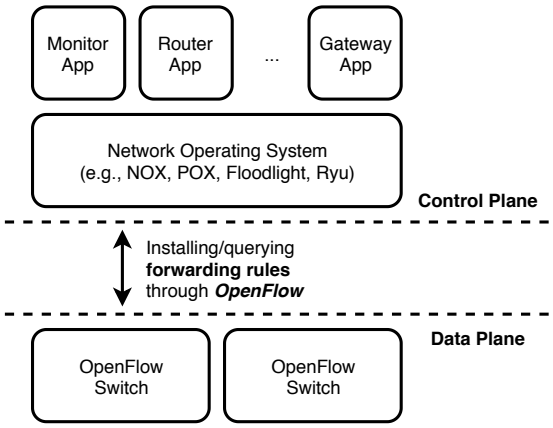
As it was the case with active networking, the adoption of the proposals above remained small. Vendors had no incentive to support interfaces like ForCES (YANG et al., 2004), as it would only lower the entry barrier for new competitors, without providing clear product advantages (FEAMSTER; REXFORD; ZEGURA, 2014). Other initiatives had limited scope, thus becoming less interesting. One example is RCP (FEAMSTER et al., 2004), which reused BGP (REKHTER; HARES; LI, 2006) functionality to facilitate deployment on existing devices, but was limited to routing. Nevertheless, some research principles and concepts on control and data plane separation (e.g., mechanisms for distributed state management) remained as key contributions and were later revisited.

By 2005, Greenberg *et al.* proposed 4D (GREENBERG et al., 2005), a more conceptual research effort towards separating the control and data planes. It focused on the realization of network control and management tasks (other than routing) based on a clean slate architecture, i.e., without requiring any architectural models or interoperability with existing protocols or services. The 4D architecture was followed by projects Tesseract (YAN et al., 2007), SANE (CASADO et al., 2006), and Ethane (CASADO et al., 2007), and had direct influence on some important subsequent work on network programmability like OpenFlow (MCKEOWN et al., 2008) and NOX (GUDE et al., 2008).

Later in 2008, while 4D was still under investigation, McKeown *et al.* (MCKEOWN et al., 2008) established the Software-Defined Networking (SDN) paradigm by introducing OpenFlow (MCKEOWN et al., 2008). OpenFlow is a protocol whose initial goal was enabling almost immediate research in a more specific type of infrastructure, namely campus networks. Figure 2.1 illustrates the concept. In McKeown *et al.*'s SDN paradigm, control plane applications (which run on top of network OSes such as NOX (GUDE et al., 2008)) implement functions like IP routing, monitoring, *etc*. These apps

populate packet forwarding rules on switches, using OpenFlow. The set of rules an app can configure into switches has to observe both the OpenFlow version in use, and the set of TCP/IP protocols supported by switches. OpenFlow is highly pragmatic in design, which means that switch vendors can support it (*i*) without exposing internal working of their switches, and (*ii*) in coexistence with their proprietary management interfaces. As a result, SDN (and OpenFlow) became the first network programmability approach to reach widespread adoption. Its success led to the formation of the Open Networking Foundation[2] (ONF), an organization dedicated to fostering open networking standards that has substantially grown in the last decade and hosts projects across all areas of networking. In the next section, we will describe programmable data planes and P4, one of the most recent and groundbreaking projects hosted and supported by ONF.

Figure 2.1: Traditional SDN architecture.



Source: Adapted from (BOSSHART et al., 2014).

2.2 Programmable Data Planes and P4

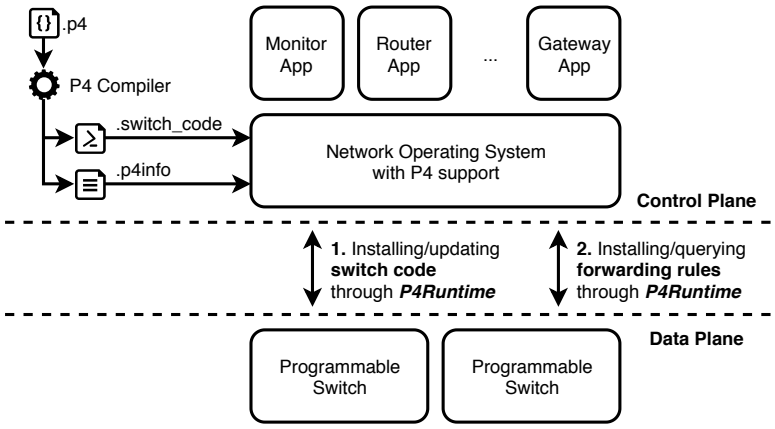
Taking a step back from the timeline presented in the previous section, with SDN, a notion that became popular was "programming" the data plane (FOSTER et al., 2010; TOOTOONCHIAN; GANJALI, 2010; MACEDO et al., 2015; DABBAGH et al., 2015), that is, populating switch tables with rules on how to forward flows in the data plane. Although not incorrect, this simplistic, yet pragmatic approach ended up revealing a more complex problem: that networking evolution is also hampered by inability of operators to properly *program* switch behavior. In other words, operators are constrained by the set of features and protocols supported by fixed-function ASIC switches, and cannot design

[2] www.opennetworking.org

custom packet headers and packet parsing routines. Another related issue is that supporting novel Layers 1, 2, and 3 protocols requires a complex and time-consuming pipeline, which includes protocol standardization (by bodies like IETF) with support from some switch vendor, and the release of an OpenFlow version that supports the protocol header fields. This process can take from months to years, thus considerably hindering the proposal and adoption of innovative protocols.

In the mid-2010s, one concept that emerged to solve the problem above is data plane programmability. It reshapes the SDN landscape by enabling network operators to reprogram forwarding devices in-field to deploy novel networking protocols, customize network behavior, and consequently develop innovative services (e.g., (KIM et al., 2015; DANG et al., 2016)). Figure 2.2 illustrates the concept. In a programmable data plane, operators can write switch code that specifies custom header fields and define packet header matching and parsing semantics. By using a vendor supplied compiler, operators can compile the code and deploy the resulting program into the switch, thus truly reshaping what protocols it supports and how it behaves. Then P4Runtime, an open interface similar in purpose to OpenFlow, can be used to populate forwarding rules built upon customized, programmed switch features. Data plane programmability is quickly gaining attention and momentum, as it represents an important step beyond SDN.

Figure 2.2: SDN architecture extended with programmable data planes and P4.



Source: Adapted from (BOSSHART et al., 2014).

In a programmable data plane-enabled environment, flow rule specification is no longer determined/restricted by the switch/OpenFlow specification. Instead, operators can use domain-specific languages (DSLs) to effectively program network behavior, from control plane applications to forwarding devices. P4 (BOSSHART et al., 2014) is the de-facto, high-level DSL to program packet processing by individual forwarding devices. P4 is target independent, i.e., suitable for describing the behavior of various types of devices,

going from fixed-function ASICs, passing through NPUs and FPGAs, and all the way to software switches. The language abstracts packet parsing and processing, by providing a generalized *forwarding model*. P4 code is logically organized into (a) data declaration, (b) parser logic, and (c) match+action tables and control flow sections. Each of them is mapped to specific elements of the forwarding model (see Figure 2.3).

Figure 2.3: P4 code sections, and mapping to the abstract forwarding model (Adapted from Kim and Lee (KIM; LEE, 2016)).

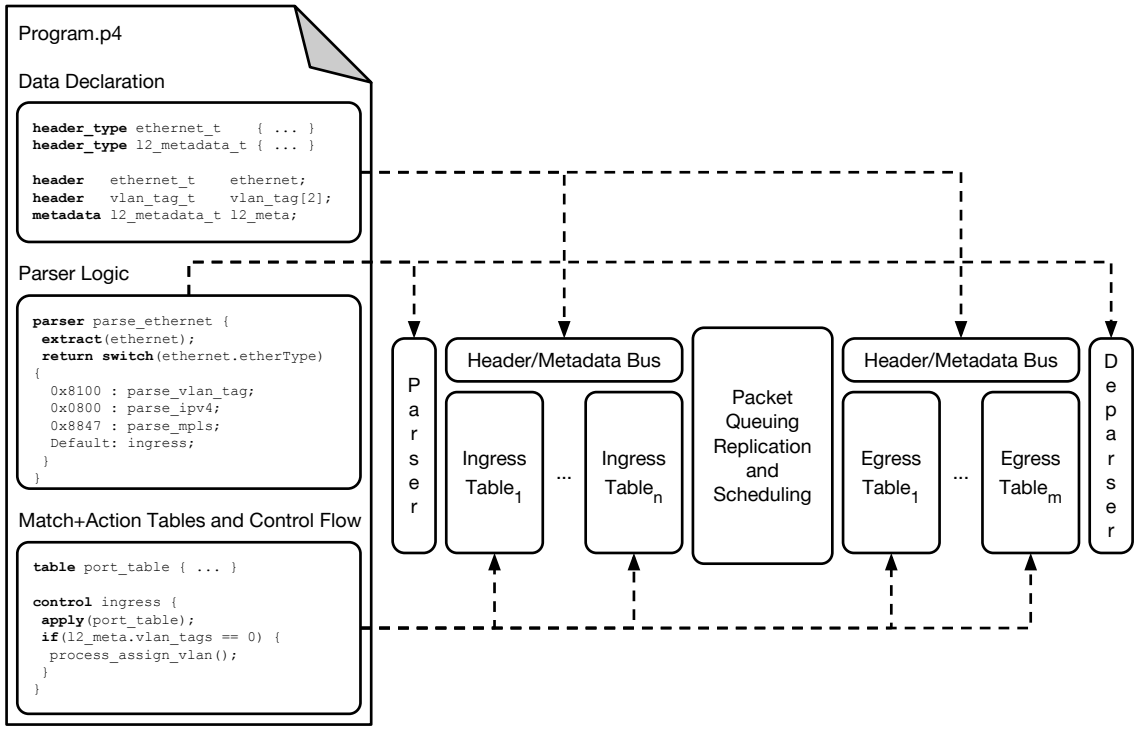


Figure 2.4: P4 header declaration examples.

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

(a) Ethernet header.

```
header custom_protocol_h {
  bit<16> src_tag;
  bit<16> dst_tag;
  bit<16> next_protocol;
}
```

(b) Custom protocol header.

The *data declaration* section defines packet header format and meta-data information that can be used for its parsing. This section is mapped into a header and meta-data bus that carries this information through all the processing stages. Header types are declared similarly to structs in C, i.e., fields are defined in a specific order and with a predetermined length. Figures 2.4a and 2.4b illustrate the declaration of the standard Ethernet header and of an arbitrary custom protocol that uses tags as source and destination identifiers, respectively.

The *parser logic* section specifies how, when, and in what order each of the headers are to be parsed. This section of a P4 program is mapped to the parser and deparser elements of the forwarding model (see Figure 2.3). These elements are then responsible for extracting the header field from packets on their ingress (parser) and serializing their (possibly) updated values back to the packets on their egress (deparser).

The last program section, *match+action tables and control flow*, specifies lookup tables capable of matching on arbitrary header fields and modifying packet headers (and meta-data) through custom defined actions. It also expresses control functions that define in which order and circumstances each table should be executed. This section is mapped to configurable match+action capable elements in both ingress and egress pipelines. The ingress pipeline carries out (egress-agnostic) packet modification and also stipulates egress intentions (e.g., which port to forward a packet). The egress pipeline realizes further necessary packet modifications, e.g., rewriting an Ethernet header source address field with the egress port MAC address of the switch.

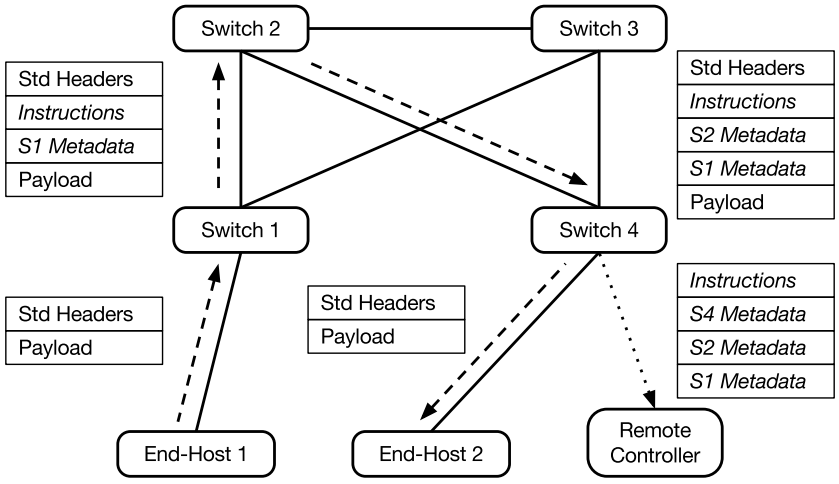
2.3 In-Band Network Telemetry

Data plane programmability makes feasible a novel method of collecting and transmitting network measurements, called *In-band Network Telemetry* (INT) or *in-situ* OAM (BROCKNERS et al., 2017). This method consists of recording operations, administration, and maintenance information within a data packet as it traverses a network. Several frameworks and techniques have been proposed to realize in-band network telemetry, e.g., TPP (JEYAKUMAR et al., 2014) and INT (KIM et al., 2015), and Cisco iOAM (CISCO, S.a.). These put forward the idea of allowing data packets to query instantaneous switch-internal state metrics – such as queue sizes, queuing latency, and link utilization – and to store this information on telemetry purposed headers.

Figure 2.5 depicts the execution flow of INT (KIM et al., 2015), the currently most prominent in-band telemetry framework that can be readily implemented using P4. The INT architecture abstraction is composed of a remote monitoring controller and of source, transit, and sink nodes, each of which represents a role in its instantiation. Each programmable switch within the path of a packet (as it is transmitted through the network) may assume one or more roles. The figure illustrates a scenario where *End-Host 1* sends a traditional data packet to *End-Host 2* through a network of INT-capable switches.

Source nodes (in our example, *Switch 1*) are responsible for embedding measure-

Figure 2.5: Architecture of the INT framework for in-band network telemetry (Adapted from Thomas and Laupkhov (THOMAS; LAUPKHOV, 2016)).



ment instructions (typically in the form of header values) into regular or probing packets. *Transit nodes* execute the instructions and append measured values into the packets. In our example, *Switches 1*, *2*, and *4* assume the role of transit nodes, since the packet path is *End-Host 1* $\rightarrow$ *Switch 1* $\rightarrow$ *Switch 2* $\rightarrow$ *Switch 4* $\rightarrow$ *End-Host 2*. Lastly, *sink nodes* (in our example, *Switch 4*) retrieve the results of the instructions and report (appropriate subsets of) them to a controller. Examples of metadata that can be collected at each switch are the switch ID, the ingress/egress port ID, timestamp, byte count, drop count, and link utilization, as well as a queue ID, occupancy, congestion status, and average length. In-band Network Telemetry provides a way for monitoring networks and services with unprecedented levels of accuracy and detail (JEYAKUMAR et al., 2014; KIM et al., 2015).

3 ORCHESTRATING IN-BAND NETWORK TELEMETRY

Considering Question 1 introduced in Section 1.2 – i.e., *how to orchestrate in-band network telemetry collection efficiently and effectivelly?* – in this chapter, we formalize the challenge of orchestrating In-band Network Telemetry actions as an optimization problem[1]. We start, in Section 3.1, by further contextualizing and motivating this work. In Section 3.2, we introduce the INTO problem context and variables along with its general optimization goal. In Section 3.3, we present two variations of the INTO problem, propose mathematical programming models to solve them and prove that both are NP-Complete problems. In Section 3.4, we propose heuristic algorithms to produce high-quality solutions for realistic network topologies in a timely fashion. In Section 3.5, we evaluate and compare the proposed optimization models and heuristic algorithms. Finally, in Section 3.7, we summarize our findings and present our final remarks on the INTO problem.

3.1 Motivation

As previously discussed in Section 2.3, In-band Network Telemetry (INT) enables monitoring networks and their services with high accuracy and level of detail. Despite these benefits in achievable quality, because it uses production traffic, it is crucial to consider constraints imposed by the traffic and potential impacting factors to perform INT effectively and efficiently. The primary constraint of INT is that packets cannot exceed the network MTU. Therefore, the length of the telemetry data that can be embedded in a packet is limited by the difference between its original size and the MTU. The smaller the packet, the larger its telemetry capacity (i.e., the number of metadata items it can transport). Any proposed orchestration strategy has to comply with this constraint, which may limit the number of interfaces that can be monitored in a network. Next, we present and discuss the main factors related to INT that may lead to network performance degradation.

[1]This chapter is based on the following publication:

- Jonatas Adilson Marques, Luciano Paschoal Gaspary. **Explorando Estratégias de Orquestração de Telemetria em Planos de Dados Programáveis.** XXXVI Brazilian Simposium on Computer Networks and Distributed Systems (SBRC 2018) (MARQUES; GASPARY, 2018).
- Jonatas Adilson Marques, Marcelo Caggiani Luizelli, Roberto Irajá Tavares da Costa Filho, Luciano Paschoal Gaspary. **An Optimization-based Approach for Efficient Network Monitoring using In-band Network Telemetry** Journal of Internet Services and Applications (2019) (MARQUES et al., 2019).

(a) **Embedding telemetry data into packets causes their size to increase along their paths.** Making packets increase in size may cause jitter in their transmission. Jitter may degrade the QoS of many applications, especially of non-elastic ones (e.g., VoIP, virtual reality video streaming).

(b) **Packet forwarding devices have limited processing capacity.** The generation of telemetry report packets uses this capacity, thus, forwarding too many reports may saturate devices.

(c) **Monitoring sinks and analyzers have limited processing capacity.** Receiving too many report packets and too many telemetry data may saturate these machines, which could impair their capacity to monitor the network correctly.

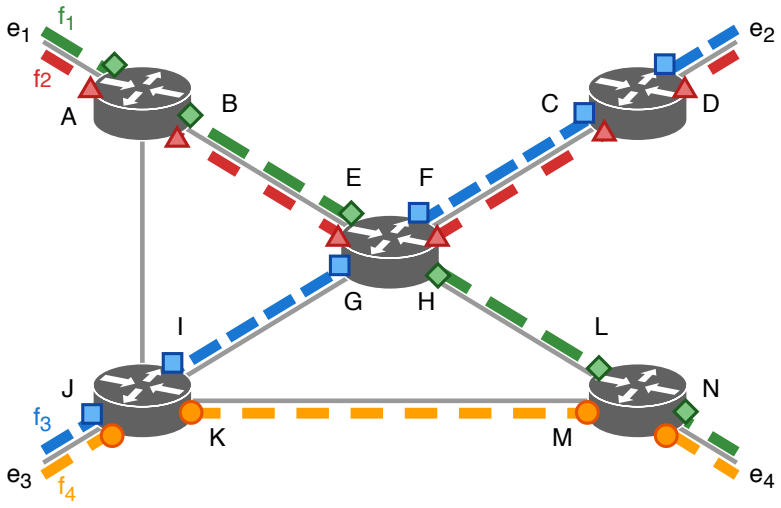
(d) **Network links have limited bandwidth.** The telemetry data transported in production packets uses the bandwidth of links in the network. If too much data is inserted into packets and reported to sinks, the growth in data volume may saturate links and devices.

The level of impact of the factors mentioned above is intrinsically related to the assignment of INT tasks to the traffic in the network. Figure 3.1 illustrates the "full" assignment of telemetry tasks, which is the straightforward method to carry out INT proposed in its original designs (JEYAKUMAR et al., 2014). The network in Figure 3.1 is composed of five forwarding devices and has endpoints to four other networks ($e_1 - e_4$). Moreover, there are four packet flows of the same traffic type: $f_1 : e_1 \leftrightarrow e_4$, $f_2 : e_1 \leftrightarrow e_2$, $f_3 : e_2 \leftrightarrow e_3$, and $f_4 : e_3 \leftrightarrow e_4$. The full assignment represents a scenario where every flow in the network would collect (and transport) metadata items from all device interfaces in its path. For example, the packets of flow f_4 collect information about interfaces J, K, M, and N, as it is indicated by the orange circles in the figure.

The full assignment has significant drawbacks. First, it is not aware of telemetry demands and capacities. For example, consider the case where interfaces J, K, M, and N had each four telemetry items to be collected, and the telemetry capacity of flow f_4 was 12 items (according to the typical size of packets). In this case, the full assignment would be unfeasible, because by the time packets coming from e_3 arrived at interface N they would not have enough space to collect the four items from it. Therefore, the full assignment does not guarantee that, in practice, flows will cover all interfaces in their paths.

Second, all flows are subject to the performance degradation factors discussed previously in this section, as the full assignment is not selective in its choices. Third, all telemetry supporting tasks (i.e., telemetry header creation and extraction, report packet

Figure 3.1: Example of full assignment.



generation and transmission) tend to be executed by edge devices, increasing their probability of being saturated. Fourth, device interfaces are often monitored by multiple flows, each of them collecting the instantaneous value of the same metadata items. Previous work (KIM et al., 2015; ZHANG et al., 2017) has shown that it is possible to obtain instantaneous metadata with microsecond granularity using only one out of all flows traversing an interface or forwarding device. Furthermore, since all flows in our example scenario are of the same traffic type, behavioral and performance metadata (e.g., forwarding rules, queue delay) is expected to be similar. Finally, in this assignment, the INT overhead is highly influenced by the level of activity in the network (i.e., the number of flows). Thus, an increase in network activity may inadvertently saturate its links and devices.

In summary, we advocate that network monitoring through INT requires some sort of task orchestration to be viable in practice. In the next sections, we present our proposed solution, starting, next, with the formalization of INT orchestration as an optimization problem.

3.2 INTO

In general terms, the problem under study – entitled In-Band Network Telemetry Orchestration (INTO) problem – consists in monitoring network device interfaces effectively (covering all monitoring demands) and efficiently (minimizing resource consumption and processing overheads). We start the problem definition by formally describing the input and output of our optimization models. For convenience, Table 3.1 presents the

complete notation used in the formulation.

Table 3.1: Summary of symbols.

Symbol	Definition
$G = (D, I)$	Physical infrastructure G.
D	Set of programmable forwarding devices.
I	Set of device interfaces.
$\delta(i)$	Interface i telemetry demand. The number of telemetry items to be collected from interface i.
F	Set of active flows in G.
$\rho(f)$	Set of interfaces through which packets from flow f are forwarded.
$\kappa(f)$	Flow f telemetry capacity. The maximum number of items packets from flow f may transport.
$\Phi : I \rightarrow F$	An assignment function of device interfaces I to network flows F.
$x_{i,f}$	Binary variable which indicates whether flow f is assigned to cover interface $i \in I$.
y_f	Binary variable of the INTO Concentrate optimization model which indicates whether flow f is a telemetry-active flow.

The INTO problem considers a physical network infrastructure $G = (D, I)$ and a set of network aggregate flows F. For this formalization, we assume that the set of active aggregate flows is mostly stable in the network. Set D in network G represents the programmable forwarding devices $D = \{1, 2, ..., |D|\}$. Each device $d \in D$ has a set of network interfaces that are connected to other devices in the network. We denote the interface of a device d_a that is connected to a device d_b by the tuple (d_a, d_b). Similarly, the interface of d_b that is connected to d_a is denoted by (d_b, d_a). The set of all device interfaces in the network is denoted by I. For each interface $i \in I$, there is an associated monitoring demand, a fixed number of telemetry items $\delta(i) \in \mathbb{N}^+$ that need to be collected periodically by flows in F. The interface telemetry demands are determined by monitoring policies, which are influenced by, for example, the level of activity of each interface or a previously detected event of interest. The telemetry items that can be collected include any information that can be made readily available for reading on a P4 program. These include (but are not limited to) the metadata defined in the specification for INT (GROUP, 2018) (e.g., node and interface IDs, ingress and egress timestamps,

hop latency, link utilization, queue occupancy), flow and packet counters, and stateful registers. We note that stateful registers enable implementing streaming and sketching algorithms, which can be designed for many different purposes such as calculating entropy of frequencies (LAPOLLI; MARQUES; GASPARY, 2019) and finding the most frequent items (SIVARAMAN et al., 2017).

The set F represents a group of *aggregate packet flows* of the same traffic type that are active in the network. In the case where the operator wants to monitor different types of traffic (e.g., scientific computing, video streaming, VoIP), a separate problem instance could be created for each one with their respective flows. This is done because the metadata values observed by a flow may be different from the values observed by other flows, specially for performance-related metadata (LEE; DUFFIELD; KOMPELLA, 2010). For example, if a network prioritizes forwarding VoIP traffic in detriment of Web traffic, the queueing time observed by packets of each type may be different. Thus, if an operator wants to obtain metadata that is highly consistent with a specific type of traffic, the best approach to guarantee this would be to create a problem instance for each type of traffic. We note that the INTO problem definition and our solutions do not preclude operators from treating all traffic as a single type. Creating separate instances is therefore a recommendation for the obtention of more consistent monitoring data.

Each flow $f \in F$ has two endpoints (ingress and egress) and is routed within the network infrastructure G using a single path. We denote the path $\rho(f)$ of a specific flow f as a list of interfaces through which its packets are forwarded. For example, a network flow f from endpoint s to t routed through forwarding devices 1, 3, and 4 has $\rho(f) = (1, s), (1, 3), (3, 1), (3, 4), (4, 3), (4, t)$. The first forwarding device interface visited is that of device 1 connected to the ingress endpoint (s), and the last is that of device 4 connected to the egress endpoint (t). Associated with each flow f is also a telemetry capacity $\kappa(f) \in \mathbb{N}^+$, which is the maximum number of items each packet of the flow may transport. The capacity of the flows is determined by factors such as forwarding protocols (e.g., IPv4, IPv6, NSH (QUINN; ELZUR; PIGNATARO, 2018)), packet sizes, and network monitoring policies. The telemetry capacity may differ from packet to packet in a single flow, but we expect that the distribution of sizes is stable to be estimated with historic data. The capacity of (aggregate) flows can be defined according to percentile values of the distribution of sizes in order to guarantee that most (e.g., 90%) of the packets will have enough space to collect the metadata of the interfaces they are assigned to cover.

Given the problem input, an INTO optimization model will try to find a feasible assignment $\Phi : I \rightarrow F$ that optimizes a specific objective function, where $\Phi(i) = f$ indicates that flow $f \in F$ should cover interface $i \in I$. A feasible assignment is one where (*i*) each interface is covered by exactly one of all flows in F that pass through it and (*ii*) no flow $f \in F$ is assigned to cover more interfaces than its capacity allows, i.e., the sum of demands from all interfaces covered by a flow does not exceed its capacity.

We highlight two important design decisions in our models. First, assignment function Φ does not enable partitioning the demand of an interface across multiple flows. We chose this type of assignment because some of the items to be collected on an interface are interdependent. For example, when collecting the transmission utilization it is also necessary to collect the ID of the respective interface. Enabling items to be balanced in different flows would require adjusting the models to assign flows to cover individual items instead of interfaces and introducing additional restrictions to force certain assignments in order to satisfy interdependency requirements. At this point, it is not clear that enabling such granularity in assignment would bring advantages enough to counter the complexity that would be introduced to the models, but we do consider further investigating this possibility in a future work.

Second, our model assigns a *single* aggregate flow per interface. We observe that, in some cases, it may be useful to introduce limited assignment redundancy. For example, to guarantee coverage and reduce the number of necessary configuration updates in a scenario of highly frequent changes in the set of flows. In our work, this redundancy is implicitly achieved through the employment of the "grain" of *aggregate flows* traversing a network core, which tend to present small and few changes over time. We leave exploring alternative options for both of these design decisions for future work.

3.3 Problem Variations

In view of the key traffic constraint and performance influencing factors discussed in Section 2.3, we define two optimization problems, namely, INTO Concentrate and INTO Balance. Next, we present these problems by describing their objective functions, proposing optimization models to solve them and proving that they are NP-Complete problems.

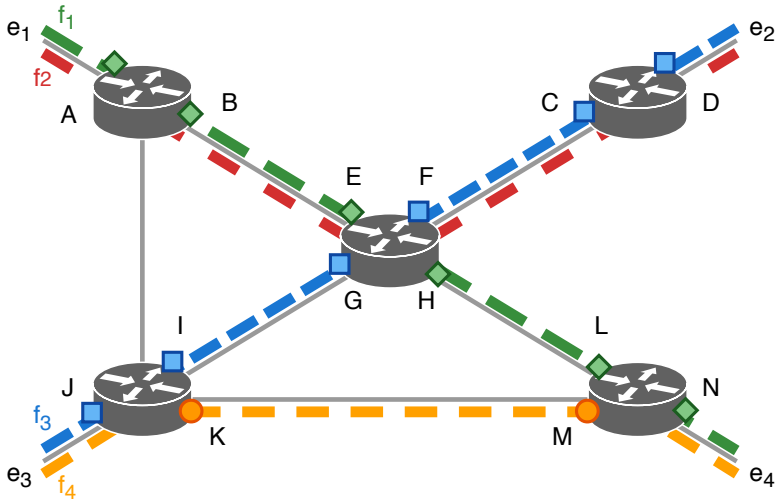
3.3.1 INTO Concentrate

The number of flows participating in monitoring the network via INT dictates the number of telemetry report packets generated periodically since a report is sent for each packet of each telemetry-active flow. As previously discussed, creating too many report packets may saturate the forwarding devices that are tasked with their generation and the machines tasked with their analysis. Therefore, one possible optimization goal is minimizing the number of telemetry-active flows. That is the objective of the INTO Concentrate problem.

Example Assignment

Figure 3.2 shows an example "Concentrated assignment", i.e., an assignment that uses the optimal number of telemetry-active flows for our running example previously presented in Figure 3.1. The example assignment shows that it is possible to cover all device interfaces using only three out of the four flows in the network. Flows f_1 and f_3 are assigned to cover six interfaces each, {A, B, E, H, L, N} and {J, I, G, F, C, D}. Flow f_4 is assigned to monitor the remaining two uncovered interfaces K and M, while flow f_2 is not assigned to cover any interface.

Figure 3.2: Concentrated assignment example.



Optimization Model

The proposed optimization model to solve INTO Concentrate is presented next as an integer linear program. Variable $x_{i,f}$ indicates whether flow f should cover interface

$i \in I$. The values of all variables $x_{i,f}$ define the assignment function Φ, i.e., $x_{i,f} = 1 \rightarrow \Phi(i) = f$. Variable y_f indicates whether flow f is a telemetry-active flow (i.e., if it covers at least one interface). This second set of variables is necessary to compute the objective function of the problem, the number of telemetry-active flows (Equation 3.1).

$$\min \Phi_c = \sum_{f \in F} y_f \tag{3.1}$$

$$\text{s.t.} \sum_{f \in F} x_{i,f} = 1, \qquad \forall i \in I, \delta(i) > 0 \tag{3.2}$$

$$\sum_{i \in \rho(f)} x_{i,f} \cdot \delta(i) \leq y_f \cdot \kappa(f), \qquad \forall f \in F \tag{3.3}$$

$$x_{i,f} \in \{0, 1\}, \qquad \forall i \in I, \forall f \in F \tag{3.4}$$

$$y_f \in \{0, 1\}, \qquad \forall f \in F \tag{3.5}$$

The objective function (Equation 3.1) defines the minimization of the number of telemetry-active flows by the sum of the y variables. Constraint set in Equation 3.2 ensures that all interfaces $i \in I$ with positive demand are covered by some flow $f \in F$. Constraint set in Equation 3.3 bounds the number of telemetry items each flow $f \in F$ can be assigned to collect and transport according to its capacity $\kappa(f)$. Equation 3.3 also activates y_f when any telemetry item is assigned to flow $f \in F$. Constraint sets in Equations 3.4 and 3.5 define the domains of variables $x_{i,f}$ and y_f, which are binary.

Proof of NP-Completeness

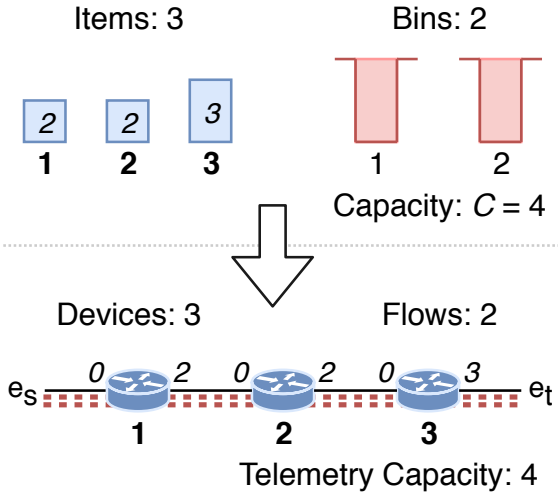
We will now prove that the decision version of the INTO Concentrate problem is an NP-Complete problem. Given a network infrastructure $G = (D, I)$, the set of network flows F, and an integer number n; the goal of the decision version of the problem is to determine whether there exists a feasible assignment Φ where no more than n flows are used to transport telemetry items.

Lemma 1. *INTO Concentrate belongs to the class NP.* We prove that the decision version of the INTO Concentrate problem is in NP by way of a verifier. Given any assignment $\Phi : I \rightarrow F$, the verifier needs to check three conditions. First, that the number of telemetry-active flows is indeed at most n (in $\mathcal{O}(|F|)$ time). Second, that every interface is covered (in $\mathcal{O}(|I|)$ time). Third, that no flow collects more telemetry items than its capacity (in $\mathcal{O}(|I||F|)$). For a *yes* instance (G, F, n), the certificate is any feasible assign-

ment using n flows in F; the verifier will accept such an assignment. For a *no* instance (G, F, n), it is clear that no assignment using n flows of F will be accepted by the verifier as a feasible assignment.

Lemma 2. *Any Bin Packing problem instance can be reduced in polynomial time to an instance of the INTO Concentrate Decision problem.* An instance of the Bin Packing Problem (BPP), which is a classical NP-Complete problem (GAREY; JOHNSON, 1979), comprises a set A of items, a size s_a for each item $a \in A$, a positive integer bin capacity C, and a positive integer n. The decision version of the BPP asks if its possible to pack all items into n bins, i.e., if there is a partition of A into n disjoint sets $(B_1, B_2, ..., B_n)$ such that the sum of sizes of the items in each subset is at most C. The INTO Concentrate problem is a generalization of the BPP where bins may have different capacities and each item may only be put on a specific subset of all bins. We reduce any instance of the BPP to an instance of the INTO Concentrate problem using the following procedure. The reduction has polynomial time complexity $\mathcal{O}(n|A|)$.

Figure 3.3: Example BPP reduction to INTO Concentrate. A BPP instance with $|A| = 3$ items, bin capacity $C = 4$, and number of bins $n = 2$ is reduced to an INTO Concentrate instance with 3 forwarding devices and 2 flows with telemetry capacity equal to 4 items.



1. An infrastructure G is created with $|A|$ forwarding devices and two endpoints e_s and e_t. See example in Figure 3.3, device numbers are shown in bold. The devices are connected in line, i.e., device $a \in A$ is connected to device $b = a + 1, b \in A$. Devices $a = 1$ and $a = |A|$ are also connected to endpoints e_s and e_t, respectively. The interface of each device $a = 1, 2, ..., |A|$ that is connected to the next device (or to endpoint e_t in the case of device $a = |A|$) has telemetry demand equal to s_a, while the other interface has no telemetry demand. The interface demands are shown in italics in the figure.

2. Next, n flows ($B_1, B_2, ..., B_n$) are created. Each flow has endpoints e_s and e_t and is routed through a path comprising all forwarding device interfaces in G, i.e., $\rho(B_i) = (1, e_s), (1, 2), (2, 1), ..., (|A|, e_t)$, $i = 1, 2, ..., n$. The telemetry capacity of all flows is C.

Theorem 1. *The INTO Concentrate decision problem is NP-Complete.*

Proof. By the instance reduction presented, if a BPP instance has a solution using n bins, then the INTO Concentrate problem has a solution using n flows. Consider that each item a of size s_a packed into a bin B_i corresponds to the coverage of the interface of device a with positive telemetry demand by flow B_i. Conversely, if the INTO Concentrate problem has a solution using n flows, then the corresponding BPP instance has a solution using n bins. Covering the positive demand interface of device a with flow B_i corresponds to packing an item of size s_a into bin B_i. Lemmas 1 and 2 complete the proof. □

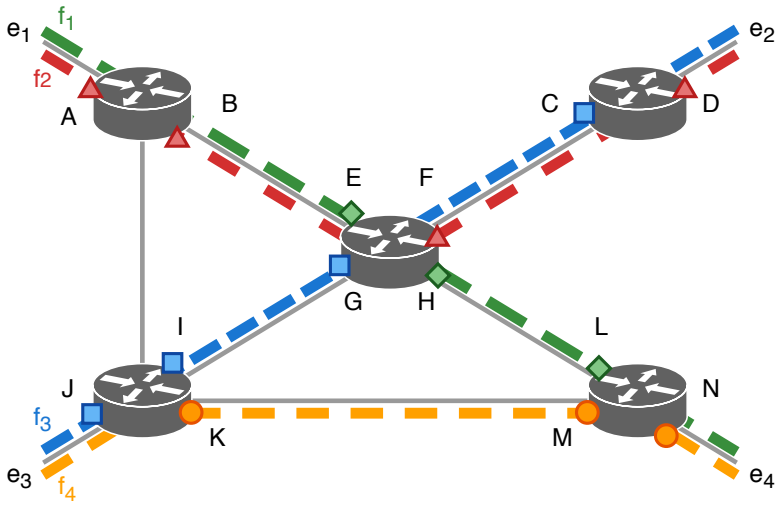
3.3.2 INTO Balance

The number of telemetry items to be collected by each flow determines how much a packet will grow in size during its forwarding inside the network. If too many telemetry items are concentrated into one flow, the links through which its packets are forwarded may be saturated by significant growth in data volume. Consequently, another optimization goal would be to balance the telemetry demands among as many flows as possible. This assignment strategy would minimize the probability of saturating any single link because the variation in data volume becomes minimal for each flow. The INTO Balance objective is to minimize the maximum number of telemetry items transported (i.e., telemetry load) by any single flow.

Example Assignment

Figure 3.4 illustrates a possible Balanced assignment, i.e., one that achieves the optimal telemetry load balance, for the example scenario previously presented in Figures 3.1 and 3.2. This example assignment shows that it is possible to cover all device interfaces in the network while assigning no more than four interfaces (i.e., collecting $4 \times 4 = 16$ items) per telemetry-active flow. The optimal load balance value is determined by three factors: (*i*) the maximum telemetry demand per single interface, (*ii*) the fraction between the sum of all interface demands and the number of flows, and (*iii*) the

Figure 3.4: Balanced assignment example.



different flow to interface assignment options. In the example, all interfaces have equal demand of four items. The total sum of demands 14 interfaces $\times$ 4 items $=$ 56 items divided by the four flows equals 14 items per flow. Since a flow must collect all (or none) of the items of an interface, the optimal load balance is found when two flows are assigned to collect 16 items each (i.e., cover four interfaces) and the other two are assigned to collect 12 items (i.e., three interfaces). The assignment shown in Figure 3.4 follows this distribution. Flows f_2 and f_3 are assigned to four interfaces each, {A, B, D, F} and {C, G, I, J}. Likewise, f_1 and f_4 are assigned to three interfaces each, {E, H, L} and {K, M, N}.

The objective function (Equation 3.6) defines the minimization of k, the maximum telemetry load of each flow. Constraint set in Equation 3.7 ensures that all interfaces $i \in I$ (with positive demand) are covered by some flow $f \in F$. Constraint set in Equation 3.8 bounds the number of telemetry items each flow $f \in F$ can be assigned to collect and transport according to its capacity $\kappa(f)$. Constraint set in Equation 3.9 guarantees that k is at least the maximum number of items to be collected by any single flow. Constraint set in Equation 3.10 defines the domains of variables $x_{i,f}$, which are binary. The constraint in Equation 3.11 defines the domain of variable k.

Optimization Model

Next, we present the integer linear program to solve the INTO Balance optimization problem. Variable $x_{i,f}$ indicates whether flow $f \in F$ should cover interface $i \in I$. As was the case with the Concentrate model, the values of variables $x_{i,f}$ define the assignment function Φ, i.e., $x_{i,f} = 1 \rightarrow \Phi(i) = f$. Variable k indicates the maximum number

of telemetry items assigned to be collected and transported by a single flow.

$$\min \Phi_b = k \tag{3.6}$$

$$\text{s.t.} \sum_{f \in F} x_{i,f} = 1, \qquad \forall i \in I, \delta(i) > 0 \tag{3.7}$$

$$\sum_{i \in \rho(f)} x_{i,f} \cdot \delta(i) \leq \kappa(f), \qquad \forall f \in F \tag{3.8}$$

$$\sum_{i \in \rho(f)} x_{i,f} \cdot \delta(i) \leq k, \qquad \forall f \in F \tag{3.9}$$

$$x_{i,f} \in \{0, 1\}, \qquad \forall i \in I, \forall f \in F \tag{3.10}$$

$$k \geq 0 \tag{3.11}$$

Proof of NP-Completeness

We now prove that the decision version of the INTO Balance problem is an NP-Complete problem. Given a network infrastructure $G = (D, I)$, the set of network flows F, and an integer number n; the goal of the decision version of the problem is to determine if there exists a feasible assignment Φ where no flow is assigned to transport more than n telemetry items.

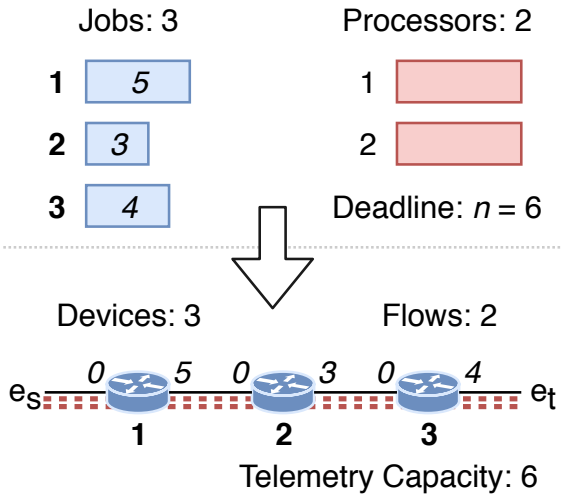
Lemma 3. *INTO Balance belongs to the class NP.* Similarly to Lemma 1, we prove that the decision version of the INTO Balance problem is in NP by way of a verifier. Given any assignment $\Phi : I \rightarrow F$, the verifier needs to check two conditions. First, that no flow collects more than n telemetry items and than its capacity (in $\mathcal{O}(|I||F|)$). Second, that every interface is covered (in $\mathcal{O}(|I|)$ time). For a *yes* instance (G, F, n), the certificate is any feasible assignment where each telemetry-active flow transports at most n items; the verifier will accept such an assignment. For a *no* instance (G, F, n), it is clear that no assignment making each telemetry-active flow collect at most k items will be accepted by the verifier as a feasible assignment.

Lemma 4. *Any Multiprocessor Scheduling problem instance can be reduced in polynomial time to an instance of the INTO Balance Decision problem.* An instance of the Multiprocessor Scheduling Problem (MSP), which is a known NP-Complete problem (GAREY; JOHNSON, 1979), comprises a set J of jobs, a length l_j for each job $j \in J$, a set of m processors, and a deadline n (a positive integer). The goal of the decision version of the MSP is to decide whether there exists a scheduling of jobs on the m processors

$(P_1, P_2, ..., P_m)$ such that all jobs finish before elapsed time n. We reduce any instance of the MSP to an instance of the INTO Balance problem using the following procedure. The reduction has polynomial time complexity $\mathcal{O}(m|J|)$.

1. An infrastructure G is created with $|J|$ forwarding devices and two endpoints e_s and e_t. See example in Figure 3.5, device numbers are shown in bold. The devices are connected in line, i.e., device $j \in J$ is connected to device $k = j+1, k \in J$. Devices 1 and $|J|$ are also connected to endpoints e_s and e_t, respectively. The interface of each device $j = 1, 2, ..., |J|$ that is connected to the next device (or to endpoint e_t for device $|J|$) has telemetry demand equal to l_j, while the other interface has no telemetry demand. The interface demands are shown in italics in the figure.
2. Next, m flows $P_1, P_2, ..., P_m$ are created. Each flow has endpoints e_s and e_t and is routed through a path comprising all device interfaces in G, i.e., $\rho(P_i) = \{(1, e_s), (1, 2), (2, 1), ..., (|J|, e_t)\}$, $i = 1, 2, ..., m$. The telemetry capacity of all flows is n.

Figure 3.5: Example MSP reduction to INTO Balance. An MSP instance – with $|J| = 3$ jobs, number of processors $m = 2$, and deadline $n = 6$ – is reduced to an INTO Concentrate instance – with 3 forwarding devices and 2 flows with telemetry capacity equal to 6.



Theorem 2. *The INTO Balance decision problem is NP-Complete.*

Proof. By the instance of the reduction presented, if an MSP instance has a solution schedule where all jobs finish within n time units, then the INTO Balance problem has a solution with no more than n items assigned to be transported by any single flow. Consider that each job j of length l_j scheduled to a processor P_i corresponds to the coverage of the interface of device j (with positive telemetry demand) by flow P_i. Conversely, if the INTO Balance problem has a solution where each flow is assigned to collect no more

than n telemetry items, then the corresponding MSP instance has a solution where every job complete within n time units. Covering the positive demand interface of device j with flow P_i corresponds to scheduling a job of length l_j to process P_i. Lemmas 3 and 4 complete the proof. □

3.4 Heuristic Algorithms to Solve the INTO Problems

In this section, we propose and formalize two heuristic algorithms (Concentrate and Balance) designed to produce high-quality solutions to the two optimization problems (INTO Concentrate and INTO Balance, respectively) in a timely manner.

3.4.1 Concentrate Heuristic Algorithm

As mentioned in Section 3.3.1, given the challenge of optimizing in-band telemetry orchestration, one possible strategy is to minimize the number of flows that will be used to transport telemetry data. In this section, we propose Concentrate, a heuristic algorithm focused on minimizing this number. Algorithm 1 shows the pseudo-code of Concentrate. Next, we detail its procedures.

The algorithm has two input parameters: the set I of all active device interfaces and the set F of all packet flows in the network. The algorithm maintains two main data structures, CoveredBy and Monitors, that indicate which flow covers each interface (i.e., function Φ) and which interface is to be monitored by each flow, respectively. These data structure also constitute the algorithm's output that is used to generate packet processing rules and configure a network. Initially, no flow has been assigned to monitor any interface yet; in Lines 1-2, CoveredBy and Monitors are initialized to reflect this.

The algorithm also maintains four other auxiliary variables: κ_r (Line 3), NIFs (Line 4), SIFs (Line 5), and UFs (Line 6). Variable κ_r indicates the remaining telemetry capacity available for each flow. It is initialized with the telemetry capacity κ of the flows and is updated as assignments are made by the algorithm. NIFs indicates, for each flow, the number of interfaces not yet covered in its path. Variable SIFs keeps a sorted list of the interfaces in the path of a flow ordered non-decreasingly by the number of flows passing through them (and by the telemetry demand in case of a tie). Set UFs consists of all currently unassigned flows, initially UFs $= F$. Lines 7-16 contain the main repeat loop

Algorithm 1 Concentrate heuristic pseudocode.

Input: I, F

1: $\text{CoveredBy}(i) \leftarrow \text{Null}, \forall i \in I$
2: $\text{Monitors}(f) \leftarrow \varnothing, \forall f \in F$
3: $\kappa_r(f) \leftarrow \kappa(f), \forall f \in F$
4: $\text{NIFs}(f) \leftarrow |\rho(f)|, \forall f \in F$
5: $\text{SIFs}(f) \leftarrow \textsc{Sort}(i \in \rho(f), |\textsc{Flows}(i)|, \delta(i)), \forall f \in F$
6: $\text{UFs} \leftarrow F$
7: **while** $\text{UFs} \neq \varnothing$ **and** $\exists i \in I, CoveredBy(i) = Null$ **do**
8: $\quad f_{max} \leftarrow \textsc{Max}(f \in \text{UFs}, \text{NIFs}(f), \kappa(f))$
9: $\quad \text{UFs} \leftarrow \text{UFs} - f_{max}$
10: $\quad$ **for** $i \in \text{SIF}(f)$ **do**
11: $\quad\quad$ **if** $\text{CoveredBy}(i) = \text{Null}$ **and** $\delta(i) <= \kappa_r(f_{max})$ **then**
12: $\quad\quad\quad \text{CoveredBy}(i) \leftarrow f_{max}$
13: $\quad\quad\quad \text{Monitors}(f_{max}) \leftarrow \text{Monitors}(f_{max}) \cup i$
14: $\quad\quad\quad \kappa_r(f_{max}) \leftarrow \kappa_r(f_{max}) - \delta(i)$
15: $\quad\quad\quad$ **for** $f \in \textsc{Flows}(i)$ **do**
16: $\quad\quad\quad\quad \text{NIFs}(f) \leftarrow \text{NIFs}(f) - 1$

Output: CoveredBy, Monitors

of the algorithm. At each iteration of the outermost loop, the algorithm first selects the unassigned flow f_{max} with the maximum number of interfaces still not covered in its path (Line 7) and removes it from UFs (Line 8). In case of a tie, the algorithm selects any of the flows with the maximum telemetry capacity. After finding f_{max} from F, it is assigned to monitor every interface still uncovered in its path following the ordering given by $\text{SIF}(f)$ (Lines 10-13) and respecting its telemetry capacity (Lines 11). The remaining telemetry capacity is updated after each assignment (Line 14). For every assignment, the value of NIFs of all flows traversing the interface just covered is decreased by one (Lines 15-16), as the interface is no longer uncovered and new flows cannot be assigned to monitor it. The main loop is repeated until every flow has been considered or all interfaces have been covered by the solution.

The worst-case computational complexity of this algorithm is given by $\mathcal{O}(n) + \mathcal{O}(m) + \mathcal{O}(m \cdot n \cdot \log n) + \mathcal{O}((n+m) \cdot (n \cdot m)) = \mathcal{O}(n^2 \cdot m + n \cdot m^2)$, where n is the number of interfaces in I and m is the number of flows in F. Therefore, the algorithm runs in polynomial time to the number of interfaces and flows in the network.

3.4.2 Balance Heuristic Algorithm

In this subsection, we formalize the Balance algorithm, which strives to minimize the maximum number of telemetry items to be transported by any single flow. Algorithm 2 shows the pseudo-code of Balance. Balance has the same input parameters and output data structures of Algorithm 1. Algorithm 2 also maintains the same κ_r and NIFs variables of Concentrate, which indicate the remaining telemetry capacity and the number of interfaces not yet covered in the path of each flow, respectively. Balance has two additional variables: NFs and NCIFs. NFs indicates, for every interface, the number of flows with available capacity passing through it. NCIFs is the set of all intefaces that are not yet covered by any flow.

Algorithm 2 Balance heuristic pseudocode.

Input: I, F

1: $\text{CoveredBy}(i) \leftarrow \text{Null}, \forall i \in \mathbb{I}$
2: $\text{Monitors}(f) \leftarrow \varnothing, \forall f \in F$
3: $\kappa_r(f) \leftarrow \kappa(f), \forall f \in F$
4: $\text{NIFs}(f) \leftarrow |\rho(f)|, \forall f \in F$
5: $\text{NFs}(i) \leftarrow |\textsc{Flows}(i)|, \forall i \in \mathbb{I}$
6: $\text{NCIFs} \leftarrow I$
7: **while** $\text{NCIFs} \neq \varnothing$ **and** $\exists f \in F, \kappa_r(f) > 0$ **do**
8: $\quad i_{mm} \leftarrow \textsc{MinMax}(i \in \text{NCIFs}, \text{NFs}(i), \delta(i))$
9: $\quad \text{NCIFs} \leftarrow \text{NCIFs} - \{i_{mm}\}$
10: $\quad \text{IFFs} \leftarrow \{f : f \in \textsc{Flows}(i_{mm}) \text{ and } \kappa_r(f) \geq \delta(i_{mm})\}$
11: $\quad f_{min} \leftarrow \textsc{Min}(f \in \text{IFFs}, \kappa(f) - \kappa_r(f), \text{NIFs}(f))$
12: $\quad \text{CoveredBy}(i_{mm}) \leftarrow f_{min}$
13: $\quad \text{Monitors}(f_{min}) \leftarrow \text{Monitors}(f_{min}) \cup \{i_{mm}\}$
14: $\quad \kappa_r(f_{min}) \leftarrow \kappa_r(f_{min}) - \delta(i_{mm})$
15: $\quad$ **for** $f \in \textsc{Flows}(i_{mm})$ **do**
16: $\quad\quad \text{NIFs(f)} \leftarrow \text{NIFs}(f) - 1$
17: $\quad$ **if** $\kappa_r(f_{min}) = 0$ **then**
18: $\quad\quad$ **for** $i \in \rho(f_{min}) \cap \text{NCIFs}$ **do**
19: $\quad\quad\quad \text{NFs}(i) \leftarrow \text{NFs}(i) - 1$

Output: CoveredBy, Monitors

Lines 7-19 contain the main repeat loop of the algorithm. At each iteration of the outermost loop, the algorithm first selects the uncovered interface i_{mm} with the minimum number of flows with available capacity passing through it (Line 8) and removes it from NCIFs (Line 9). In case of a tie, any of the interfaces with maximum telemetry demand

may be chosen. Next, from all flows traversing i_{mm}, the f_{min} flow collecting the least amount of telemetry items (i.e., $\kappa(f) - \kappa_r(f)$) out of those with available capacity is selected (Lines 10-11). In case of a tie, the flow with the minimum number of interfaces still not covered in its path is chosen. If the tie persists, the algorithm selects any one of the flows that tied in both of the steps. When i_{mm} and f_{min} have been found, flow f_{min} is assigned to monitor interface i_{mm} (Lines 12-13). After the assignment, some adjustments are made to variables κ_r, NIFs, and NFs. The remaining capacity κ_r of flow f_{min} is decreased by the demand δ of interface i_{mm} (Line 14). The number of uncovered interfaces NIFs is decreased by one for every flow passing through interface i_{mm} (Lines 15-16). If f_{min} has used all its capacity (Line 17) after been assigned to monitor i_{mm}, then it is necessary to update the NFs value for every interface through which f_{min} passes (Lines 18-19). This procedure is repeated until all active device interfaces in the network are covered, or there is not any flow with available capacity.

The worst-case complexity of this algorithm is given by $\mathcal{O}(n+m) + \mathcal{O}((n+m) \cdot (n+m)) = \mathcal{O}(n^2 + m^2)$, where n is the number of interfaces in I and m is the number of flows in F.

3.5 Evaluation

In this section, we present computational experiments with the models and algorithms introduced in the previous two sections of this chapter. We start by describing the experimental setup and the dataset used in the experiments. Then, we detail two sets of experiments. The first set of experiments evaluates the proposed mathematical programming models and heuristic algorithms with regards to solution quality and processing time. We refer to the models of Section 3.3 as CMP (Concentrate) and BMP (Balance) and to the algorithms of Section 3.4 as CH (Concentrate Heuristic) and BH (Balance Heuristic). The second set of experiments compares the heuristics with regards to the performance factors introduced in Section 2.3. We also consider the "full assignment" as a comparison baseline, refering to it as FA.

The experiments were carried out in a computer with an Intel Core i7-5557U CPU running at 3.10GHz, 16GB of DDR3 1867MHz RAM, and Apple macOS 10.12 operating system. The GLPK Solver[2] 4.65 was used to solve the mathematical programs. The

[2]GNU Linear Programming Kit Linear Programming/Mixed-Integer Programming Solver (https://www.gnu.org/software/glpk/).

computation time limit to find a solution was set to 10 minutes. The proposed heuristic algorithms were implemented in C++ and compiled with the Apple LLVM version 9.0.0 (clang-900.0.39.2) compiler.

The dataset used in the experiments is composed of 260 topologies of real wide area networks catalogued by the Internet Topology Zoo (ITZ) project (KNIGHT et al., 2011) and traffic matrices made available in Repetita (GAY; SCHAUS; VISSICCHIO, 2017). ITZ topologies range from 4 to 197 points-of-presence and from 16 to 880 interfaces. The maximum (minimum) network diameter is 37 (3) hops. To generate realistic traffic matrices, Gay *et al.* (GAY; SCHAUS; VISSICCHIO, 2017) followed the randomized gravity model proposed in (ROUGHAN, 2005). We converted the available topologies to INTO problem instances. For every link in the topology, we considered there were two interfaces, one in each extremity. Interface telemetry demands were randomly chosen from the range from four to ten items following a uniform distribution. We choose this range since four can be considered the minimum amount of items to identify the source of a metadata field and its value (e.g., device ID + queue ID + queue ingress timestamp + queue delay) and ten is the number of common metadata fields that can be exported by devices according to (GROUP, 2018). For each pair of network endpoints with positive demands according to the traffic matrices, we considered there is a single aggregate network flow (i.e., we differentiate flows by the pair of network entry point and exit point). Flow telemetry capacities were randomly chosen following a normal distribution. Otherwise stated, the mean telemetry capacity is equal to 35 items, and the standard deviation is 5 items, which amounts for approximately 10% of the typical 1500-byte MTU being available for transporting metadata in each packet.

3.5.1 Mathematical Programming Models and Heuristic Algorithms

To work in practice, a procedure to solve any variation of the INTO problem should be able to provide high-quality solutions within short time intervals. This is necessary so that the monitoring campaign can quickly adapt to network policy changes and traffic fluctuation. In this first set of experiments, we evaluate the solution quality and processing time of the algorithms introduced in Section 3.4, namely Concentrate Heuristic (CH) and Balance Heuristic (BH), by comparing them to the mathematical programming models.

We start by analyzing the processing times of the approaches. Figure 3.6 shows

the time taken by the GLPK Solver to run each of the INTO problem instances for the mathematical programming models. As expected, it grows quickly with relation to instance size for both models. We note that the time to process an instance is also subject to other network characteristics besides size (e.g., diameter, degree of connectivity of nodes, heterogeneity of paths). Thus, in the graphs, we can expect that some networks, although smaller than other, take more time to be solved.

For the Concentrate Mathematical Program (CMP) model, the solver was not able to find an optimal solution and check optimality for networks with more than 90 interfaces (14 devices) within the 10-minute time limit. The solver was able to find the optimal solution within the time limit for the INTO Balance instances with up to 252 interfaces (56 devices) using the Balance Mathematical Program (BMP) model. From these results, we conclude that solving CMP is generally harder than BMP. The difference in processing time between the two approaches can be attributed in part to the fact that the mathematical model CMP has more decision variables than BMP. Models with more variables tend to have larger solution spaces and, thus, take longer to explore their totality in order to check optimality. In summary, Figure 3.6 shows that solving both versions of the INTO problem using the mathematical models takes considerable time, making their use impracticable in real, highly dynamic scenarios.

Figure 3.6: Processing times for the mathematical programming models.

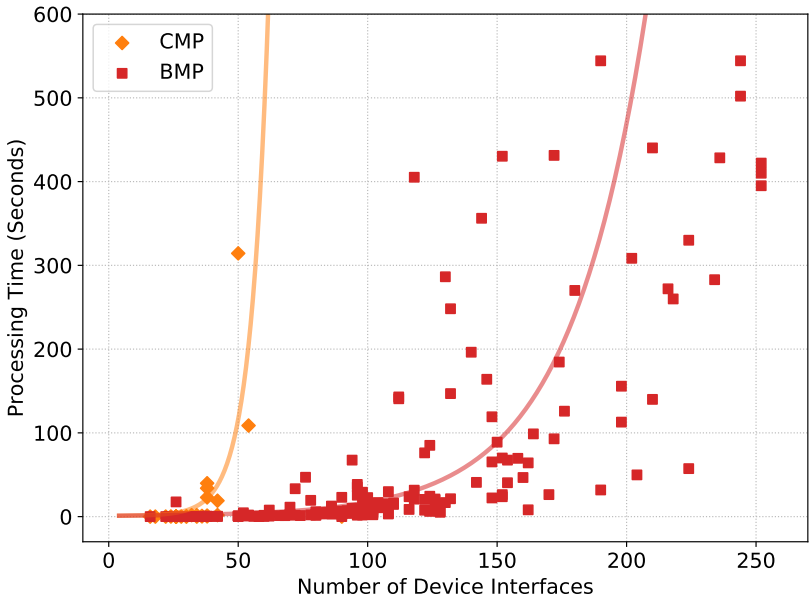
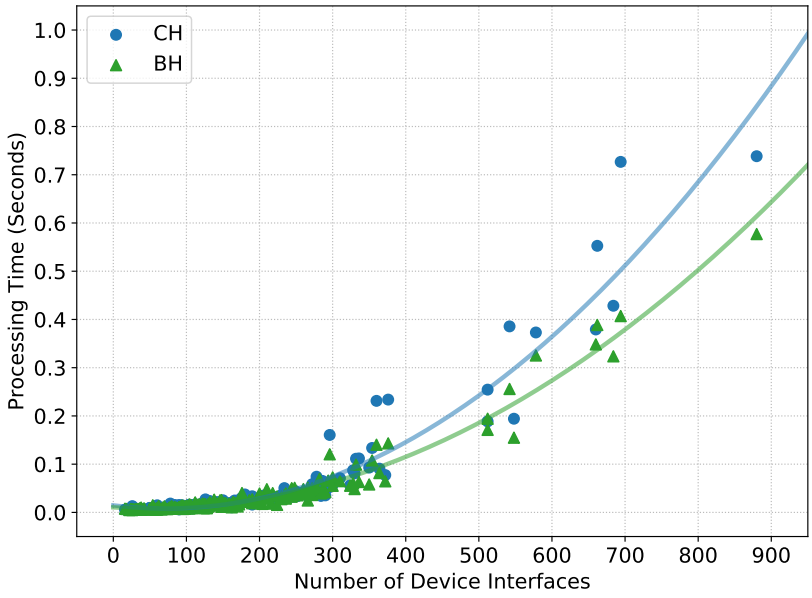


Figure 3.7 shows the processing time of the heuristic algorithms CH and BH as a function of the network size. Both algorithms generate feasible solutions to all instances of their corresponding INTO variation in less than one second. These results are up to

Figure 3.7: Processing times for the heuristic algorithms.



three orders of magnitude lower than the processing times required by the mathematical models. The short processing times achieved by both heuristic algorithms argues in favor of their adequacy to be applied in highly dynamic networks. Next, to confirm that they are adequate, we also evaluate the quality of the solutions given by them.

To evaluate the quality of the solutions provided by the heuristics, we compare their objective function values to lower bound models. We start by comparing the lower bound models and the mathematical programming models to estimate how close to the optimal values are the lower bounds. Then, we compare the lower bounds with the heuristic algorithms solutions considering the estimated gap. The lower bound for INTO Concentrate is computed by exchanging Equation 3.3 of the CMP model by Equation 3.12.

$$x_{i,f} \leq y_f, \forall i \in I, \forall f \in F \tag{3.12}$$

The description of the variables can be found in Table 3.1. The original equation had two purposes: (*i*) guarantee that no flow is assigned to collect and transport more items then its capacity allows and (*ii*) activate the variable y_f (which is used by the objective function of CMP to count the number of telemetry-active flows) when any telemetry item is assigned to flow $f \in F$. With the new equation we assume a scenario where all flows would have unlimited telemetry capacity. The new, simpler constraint presented in Equation 3.12 does not consider telemetry demands and capacities (first purpose), eliminating a summation and making it quicker to compute. Only the second purpose of the

original equation is kept, i.e., all flows performing at least one telemetry action will be accounted for in the objective function.

In our experiments, the solver was able to find feasible solutions for the CMP model for 248 out of the 260 networks. Out of those 248 cases, the solver was able to certify optimality for only 28 instances. No feasible solution was found for 12 problem instances. The mean gap between the lower bound and CMP across all 248 feasible solutions found by the solver was 12.62 flows (with standard deviation equal to 11.68 flows). The minimum and maximum gap values were 0 and 50 flows, respectively. Compared to the lower bound, the mean gap for CH was 9.82, and the standard deviation was 7.93. The minimum and maximum gap values were 0 and 49 flows. Comparing the gaps, we conclude that the solutions generated by CH are slightly better than the feasible solutions provided by the CMP model within the 10-minute time limit. Thus, the CH algorithm is not only able to generate solutions within one second but also provides high-quality solutions.

To evaluate the quality of the solutions found to the INTO Balance variation, we compute a lower bound for each instance as the maximum value between (*i*) the maximum telemetry demand of a single interface and (*ii*) the sum of all demands divided by the total number of flows.

Out of all 260 instances evaluated, for the BMP model, the solver was able to find feasible solutions within the time limit for 223. The solver was able to certify optimality for 174 out of these 223 cases. No feasible solution was found within 10 minutes for 37 instances. In our comparison, the optimal solution matched the lower bound in 171 out of the 174 optimal cases. Considering all 223 feasible solutions found, the mean lower bound to BMP gap was 0.84 items (with standard deviation equal to 2.37). The minimum and maximum gap values were 0 and 14 items, respectively. Concerning the BH algorithm, the mean gap to the INTO Balance lower bound was 0.09 items (the standard deviation was 0.63). The minimum and maximum gap values were 0 and 5 items. The BH algorithm was able to generate an optimal solution for all but 5 instances. Both BMP and BH have solutions with object function values very close to the lower bounds.

From the experiments in this subsection we conclude that both of the proposed algorithms, CH and BH, generate high-quality solutions within short processing times, and, thus, are well suited to be applied in highly dynamic networks.

3.5.2 Comparison of the INTO problem variations

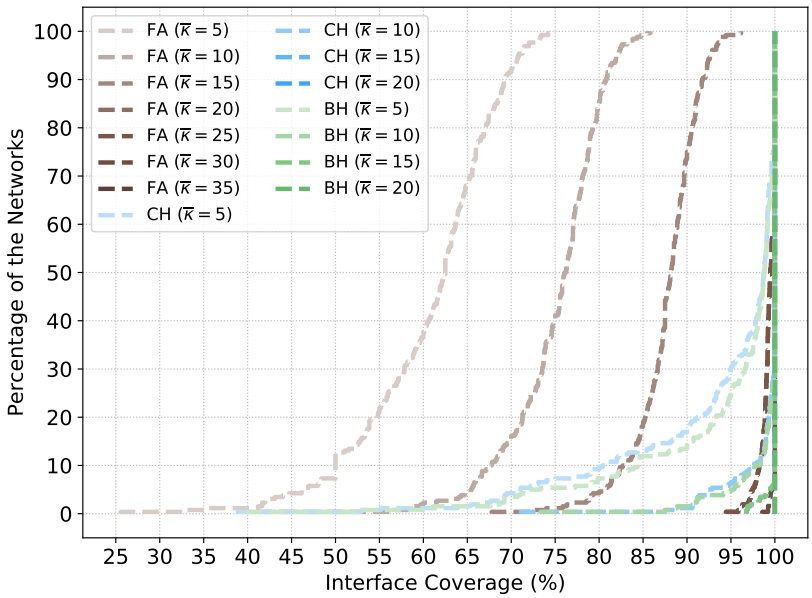
When optimizing the use of INT to perform monitoring, an operator may opt to configure the network according to one of the two variations of the INTO problem. In this remaining set of experiments, we compare the solutions generated by the proposed heuristic algorithms (CH and BH) and the baseline full assignment (FA). To evaluate the algorithms, we consider the main INT constraints and performance influencing factors introduced in Section 2.3.

Maximizing Interface Coverage

As previously discussed, the telemetry capacity of flow packets in a network may limit how many of the device interfaces can be monitored. In this subsection, we evaluate how sensitive the INT orchestration strategies are regarding interface coverage.

For each of the assignment strategies, we set the initial mean packet telemetry capacity to 5 items (the standard deviation was kept at 5 items for the whole experiment). We then calculated the interface coverage – i.e., the percentage of device interfaces covered by at least one flow – for every network of the Internet Topology Zoo (KNIGHT et al., 2011). Finally, we increase the mean capacity in steps of 5 items until each assignment strategy was able to provide a solution with full (100%) coverage for all networks. Fig. 3.8 presents the results of this experiment as a CDF where the x-axis indicates cover-

Figure 3.8: Interface Coverage.



age levels and the y-axis indicates percentages of the networks. For every pair of strategy and capacity level, we plot a curve in the graph. For the full assignment (FA curves in the plot), we report the real coverage levels that would be achieved without exceeding the packet telemetry capacity.

The main conclusion from the results presented in Fig. 3.8 is that the proposed heuristics (CH and BH) are more resilient to lower levels of telemetry capacity than the full assignment (FA). Even when the capacity is at its lowest level (5 items), they can achieve 100% interface coverage (full coverage) for about 25% of the networks. Furthermore, in the same scenario, about 80% of the networks have good coverage of at least 90% of interfaces. CH and BH present similar coverage for all capacity levels, with a slightly better, yet noticeable result observed for the latter. While both of the heuristics achieve full coverage when the mean telemetry capacity is 20 items, the full assignment only achieve it when that capacity is at least 35 items.

In the following subsections, to avoid biasing or tainting the comparison, we will only present and consider the results where each strategy was able to achieve full coverage for all networks. That is, when the capacity is of at least 35 and 20 items for the full assignment and heuristic algorithms, respectively. Next, we continue the evaluation by considering quality and cost metrics related to the four performance factors introduced in Section 2.3.

Minimizing Flow Packet Load

One of the INT factors that may influence network performance is intrinsically related to the extent to which packets vary in size along their paths. The number of telemetry items flow packets collect from device interfaces (i.e., its telemetry load) determines this variation. The best approach to avoid causing significant jitter or drift in transmission times is to minimize the telemetry load of packets. Figure 3.9 presents the cumulative distribution function of the mean flow packet load. For each orchestration strategy, the figure presents a curve representing the distribution of values when the mean telemetry capacity ($\bar{\kappa}$) is 35 items. The figure also presents 95% confidence intervals as filled areas in the graph.

The results in Figure 3.9 show that BH achieves the best possible telemetry load per flow packet. The mean load is approximately seven items, which is also the mean telemetry demand of interfaces. This value indicates that in its solutions, BH tends to assign each telemetry-active flow to monitor one interface. The variability shown in the

Figure 3.9: Mean and confidence interval for flow packet load when the mean telemetry capacity is 35 items.

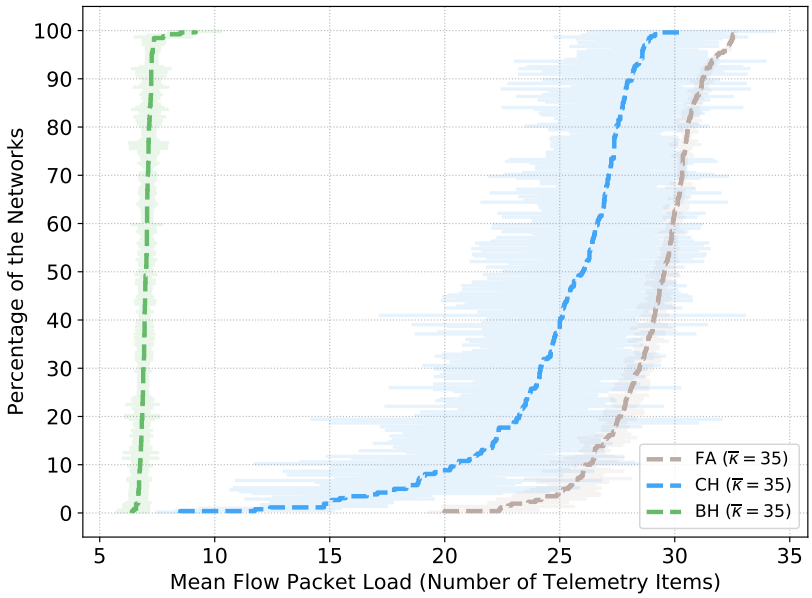
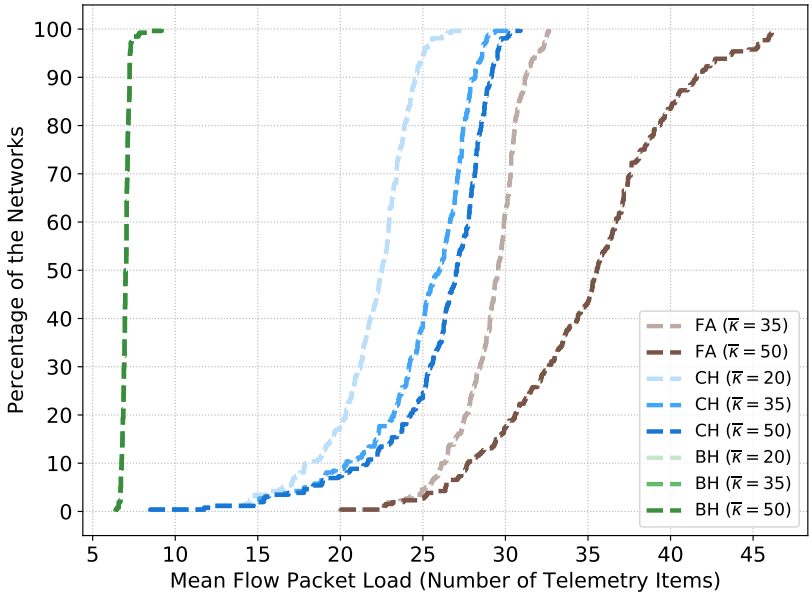


Figure 3.10: Evaluation of flow load as telemetry capacity varies.



graph for BH is due to the variation of the interface telemetry demands. In all cases, CH and FA present higher load values than BH. CH presents higher values than BH because when minimizing the number of telemetry-active flows it tends to use as much as possible the capacity of (a small subset of) flows. CH also presents high variability for flow packet load, which is due to its greedy rationale to flow assignment. Its heuristic causes a group of flows to use most of their packet capacity while another group uses only a fraction to cover the remaining interfaces.

Figure 3.10 illustrates the influence of the mean capacity of flow packets over the mean telemetry load. It reveals that BH is not influenced by the capacity. This phenomenon was expected since an increase in capacity has no impact in minimizing the maximum flow packet load (i.e., the optimization goal of BH). CH presents small increases in flow load as capacity increases. In turn, FA increases significantly the telemetry load imposed on flows when more capacity is available, which aligns with its objective of collecting as many items as possible with all flows. In a case where the telemetry capacity is assumed to be infinite, the mean flow packet load of FA would be an approximation of the mean telemetry demand of all flow paths in a network.

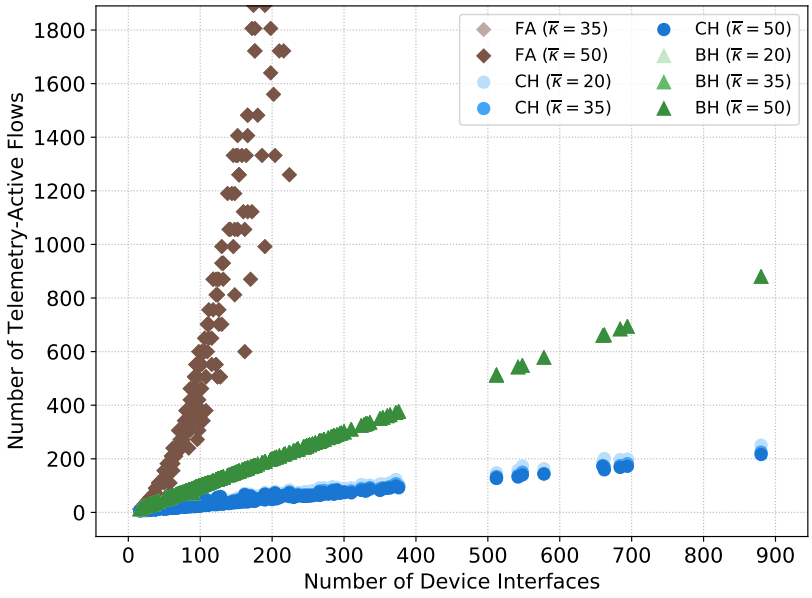
Minimizing Flow Usage

The limitation on the processing capacity of forwarding devices and monitoring sinks (Section 2.3, Items (b) and (c)) prompts orchestration strategies to minimize the number of telemetry reports generated periodically. This minimization has the objective of alleviating the packet processing overhead on both forwarding devices and monitoring sinks/analyzers. In this subsection, we evaluate the flow usage (i.e., the number of telemetry-active flows) demanded by the strategies, as this number directly influences the number of reports generated.

Figure 3.11 presents the flow usage as a function of the number of device interfaces in a network. For every combination of strategy, telemetry capacity and network we plot a point in the graph. We limit the y-axis to $1\,800$ flows to be able to compare the flow usage of the proposed heuristics. The total number of flows for the largest tested network (i.e., the one with 880 interfaces) is $38\,612$.

CH uses the minimum amount of flows across all strategies, which was expected since its main optimization objective is to minimize this value. CH typically assigns each telemetry-active flow to monitor about four interfaces, a 4:1 interface to flow ratio. BH presents a direct relationship between the number of active flows and of interfaces, i.e., each flow covers a single interface (1:1 ratio). Thus, BH uses, in the general case, four times more flows than CH. For a mean telemetry capacity of 35 items, CH uses 225 flows to cover the largest network in the dataset, which has 880 forwarding device interfaces, while BH uses 880 flows. For the same scenario, FA assigns all $38\,612$ flows (not shown in the graph) in the network to carry out telemetry. Thus, both of the heuristics use up to two orders of magnitude less flows then FA. CH is the most scalable of the strategies, followed by BH with a multiplicative increase by a constant factor of about four.

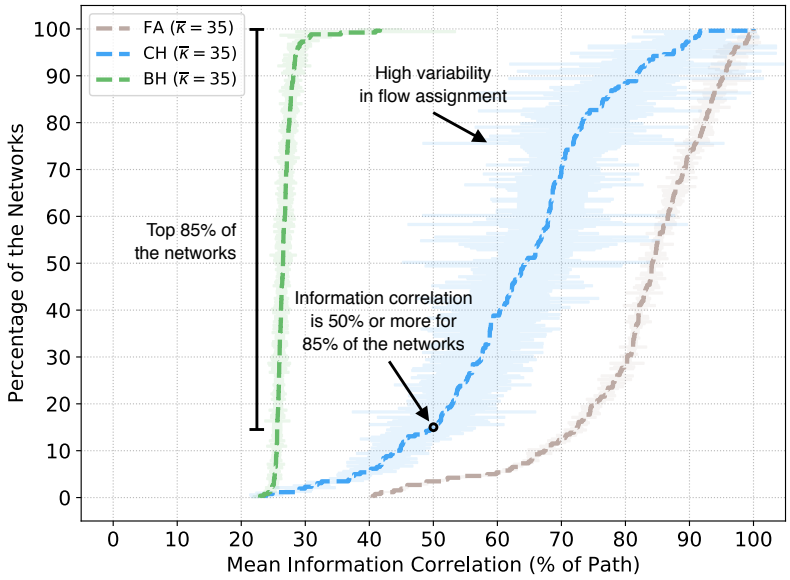
Figure 3.11: Flow usage as a function of the number of interfaces in a network.



Maximizing Information Correlation

In addition to minimizing the number of telemetry-active flows, the limited processing capacity of monitoring sinks (Section 2.3, Item (c)) also motivates strategies to maximize the correlation of the information contained in each telemetry report received by the monitoring sinks, simplifying their analysis. In this evaluation, we consider the percentage of interfaces a flow covers from its path as a measure of this correlation.

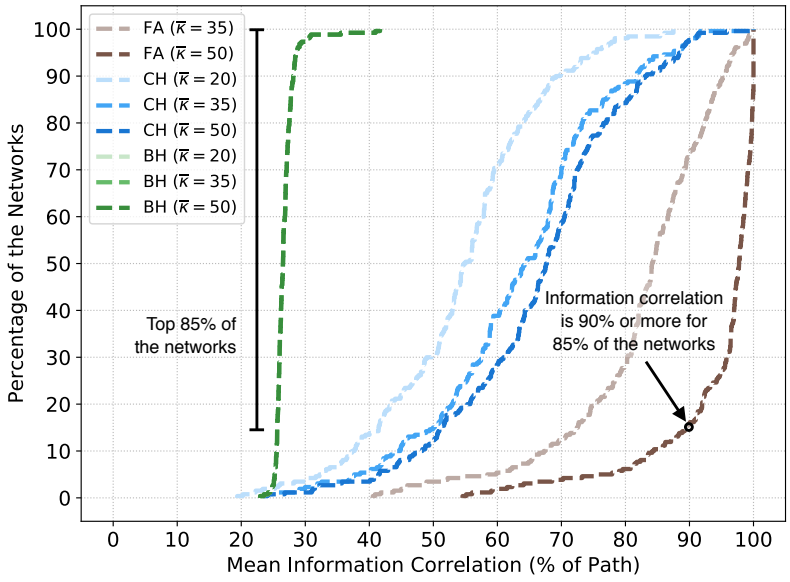
Figure 3.12: Mean and confidence interval for information correlation when the mean telemetry capacity is 35 items.



The results of the experiment are shown in Figures 3.12 and 3.13. Figure 3.12

presents the CDF of the mean information correlation for the networks. The graph has a curve for each evaluated strategy showing the distribution of values when the mean telemetry capacity is 35 items. The figure also presents the 95% confidence intervals as filled areas in the graph. Figure 3.13 shows the effect of the telemetry capacity on the mean correlation. For clarity, we omit confidence intervals in this second figure.

Figure 3.13: Evaluation of information correlation as the telemetry capacity of flows varies.



The results in Figures 3.12 and 3.13 support the expected behavior of the heuristics. According to Figure 3.12 CH presents the best results among the heuristics, with (the top) 85% of networks having 50% or more information correlation. Note in the figure that the lower 15% of the networks (0%–15%) have an x-axis value below 50, while the remaining 85% (15%–100%) have a value equal or above 50. This means that a typical report covers at least half the path of a flow. We highlight that the results for FA in Figure 3.13 show that reaching high correlation is very costly in terms of packet telemetry capacity. More specifically, to achieve 90% correlation for about 85% of the networks, the mean telemetry capacity must be about 50 items per packet. Observe (in Figure 3.13) that the lower 15% of the networks have an x-axis value below 90 for curve FA ($\bar{\kappa} = 50$), while the remaining upper 85% have values above 90.

Going back to Figure 3.12, it also presents the significant variability in flow assignment (see filled area) regarding CH found previously in Figure 3.9. This behavior confirms the previous conclusion that CH tends to assign a group of flows to cover (almost) their entire path, while another group covers only a few remaining interfaces each. BH is not influenced by telemetry capacity variation. This is concluded by the fact that

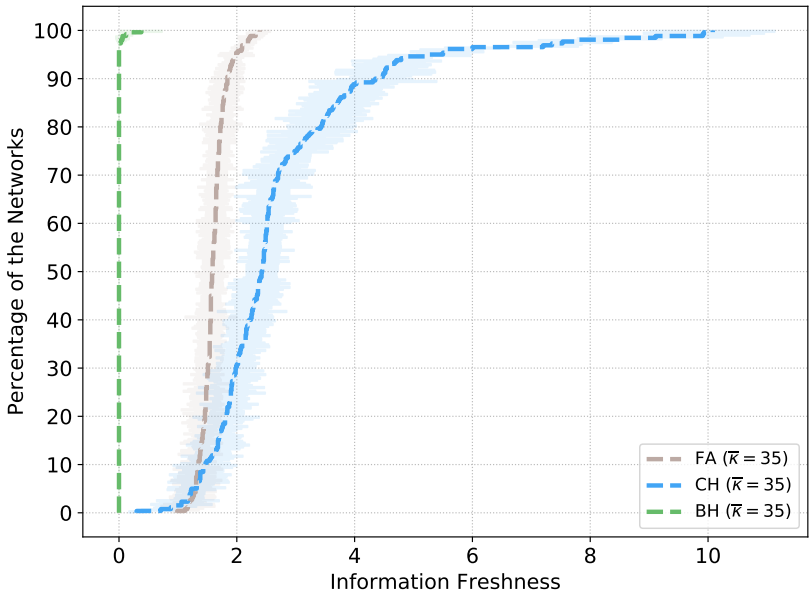
all curves for BH in Figure 3.13, representing different levels of telemetry capacity, superimpose each other. BH achieves at least 25% correlation for most of the networks in all scenarios, i.e., flows tend to cover about one-fourth of their paths.

Optimizing Information Freshness

As the last aspect of our comparison, we consider the fact that network links have finite bandwidth (Section 2.3, Item (d)). The limitation on link bandwidth argues for (*i*) avoiding transporting telemetry data through too many links in a network and for (*ii*) distributing the origin of telemetry reports as much as possible among network devices. To measure the extent to which strategies conform to this task we use the mean information freshness, i.e., the mean number of hops the information collected at interfaces is transported in-band before being reported to a monitor sink. This metric is connected to the bandwidth limitation since as information stays longer in a packet it losses its freshness and leads to higher bandwidth consumption.

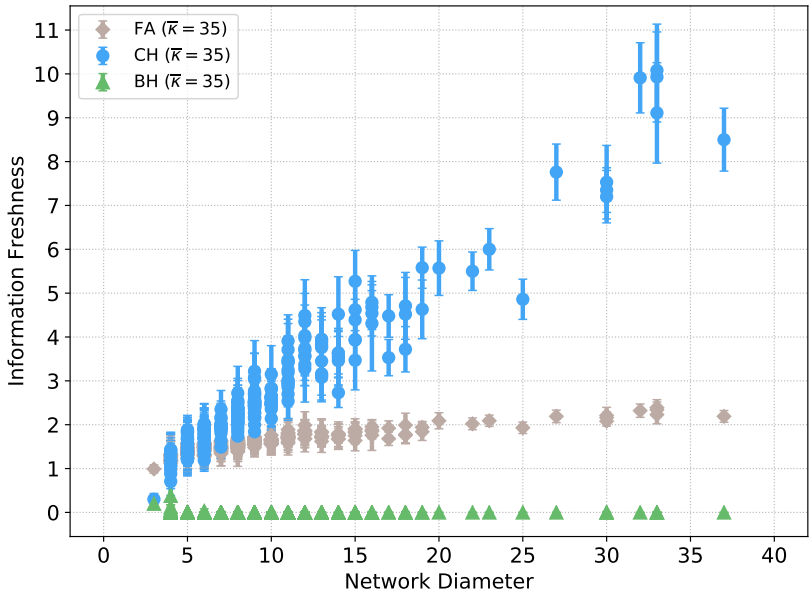
Figures 3.14 and 3.15 present the freshness results. Figure 3.14 shows the CDF of the mean information freshness considering the three orchestration strategies. The filled areas in the graph represent the 95% confidence intervals. Figure 3.15 shows the freshness as a function of the network diameter (i.e., the length of the longest flow path).

Figure 3.14: Mean and confidence interval for information freshness with mean telemetry capacity equal to 35 items.



BH keeps information freshness at the optimal value (zero) for most of the networks analyzed (Figure 3.14). A value zero for freshness indicates that most information

Figure 3.15: Information freshness as a function of the network diameter.



is reported to a monitoring sink immediately after being collected, which results in the best possible monitoring traffic distribution across a network. Figure 3.15 indicates that the network diameter has little to no effect on freshness for BH. FA typically causes flows to transport telemetry information for about two hops before reporting it. The freshness has slightly worse values for larger networks. CH presents the worst freshness values among strategies and is significantly influenced by the network diameter. CH comes to the point of making flows transport information up to 10 hops (in the mean case) for networks with a diameter of about 33 hops. Thus, CH tends to concentrate more the points where reports are generated, which may lead to link saturation.

Summary of Results

Table 3.2 summarizes the results of our evaluation. Comparing the heuristics against each other and with the baseline full assignment approach, we first observed that both of the heuristics scale well with network size. This contrasts with the full assignment, which causes considerable overhead on network traffic and devices. Furthermore, the heuristics are also better at achieving full interface coverage. We also concluded that the heuristic to solve INTO Concentrate (CH) performs well in minimizing the overhead on forwarding devices and monitoring sinks. The number of telemetry reports that have to be generated periodically is minimal, and the information reported to sinks has a significant level of correlation to ease their analysis. As a consequence, CH is the most scalable of

Table 3.2: Summary of the evaluation results.

Approach	Scalability	Interface Coverage	Packet Load	Flow Usage	Information Correlation	Information Freshness
FA	Poor –	Fair ○	Fair ○	Poor –	Best ●	Fair ○
CH	Best ●	Best ●	Good ◐	Best ●	Good ◐	Fair ○
BH	Good ◐	Best ●	Best ●	Good ◐	Fair ○	Best ●

the strategies and may be particularly recommended for medium to large networks. The INTO Balance heuristic (BH) imposes the lowest telemetry load per monitoring-active flow and results in the best distribution of the telemetry load among forwarding devices and links in a network. It also provides data with the most freshness. These results suggest that BH may be an adequate strategy for low latency networks or when monitoring traffic highly sensitive to packet size changes.

3.6 Related Work

In this section, we review previous work that investigated the use of forwarding device mechanisms to monitor networks and services. NetSight (HANDIGOL et al., 2014) has packet mirroring as its fundamental monitoring piece, which is available in both OpenFlow (MCKEOWN et al., 2008) and traditional switches. In NetSight, every switch in the path of a packet creates a copy of it, called postcard, and sends it to a logically centralized control plane. The multiple postcards of a single packet passing through the network are combined in the control plane to form a packet history that tells the complete path the packet took inside the network and the modifications it underwent along the way. Depending on the size and level of activity of the network, NetSight may generate a considerable volume of monitoring data. To overcome this issue, Everflow (ZHU et al., 2015) applies a match+action mechanism to filter packets and decide which ones should be mirrored, thus reducing the overheads at the cost of monitoring accuracy and level of detail. Stroboscope (TILMANS et al., 2018) also applies packet mirroring to monitor networks. It answers monitoring queries by scheduling the mirroring of millisecond-long slices of traffic while considering a budget to avoid network performance degradation.

Payless (CHOWDHURY et al., 2014) and AdaptiveSampling (CHENG; YU, 2017) followed a different direction of the previous work. They make use of two OpenFlow-enabled switch features to monitor a network: (i) the capability to store statistics related

to table entries (which may represent flows) and (ii) the possibility of exporting these statistics via a polling mechanism. In these works, the polling frequency influences the quality (e.g., freshness) of the monitoring data. The higher the frequency, the more fine-grained the data. Payless and AdaptiveSampling dynamically and autonomously adjust the polling frequency to achieve a good trade-off between monitoring quality and cost. Another work in the context of software-defined measurement is DREAM (MOSHREF et al., 2014). DREAM is a system that dynamically adapts resources (i.e., TCAM entries) allocated to measurement tasks and partitions each of them among devices. DREAM supports multiple concurrent tasks and ensures that estimated values meet operator-specified levels of accuracy. Payless, AdaptiveSampling, and DREAM have the goal of monitoring traffic characteristics and keeping costs low. Assessing the network state (e.g., switch queue occupancy) is outside their scope.

The idea of orchestrating monitoring data collection across multiple devices to reduce costs is not completely new. Several investigations have been carried out with that objective. For example, Cormode *et al.* (CORMODE et al., 2005) proposed configuring devices to monitor their local variables independently and only report their values (to a centralized coordinator) when significant changes are observed. A-GAP (PRIETO; STADLER, 2007) and H-GAP (JURCA; STADLER, 2010) organize measurement nodes into a logical tree graph. When a significant change is observed for a local variable, the node sends an update to its parent node. The latter is responsible for aggregating the values from multiple devices and sending updates up the tree upon significant value change. The root node maintains network-wide aggregate values to be used for monitoring analysis. Tangari *et al.* (TANGARI et al., 2017) proposed decentralizing monitoring control. The monitoring control plane is composed of multiple distributed modules, each capable of performing measurement tasks independently. Their approach logically divides the network into multiple monitoring contexts, each with specific requirements. As a result, monitoring data tends to be aggregated and analyzed close to the source, reducing the communication overhead considerably. The rationale behind these works was conceived before the emergence of programmable data planes without considering the opportunity to program forwarding devices. As a consequence, in contrast to INTO, their solutions are based on traditional coarse-grained counters or aggregate statistics, and their data exchange models operate at control plane timescales.

Another work which shares the context of INTO is Sonata (GUPTA et al., 2018), a system that coordinates the collection and analysis of network traffic to answer operator-

defined monitoring queries. These queries consist of a sequence of dataflow operators (e.g., filter, map, reduce) to be executed over a stream of packets. Sonata is based upon stream processing systems and programmable forwarding devices, partitioning queries across them. Similar to INTO, it has the goal of minimizing the amount of data that needs to be sent to and analyzed by the control plane (i.e., stream processors). Sonata leverages data plane programmability to offload as much as possible of each query to the forwarding devices, considerably reducing the packet stream before sending it to stream processors (i.e., analyzing servers). With the same goal in mind, TurboFlow (SONCHACK et al., 2018), offloads as much as possible of a flow record generation procedure to forwarding devices, splitting responsibilities between their packet forwarding engine and their local CPUs. Different from INTO – which targets monitoring the network condition – Sonata and TurboFlow focus on the analysis of traffic characteristics. Furthermore, neither of these systems applies in-band network telemetry for metadata reporting. In the next chapter, we will explore offloading (part of) monitoring tasks to the data plane with the proposal of INTSIGHT.

In our work, we revisit the challenge of monitoring networks effectively and efficiently. We identify the new opportunities for obtaining accurate and fine-grained information about forwarding device state, behavior, and performance, considering the recent advances in SDN. We provide a foundation to help to understand the trade-offs involved in conducting network monitoring via a mechanism such as INT by identifying the key factors that impact performance associated with it. We formalize in-band network telemetry orchestration as an optimization problem. We also prove that it is an NP-Complete problem, proposing integer linear programming models to solve it. Finally, we devise heuristic algorithms to produce high-quality solutions within strict computational time.

3.7 Chapter Summary

This chapter introduced the In-band Network Telemetry Orchestration (INTO) problem. It consists in minimizing monitoring overheads in the data, control, and management planes when using In-band Network Telemetry (INT) to collect the state and behavior of forwarding device interfaces. We formalized two variations of the INTO problem and proposed integer linear programming models to solve them. The first variation of the problem, INTO Concentrate, has the goal of minimizing the number of flows used for telemetry. The second variation, INTO Balance, seeks to minimize the maxi-

mum telemetry load (i.e., the number of telemetry items transported) among packets of distinct flows. We proved that both variations belong to the NP-Complete class of problems. Through our evaluation using real network topologies, we found that both of the proposed models do indeed take a long time to solve the INTO problem. This result motivated us to design two heuristic algorithms to produce feasible solutions in polynomial computational time to the network size and number of flows. Experimental results show that the proposed heuristics produce high-quality solutions in under one second for all of the real networks evaluated. When comparing the heuristics against each other, the INTO Concentrate heuristic was found to be the most scalable, making it particularly useful for medium- and large-scale networks. The INTO Balance heuristic, in its turn, was found to be the most adequate strategy for low latency networks. We highlight that both heuristics scale well with network size, contrasting them from the full assignment, as the latter causes considerable overhead on network traffic and devices.

4 DIAGNOSING SLO VIOLATIONS WITH IN-BAND NETWORK TELEMETRY

In this chapter, we present our first step taken towards answering Questions 2 and 3 described in Section 1.2. These research questions relate to pre-processing and consolidating monitoring data in forwarding devices and using that data to detect and react to network problems in short timescales. We propose a system that offloads the tasks of aggregating and summarizing metadata and of detecting SLO violations to the data plane[1]. This chapter is organized as follows. In Section 4.1, we further motivate the need for fine-grained, accurate, and timely SLO monitoring in communication networks. In Section 4.2, we present an overview of INTSIGHT, our proposed system for network monitoring. In Section 4.3, we describe the design and implementation of INTSIGHT. In Section 4.4, we present our evaluation, where we test the effectiveness of INTSIGHT's approach and compare it with existing monitoring techniques. In Sections 4.5 and 4.6, we discuss the related work and INTSIGHT limitations, respectively. In Section 4.7, we conclude the chapter with our main considerations and lessons learned.

4.1 Motivation

Consider a communication network that serves multiple traffic classes, each with different performance expectations, i.e., service-level objectives (SLOs). An SLO might prescribe, for example, that the end-to-end delay has to be less than $5\,\mathrm{milliseconds}$ in 95% of the cases for telesurgery traffic, or that the provided bandwidth has to be higher than $1\,\mathrm{Gbps}$ at least 99% of the time (within a minute) for a video streaming traffic aggregate. While it is possible to build a highly over-provisioned network that can satisfy all SLOs simultaneously under worst-case conditions, such a network would be prohibitively expensive and inefficient. In practice, operators provision for the expected case. They rely on mechanisms for routing, packet scheduling, traffic shaping and isolation, and load balancing (GOVINDAN et al., 2016) (to cite a few) to correct performance problems and cater to fluctuations in demand. In this regime, it becomes imperative for network opera-

[1]This chapter is based on the following publication:

- Jonatas Adilson Marques, Kirill Levchenko, Luciano Paschoal Gaspary. **IntSight: Diagnosing SLO Violations with In-Band Network Telemetry.** ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2020) (MARQUES; LEVCHENKO; GASPARY, 2020a).

tors to be able to *detect* and *diagnose* SLO violations quickly. Next, we discuss these two requirements for network monitoring in more detail.

Requirement 1: Detect SLO Violations

This requirement refers to noticing violations whenever they happen inside the network. In this work, we focus on violations of bandwidth and latency SLOs, arguably the most commonly used and important ones after availability. Detecting a bandwidth violation entails monitoring the instantaneous bandwidth provided to flows. For latency, it requires inspecting the end-to-end delay of every packet subject to the SLO. In both cases, aggregated or averaged values, although sometimes useful, limit the effectiveness of the detection by hiding short-lived fluctuations in traffic (e.g., microbursts (ZHANG et al., 2017)). In summary, detecting violations requires observing the network traffic with very fine granularity, i.e., up to per-packet analysis.

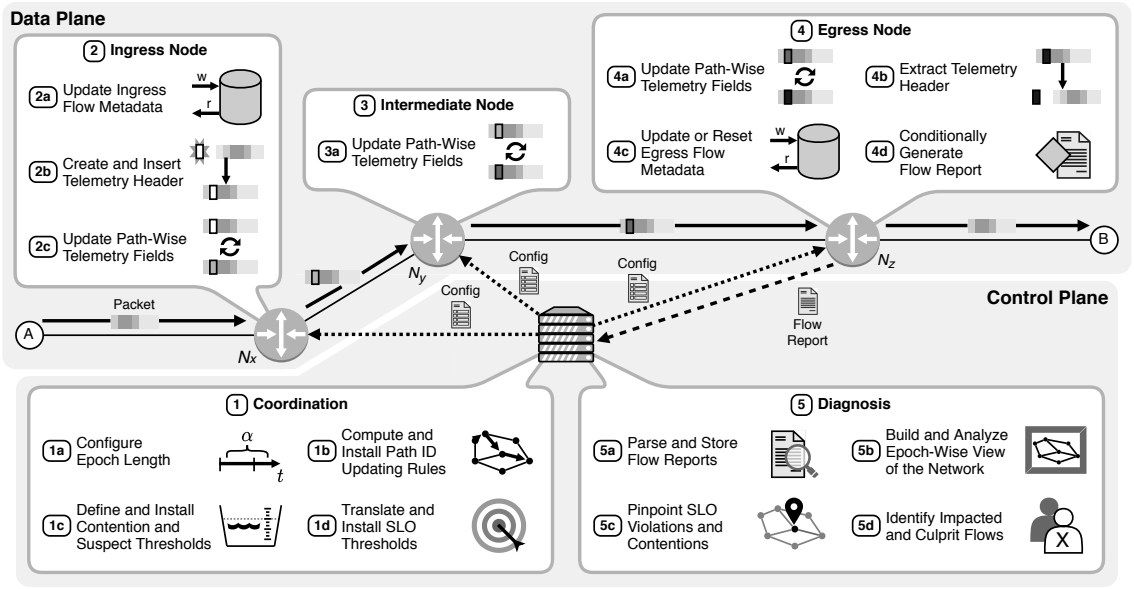
Requirement 2: Diagnose SLO Violations

Given that a violation was detected, another essential requirement is to identify its root cause. Diagnosing a bandwidth SLO violation boils down to identifying the hops along the path of the affected flow where packets were dropped, and the competing traffic at these points. The violation of an end-to-end delay SLO is diagnosed by identifying hops where the flow of interest was substantially delayed, as well as the competing traffic at these points. In short, diagnosing violations means pinpointing forwarding devices experiencing contentions or even dropping packets, and identifying the flows with active traffic on these devices.

4.2 INTSIGHT

In this section, we present an overview of INTSIGHT's approach to detect and diagnose SLO violations, focusing on end-to-end delay and bandwidth SLOs. It is inspired by in-band network telemetry (INT), but, in contrast to its classic approach of device-wise metadata collection (KIM et al., 2015), INTSIGHT uses telemetry headers as workspaces where forwarding devices compute *path-wise metadata* (e.g., path IDs, contention points, and end-to-end delays) progressively. Detection in the data plane, then, becomes a matter of comparing the computed values with SLO-defined ones. The proposed system enables

Figure 4.1: Overview of INTSIGHT's main procedures in the data and control planes.



forwarding devices to summarize monitoring data and *selectively* report events of interest to the control plane, where SLO violation *diagnosis* takes place. In Section 4.2.1, we start our overview by presenting how forwarding devices running INTSIGHT monitor packets and report events of interest[2]. Then, in Section 4.2.2, we describe how INTSIGHT's control plane application (*i*) coordinates the monitoring tasks in the data plane and how it (*ii*) analyzes the information reported by data plane devices to diagnose SLO violations.

4.2.1 INTSIGHT Data Plane

Figure 4.1 illustrates INTSIGHT's approach to network monitoring. It exemplifies the trajectory of a data packet on a network monitored by INTSIGHT. For this example, as well as for INTSIGHT's general design, we consider all forwarding devices are capable (e.g., through data plane programmability) and engaged in monitoring the traffic[3]. To guide the overview, we consider a flow of packets subject to SLOs for both end-to-end delay and provided bandwidth. The packets of this SLO traffic class (henceforth referred to simply as a *flow*) enter the network through end-point A, are routed through path $[N_x, N_y, N_z]$, and leave the network to reach end-point B. The upper section of the figure shows the procedures executed by devices according to their roles, which can be an ingress, intermediate, or egress-node role.

[2]We note that INTSIGHT does not replace or interfere with standard packet processing and forwarding procedures.

[3]In Section 4.6, we discuss the effects of partial network support to INTSIGHT.

In our example, device N_x represents the *ingress node*, i.e., the first device in the network to receive packets from our example flow. Ingress nodes have two main tasks: (*i*) storing persistent metadata about the traffic entering the network through them and (*ii*) bootstrapping the telemetry process for the packets. The first task is particularly important since these devices observe the traffic as it enters the network, providing a more accurate view of its demands. When device N_x receives a data packet coming from end-point A, it follows the three steps described next. Initially, it updates the values of the metadata fields (which are stored in arrays of registers, further explained in §4.3.2) associated with the flow to which the packet belongs (Step 2a in Figure 4.1). Table 4.1 enumerates the metadata fields that ingress nodes save for the flows. They are `Epoch`, `RxPkts`, and `RxBits`. Field `Epoch` is set to the current epoch according to the clock of the device. Fields `RxBits` and `RxPkts` are incremented by packet length and one, respectively. Next, N_x creates and inserts a telemetry header into the packet containing fields such as `Epoch` and `E2EDelay` (Step 2b). The full list of fields present in telemetry headers are also presented in Table 4.1. These fields are initialized considering current flow metadata and default values. For example, field `Epoch` is set to the same value stored in the node, and field `E2EDelay` is set to zero.

Following the two previous steps (i.e., 2a and 2b), N_x updates the path-wise fields (i.e., those marked with an asterisk in Table 4.1) in the telemetry header according to what the packet observed in this node (Step 2c). More specifically, field `PathID` is updated (according to a pre-configured lookup table) to reflect the forwarding decision made by the device (see §4.3.3). Field `ContentionPts` is marked in the case contention was observed in the device, characterized by high queue occupancy. Field `SuspicionPts` is marked when the rate of the flow is held accountable for the high utilization of the outgoing port or link considering its capacity. For both fields `ContentionPts` and `SuspicionPts`, contention and suspicion are determined by comparing measured values with thresholds pre-configured in lookup tables (further explained in §4.3.5). Finally, the delay the packet was subjected to on the device is added to field `E2EDelay` (§4.3.6).

Intermediate nodes (N_y in our example) take only a single step: they update the path-wise fields in the telemetry header (Step 3a). This is equivalent to Step 2c on ingress nodes. Compared to processing demands on ingress and egress nodes, the one on intermediate nodes is the simplest and least resource-intensive. For example, intermediate nodes do not persistently store any metadata concerning the traffic they forward. This exempts network core devices from having the same processing and memory capacity as

Table 4.1: Metadata fields maintained by IntSight at ingress devices, telemetry headers, and/or egress devices for each flow. Path-wise fields are marked with an asterisk (*).

Metadata Field	Description	Ingress Device	IntSight Header	Egress Device	IntSight Report
`Epoch`	Most recent epoch in which a packet of the flow entered the network.	✓	✓	✓	✓
`RxPkts`	Number of flow packets *received* by the network so far in the epoch.	✓	✓	✓	✓
`RxBits`	Number of flow bits *received* by the network so far in the epoch.	✓	✓	✓	✓
`PathID*`	A tuple $\langle$*source*, *destination*, *length*, *code*$\rangle$ that uniquely identifies the path traversed by packets of the flow. (initially: $\langle$*ingress node*, *null*, 0, 1$\rangle$)	–	✓	✓	✓
`ContentionPts*`	Bit array indicating the points of contention in the path of a packet. (initially: 0)	–	✓	✓	✓
`SuspicionPts*`	Bit array indicating the points where the traffic used a significant percentage of a device/link capacity in the epoch. (initially: 0)	–	✓	✓	✓
`E2EDelay*`	Sum of delays of each hop observed by a packet in its path. (initially: 0)	–	✓	–	–
`HighDelays`	Number of highly delayed flow packets so far in the epoch.	–	–	✓	✓
`TxPkts`	Number of flow packets *transmitted* out of the network so far in the current epoch.	–	–	✓	✓
`TxBits`	Number of flow bits *transmitted* out of the network so far in the epoch.	–	–	✓	✓
`EgressEpoch`	Most recent epoch in which a packet of the flow left the network.	–	–	✓	✓
`FlowID`	Code number that uniquely identifies a flow on a forwarding device.	–	–	–	✓

edge devices.

An *egress node*, i.e., the last device to process a packet before it leaves the network (N_z in Figure 4.1), is tasked with the most important steps in the monitoring procedure. After updating path-wise fields (Step 4a, same as Steps 2c and 3), it extracts the packet's telemetry header (Step 4b). This makes the packet return to its original format before being forwarded to the end-point (B in our example). Next, the egress node updates the persistent metadata fields regarding the flow (see Table 4.1) by considering the telemetry field values of the extracted header (Step 4c). Similar to ingress nodes, egress nodes store metadata fields in arrays of registers.

The update process is carried out as follows. Field `Epoch` is only updated when a new epoch starts. Fields `RxPkts` and `RxBits` are updated with the maximum values between those contained in the telemetry header and those in the registers (avoiding data corruption due to out-of-order packet arrivals, further discussed in §4.6). Field `PathID` is compared to the stored value to check for path changes (potentially triggering an additional path change report, see Section 4.3.4). Fields `ContentionPts` and `SuspicionPts` are updated (bitwise OR operation) with the new contention and suspicion points observed by the packet being processed. Field `HighDelays` is incremented by one if the value of field `E2EDelay` exceeds the SLO-defined end-to-end delay threshold (e.g., $10\,\mathrm{ms}$) for the flow (the SLO thresholds, further described in §4.3.6, are stored and queried from lookup tables filled in advance by the control plane). Fields `TxPkts` and `TxBits` are updated to account for the current packet being forwarded. Field `EgressEpoch` is updated with the current epoch in the egress node.

In case the currently received packet indicates the start of a new epoch for its flow, the egress node resets all fields to default values and then updates their values according to the telemetry header just extracted. The last step in an egress node is to conditionally generate flow reports (Step 4d), which are control packets with all the metadata stored for the respective flow during a monitoring epoch. These reports also include field `FlowID` (see Table 4.1), which indicates the unique index on the arrays of registers where the metadata about the flow was stored in the reporting device (see §4.3.2). A report is generated if, and only if, both an epoch just ended *and* an event of interest was observed for the flow. An event of interest can be an SLO violation, a device contention, or a flow suspicion. Regarding our example SLOs, end-to-end delay violations are detected when there were high delays during the epoch. In turn, bandwidth violations are detected when the provided bandwidth falls below the SLO-defined value due to packet drops and contentions. We further describe violation detection procedures in Section 4.3.6. Contentions are indicated and detected via field `ContentionPts`. Similarly, suspicion is detected by inspecting field `SuspicionPts` (§4.3.5). At the end of this step, the generated report is sent to the control plane to be analyzed and inform problem diagnosis, as further described next.

4.2.2 INTSIGHT Control Plane

INTSIGHT's control plane application has two main tasks: monitoring coordination and problem diagnosis. The steps related to both tasks are depicted in the lower section of Figure 4.1. We start our overview with the coordination task. For INTSIGHT-enabled forwarding devices to effectively monitor traffic, some parameters and lookup tables need to be configured and populated. INTSIGHT starts by configuring the same epoch length for every device in the network (Step 1a). In Section 4.3.1, we detail the work with epochs. For the moment, it suffices to know that having the same epoch length along with clock synchronization across devices enables INTSIGHT to make temporal correlations between events reported by different devices and flows. For path tracing, data plane devices need to update the telemetry field `PathID` (of packets) based on their forwarding decisions. More specifically, a lookup table is necessary in each device to map forwarding decisions into `PathID` updates. INTSIGHT pre-computes the `PathID` updating rules for every (active) path in the network and populates device lookup tables appropriately (Step 1b). This procedure also generates a decoder to translate `PathID`s to the complete sequence of visited devices, as is explained in further detail in Section 4.3.3.

Another important information that needs to be installed onto devices is contention and suspect thresholds (Step 1c). As previously mentioned, they are used in each switch (as a maximum tolerable value) to *locally* detect contentions and pinpoint suspect flows. Regarding coordination, egress nodes need to be configured with SLO thresholds to know when a violation happened (Step 1d). INTSIGHT first receives service level objective values from the SDN control plane, then translates these values into epoch-wise values, and finally installs them as thresholds onto devices. The procedures to identify contentions, suspect flows, and detect SLO violations are further described in Sections 4.3.5 and 4.3.6.

While data plane devices process and forward packets, the INTSIGHT application continually listens for flow reports coming from these devices and executes the following diagnosing steps. First, it parses the received reports and stores their information in a metadata database (Step 5a). This step persistently saves information, enabling both real-time and historical analysis. The reports are stored in the database considering temporal characteristics, i.e., their epoch. INTSIGHT regularly builds a global view of the network for every epoch and analyzes its state, behavior, and performance looking for events of interest (Step 5b).

When an event of interest (i.e., SLO violation, contention, or suspicion) is de-

tected, INTSIGHT uses the reported information to pinpoint where (i.e., in which devices) such event happened (Step 5c). This step involves translating `PathID`s into the sequence of forwarding devices they represent (using the `PathID` decoder) and correlating those with the points of contention indicated by field `ContentionPts` (a process further detailed in Sections 4.3.3 and 4.3.5). Finally, INTSIGHT diagnoses the event of interest by analyzing the behavior of all flows present in the pinpointed devices and indicating the victims and culprits (Step 5d). This step considers the information stored in the report database and involves inspecting fields such as `SuspicionPts`, `RxPkts`, and `RxBits` (further explained in Sections 4.3.5 and 4.3.6).

4.3 Design and Implementation

INTSIGHT relies on multiple complementary mechanisms and abstractions to detect SLO violations directly in the data plane and efficiently report them to the control plane. In Section 4.2, we provided an overview of how these mechanisms interact and how abstractions are applied. In this section, we individually describe these elements in greater detail.

4.3.1 Correlating Events in Time

One of the main abstractions of INTSIGHT are epochs. The discretization of time into epochs has three purposes. The first and main purpose is to enable INTSIGHT to correlate in time events observed and reported for different flows. Second, it creates the means to monitor bandwidth SLOs, which are based on intervals. Third, it allows summarizing, into a single report, multiple packet-wise observations that were due to a single event of interest, significantly reducing the overhead due to control traffic.

Forwarding devices determine the current epoch with the simple division t/α, where t is the current timestamp according to the clock on the device, and α is the epoch length (e.g., in milliseconds). The effects of time drifting on event correlation are tackled by considering a time interval with earlier and later epochs, as presented in Equation 4.1.

$$\left[\texttt{Epoch} - \left\lceil\frac{\epsilon}{\alpha}\right\rceil, \texttt{EgressEpoch} + \left\lceil\frac{\epsilon}{\alpha}\right\rceil\right] \tag{4.1}$$

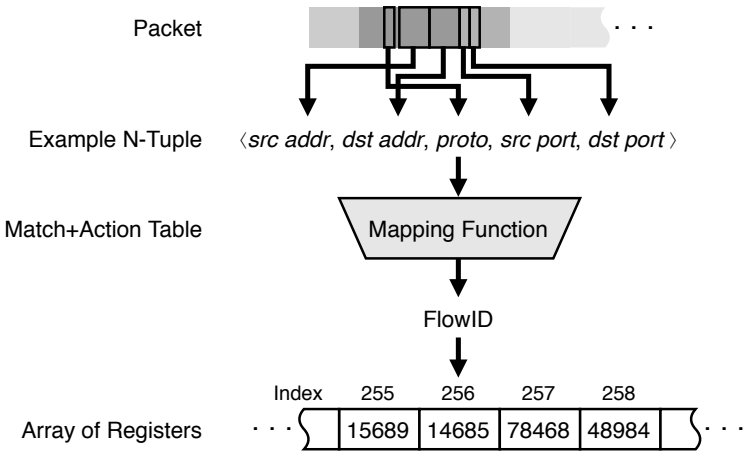
`Epoch` and `EgressEpoch` are the field values reported for a flow packet, and ϵ is the maximum clock divergence (e.g., in milliseconds) devices may have among themselves before being (re)synchronized. Note that since a packet may leave the network on a later epoch (than that of when the packet entered it), recording both the ingress and egress epochs enables INTSIGHT to more precisely identify when packets were in transit.

The appropriate epoch length α for a network depends on factors such as the granularity of the SLOs, the capacity (bandwidth and processing) of links and nodes,, and the control plane capacity. Consequently, a one-size-fits-all approach does not apply. The first two factors usually call for epochs to be shorter, while the last one calls for longer epochs. The longer epochs are, the more traffic oscillations are smoothed out, making detection and diagnosis harder. Conversely, the shorter epochs are, the more fine-grained oscillations become perceptible, making INTSIGHT more sensitive. The goal is to find a compromise with good sensitivity and acceptable reporting costs. In Section 4.6, we detail how each factor may influence the epoch length.

4.3.2 Storing Traffic Metadata Persistently

Existing programmable data plane platforms offer persistent storage in the form of arrays of registers. INTSIGHT uses each register in the array (i.e., an index) to store metadata about a single flow. This is an essential building block for the data plane procedures (§4.2.1) and is implemented by each edge forwarding device. Figure 4.2 exemplifies the actions executed by INTSIGHT to find the appropriate index for each flow.

Figure 4.2: Mapping a traffic flow to a register array index (`FlowID`) for metadata storage and analysis.



An INTSIGHT-programmed forwarding device first extracts (from the packet) the

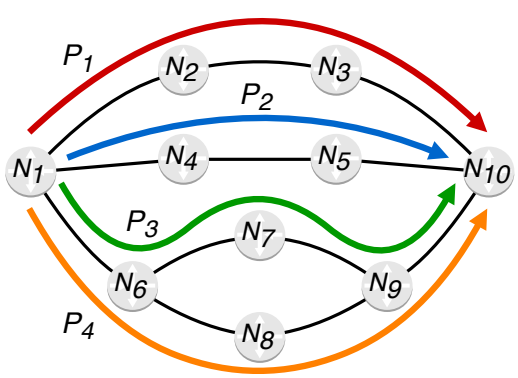
subset of header fields used to identify flows, forming a tuple. In the figure, we consider the case where five fields are used. However, INTSIGHT's design does not impose any restriction on the number and the fields that can be used for flow identification. Right after, INTSIGHT applies a function over the identification tuple to obtain a unique `FlowID` for the flow at that device. In P4, such mapping function can be implemented as a match+action table, where the key is the tuple identification, and the value is the `FlowID` itself. The latter is also the index where metadata for the flow is to be stored. An edge device has an entry in such a table with a locally unique `FlowID` for each flow entering or leaving the network through it. Flows are uniquely identified in the network by the pair $\langle$`FlowID`, *DeviceID*$\rangle$, where *DeviceID* is the unique identifier of the device that generated a given report.

4.3.3 Tracing Packet Paths

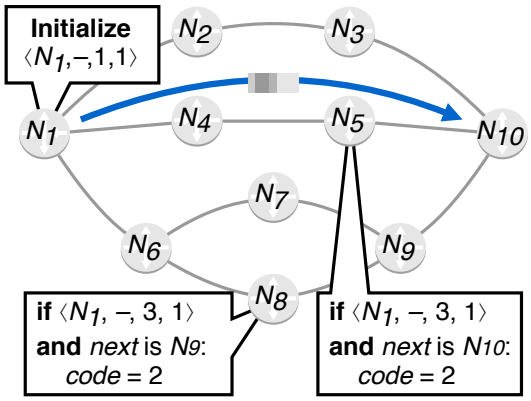
The path traversed by packets is essential information for diagnosing problems, as it allows determining which flows in the network are sharing resources and competing for them. Not surprisingly, path tracing is one of the most important mechanisms of INTSIGHT, which we implement as a coordinated effort between the data and control planes (§4.2.1 and §4.2.2). To avoid the unpredictability (w.r.t. packet space and processing requirements) of appending, to the telemetry header, the identifiers of every device through which a packet traverses, our proposed system uses a single, fixed-size tuple – `PathID` in Table 4.1 – to identify different paths inside the network. `PathID` tuples have the form $\langle$*source*, *destination*, *length*, *code*$\rangle$. Forwarding devices apply if-this-then-that rules (pre-computed in the control plane) to update the `PathID` tuple of an INTSIGHT header so that it may encode the path traversed by the respective packet. For example, consider the network in Figure 4.3a. Packets from N_1 to N_{10} can be routed through four different paths, two of which comprise four hops each (i.e., P_1 and P_2) and the other two five hops each (i.e., P_3 and P_4). Figure 4.3b presents the `PathID` updating rules INTSIGHT installs in forwarding devices to identify the path taken by each packet.

Suppose, as shown in Figure 4.3b, that a packet is routed through Path P_2 (N_1, N_4, N_5, N_{10}). N_1, the ingress node, is responsible for initializing the `PathID`; its initial value is $\langle N_1, –, 1, 1\rangle$. This value indicates that Node N_1 is the source node, the destination (or last hop) is still unknown, the path comprises a single hop (Node N_1 in this case), and the code of the path taken so far is 1 (since there is only a single path from a device to itself).

Figure 4.3: Example network for path tracing method.



(a) Network with four simple paths from N_1 to N_{10}.



(b) Rules to initialize and update the `PathID` tuple.

`PathID`	**Path**	**Sequence of Devices**
$\langle N_1, N_{10}, 4, 1\rangle$	P_1	N_1, N_2, N_3, N_{10}
$\langle N_1, N_{10}, 4, 2\rangle$	P_2	N_1, N_4, N_5, N_{10}
$\langle N_1, N_{10}, 5, 1\rangle$	P_3	$N_1, N_6, N_7, N_9, N_{10}$
$\langle N_1, N_{10}, 5, 2\rangle$	P_4	$N_1, N_6, N_8, N_9, N_{10}$

(c) `PathID` decoder dictionary.

When the packet arrives at N_4, this node increments the `PathID` length field by one and checks for `PathID` code updating rules. Since there are no rules in this node, no further updates are carried out. Next, upon packet receipt, Node N_5 executes the same procedure, first incrementing the length field and then checking for code updating rules. At this point, the value of the `PathID` tuple is $\langle N_1,–,3,1\rangle$. As shown in Figure 4.3b, the rule installed in Node N_5 states that if the current value of the `PathID` tuple is $\langle N_1, N_5, 3, 1\rangle$ and the next hop of a packet is N_{10}, then the `PathID` code should be updated to 2. At the end of the path, Node N_{10}, the egress node, updates the destination and length fields appropriately such that `PathID` becomes $\langle N_1, N_{10}, 4, 2\rangle$. After the update, it extracts the `PathID` tuple from the packet and saves it to memory (along with other flow metadata) until the end of the epoch, when the path is reported to the control plane.

In the control plane, INTSIGHT decodes the `PathID` to the full sequence of devices. For that purpose, it uses a dictionary as the example shown in Figure 4.3c. INTSIGHT's algorithm for generating the `PathID` updating rules and the decoder dictionary is described in detail next.

4.3.4 Encoding Paths

As previously described, INTSIGHT uses `PathID` tuples in the form ⟨*source*, *destination*, *length*, *code*⟩ to identify paths in a network. The first three fields in a `PathID` are natural attributes of every path, i.e., every path has a source and a destination node, as well as a length. For paths with the same values for these attributes, it is necessary to assign a unique code to each. INTSIGHT pre-computes it (in the control plane), together with the accompanying `PathID` code updating rules and a `PathID` decoder dictionary, using Algorithm 3. Our algorithm is partly inspired by the Ball-Larus algorithm (BALL; LARUS, 1996) for efficient path profiling and tracing of computer programs.

The algorithm receives as input a graph G that represents the network topology with a set V of forwarding devices and a set E of links. It outputs the set `Rules` of `PathID` code updating rules to be installed on each device in the network and a `Decoder` that maps `PathID` tuples to device sequences. After variable initializing procedures, in Line 8, for each device in the network, the algorithm gets a list of all paths from it to all other devices according to externally-defined routing policies. We recommend that the set provided to the algorithm includes the paths that could be used for routing traffic during normal operation and upon the presence of link and node failures (considering methods such as ECMP, OSPF Loop-Free Alternates, and BGP Diverse Paths are in place). We emphasize that INTSIGHT does not control how devices forward packets but simply programs forwarding devices to update the `PathID` code of packets according to decisions made by the network-specific routing protocols. While not addressed in this iteration of the work, a human operator could also be included in the loop to refine the possible paths. Hence, whenever a flow is rerouted (e.g., due to failures or re-optimization), INTSIGHT does not need to update existing (or create new) `PathID` updating rules. In the event of nodes or links being added to the network topology, then INTSIGHT requires to compute additional rules for the newly available paths.

For each received path, the algorithm finds the first node not appearing in the same order in the previously analyzed path (Lines 9–16). Right after, two main steps are taken. First, a code is calculated for each remaining hop in the path (Lines 19–24). Second, a new `PathID` updating rule is created for each device with a code different from the code of the subpath from source to that hop (Lines 25–28). Line 34 registers the path with its ID in the `Decoder` dictionary. The time complexity of Algorithm 3 is $\mathcal{O}(p \cdot l)$ in the worst case, where p is the total number of (configured) paths in the network and l is the

Algorithm 3 Path encoding algorithm.

Input: $G(V, E)$ ▷ Network Topology Graph

1: `Rules`$(v) \leftarrow \varnothing, \forall v \in V$ ▷ Path ID Updating Rules
2: `Decoder` $\leftarrow \varnothing$ ▷ ID to Node Sequence Decoder
3: **for** *source* $\in V$ **do** ▷ Source Node
4: *paths* $\leftarrow$ SORTED(GETPATHS(G, s))
5: *prev_path* $\leftarrow$ EMPTYPATH() ▷ Previous Path
6: *last_code* $\leftarrow$ SQUAREMATRIX$(|E|, -1)$ ▷ Next Code
7: *last_code*(*source*, 0) $\leftarrow 0$
8: **for** *path* $\in$ *paths* **do**
9: *prev_node* $\leftarrow$ Null ▷ Previous Node
10: *hop_num* $\leftarrow 0$ ▷ Hop Number in Current Path
11: *node* $\leftarrow$ FIRSTNODE(*path*) ▷ Current Node
12: **while** *hop_num* $<$ LENGTH(*path*)
13: **and** *prev_path*(*hop_num*) $=$ *node* **do**
14: *prev_node* $\leftarrow$ *node*
15: *hop_num* $\leftarrow$ *hop_num* $+ 1$
16: *node* $\leftarrow$ *path*(*hop_num*)
17: *path_code* $\leftarrow 0$ ▷ Code for Path
18: **while** *hop_num* $<$ LENGTH(*path*) **do**
19: *prev_code* $\leftarrow 0$ ▷ Prefix Subpath Code
20: **if** *hop_num* > 0 **then**
21: *prev_code* $\leftarrow$ *last_code*(*node*, *hop_num* $- 1$)
22: *new_code* $\leftarrow$ MAXVALUE(*prev_code*, *last_code*(*node*, *hop_num*) $+1$)
23: *last_code*(*node*, *hop_num*) $\leftarrow$ *new_code*
24: **if** *new_code* $\neq$ *prev_code* **then**
25: Add new rule to `Rules`(*node*):
26: "if `PathID` $= \langle$*source*, –, *hop_num*, *prev_code*$\rangle$
27: and next is *node* then assign *new_code*"
28: *path_code* $\leftarrow$ MAXVALUE(*path_code*, *new_code*)
29: *prev_node* $\leftarrow$ *node*
30: *hop_num* $\leftarrow$ *hop_num* $+1$
31: *node* $\leftarrow$ *path* (*hop_num*)
32: *destination* $\leftarrow$ LASTNODE(*path*) ▷ Destination Node
33: `Decoder`(*source*, *destination*, LENGTH(*path*), *path_code*)

Output: `Rules`, `Decoder`

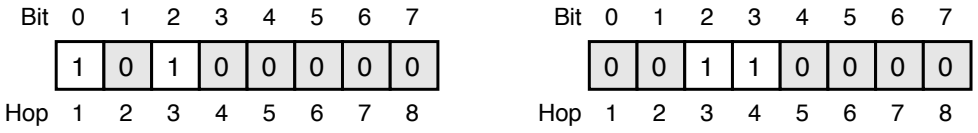
length of the longest path. INTSIGHT uses the outputs of the algorithm for two purposes. The set `Rules` is used to configure forwarding devices in the data plane, enabling them to progressively compute the `PathID` of packets. The `PathID` `Decoder` is stored on the control plane, allowing it to get the full sequence of devices from `PathID`s calculated

for packets.

4.3.5 Pinpointing Contentions and Suspects

Another important task in INTSIGHT's approach that is delegated to the data plane (§4.2.1) is to pinpoint both devices observing contentions and flows with suspiciously high demand. Regarding contentions, as previously described, detection is based on a parameterizable threshold for queue occupancy. This threshold is stored on a lookup table (implemented as a P4 match+action table in each device) and should be set to a value that is a fraction of the queue depth (e.g., one-eighth of the depth (CHEN et al., 2019)) and that represents an undesirably high hop delay in the network, which varies across different network speeds and types (MCKEOWN; APPENZELLER; KESLASSY, 2019). Lower values for this threshold will cause more reports to be generated (i.e., reporting on smaller queue buildups) while higher values cause only more sizable contentions to be reported. When a forwarding device detects a contention in one of its queues, it marks the `ContentionPts` field of the INTSIGHT header (see Table 4.1). The `ContentionPts` field of a packet is a bit array where bit b indicates whether or not the packet observed a contention on the $(b+1)$*th* hop in its path. Figure 4.4a exemplifies a case where a packet observed contentions at its first and third hops as indicated by bits zero and two, respectively. When paired with the sequence of devices of a path, field `ContentionPts` pinpoints the congested hops.

Figure 4.4: Examples field values for pinpointing contentions and suspects.



(a) `ContentionPts` (b) `SuspicionPts`

Suspiciously high demands are also detected based on pre-configured parameterizable thresholds for bitrate and packet rate. These thresholds should be set in proportion to the queue rate that would not be expected to be utilized by a single flow, for example, 50%. These thresholds are stored and queried from a lookup table, implemented as an additional P4 match+action table in each device. It maps each output queue to the respective threshold value that reflects its capacity. When a device receives a packet, it inspects the `RxPkts` and `RxBits` fields. If the value of (at least) one of these fields is higher than the

respective threshold, this means that the flow has already transmitted a significant amount of traffic during the current epoch, potentially causing a contention, and, as such, should be considered a suspect. Field `SuspicionPts` is used to indicate points where a flow is a suspect; its structure is akin to the `ContentionPts` field. Figure 4.4b shows the case where a packet was marked as belonging to a suspect flow in its third and fourth hops. In INTSIGHT's approach, the purpose of identifying *suspect* flows is to provide additional information to the control plane to help find the root cause (in terms of competing traffic) of contentions.

4.3.6 Monitoring SLO Compliance

As previously described in Section 4.2, in this chapter, we focus on monitoring end-to-end delay and provided bandwidth SLOs. INTSIGHT also executes this task entirely in the programmable data plane platform (§4.2.1). The end-to-end delay of a packet inside a network is a factor of the propagation, transmission, queuing, and processing delays. All of these delays are either available as standard metadata in programmable targets or can be accurately estimated considering readily available packet, queue, and link information on devices. To compute a packet end-to-end delay, INTSIGHT programs forwarding devices to progressively increment a path-wise telemetry header field (`E2EDelay`, see Table 4.1) with the locally-observed latencies. At the end of the path, the edge device compares the value stored on the telemetry header against the SLO-defined value, which is queried from a match+action table using the `FlowID` of the packet. If the former is higher, then the device increments by one the `HighDelays` field stored in its memory. This procedure enables edge devices to report when and how many packet-wise delay SLO violations were observed in an epoch.

Algorithm 4 Detecting violations for provided bandwidth.

```
1: if TxBits < SLO Bandwidth then
2:   if (RxPkts − TxPkts) > 0 and ContentionPts > 0 then
3:     SLO was violated in the last epoch
4:   else
5:     Demand was too low to allow meeting the SLO
6: else
7:   SLO has been met during the last epoch
```

For SLOs related to provided bandwidth, an essential indicator to monitor is the

number of successfully transmitted bits by (and out of) the network. A violation happens when the provided bandwidth falls below the agreed amount due to contentions and packet drops inside the network. INTSIGHT uses Algorithm 4 at the end of each epoch in order to detect bandwidth violations. `TxBits` represents the bandwidth provided to a flow (which is computed by its egress node when forwarding its packets), term ($\texttt{RxPkts} - \texttt{TxPkts}$) represents the number of packet drops, and condition $\texttt{ContentionPts} > 0$ checks if the flow observed any contention. One important detail is that SLO-defined provided bandwidth values are typically defined in bits per second (e.g., Mbps), while epochs have sub-second length. A per-second requirement can be translated into a per-epoch value by multiplying the original value by the epoch length in seconds. SLO-defined values for bandwidth are stored and queried from a match+action table using the `FlowID` associated with the traffic flow.

4.4 Evaluation

We evaluated INTSIGHT considering two complementary sets of experiments. The first set demonstrates, by way of two use cases, the effectiveness of INTSIGHT's approach in detecting and diagnosing SLO violations (Section 4.4.2). The second set compares INTSIGHT's resource usage to state-of-the-art monitoring and debugging approaches (Section 4.4.3). Next, we describe the experimental setup used for the evaluation.

4.4.1 Experimental Setup

Prototypes

We developed two prototypes of INTSIGHT, for the reference P4 software switch (BMv2 – behavioral model version 2 (P4 LANGUAGEM CONSORTIUM, 2014)) and the NetFPGA-SUME development board (ZILBERMAN et al., 2014). Our prototypes are available at (MARQUES; LEVCHENKO; GASPARY, 2020b). In our functional evaluation, we focus on using the BMv2 prototype (as described next). For the performance evaluation, we focus on the NetFPGA prototype.

Testbed

We used as testbed for the functional evaluation Mininet-emulated networks running on a dedicated Linux 4.4 server with 2x Intel Xeon Silver 4208 $2.1\,\mathrm{GHz}$ 8-core 16-thread processors, 8x $16\,\mathrm{GB}$ $2400\,\mathrm{MHz}$ RAM, and $2\,\mathrm{TB}$ of NVMe SSD storage. We carried out a preliminary evaluation on the testbed to identify capacity limits that guarantee consistent performance throughout the experiments. Considering the obtained results, we chose to set forwarding device queue rates to $9000\,\mathrm{pps}$, and queue depth to $1125\,\mathrm{packets}$, one-eighth of the queue rate. Link capacity was set to $100\,\mathrm{Mbps}$. Network traffic was generated using tcpreplay and consisted of synthetic UDP traces. The performance evaluation was based on analytical modeling and the NetFPGA-SUME board.

Scenarios

The scenarios considered for the functional evaluation, including the network topologies and the traffic between endpoints, are explained in more detail later, in each use case. For the comparisons of the performance evaluation, we used the REPETITA dataset (GAY; SCHAUS; VISSICCHIO, 2017), which consists of more than 260 WAN topologies and demand matrices. We evaluated each monitoring approach considering all available networks. In the interest of clarity, for this discussion, we selected six of the most representative networks (the full results, considering all 260 networks, is available (MARQUES; LEVCHENKO; GASPARY, 2020b)). Table 4.2 presents the number of nodes and links, the total packet rate, and the average path length for each of the selected networks. For this evaluation, we assumed approaches may need to monitor up to 512 different (aggregate) flows (i.e., classes of traffic) for each pair of source and destination forwarding devices (since this information is not available in the dataset), which amount to about 50 million different flows in the Sprintlink network (the largest of the selected topologies).

INTSIGHT

For the functional evaluation, INTSIGHT was configured with an epoch length of $2^{16} = 65{,}536\,\mathrm{microseconds}$ (≈15 epochs per second). The contention threshold was set to $140\,\mathrm{packets}$ (i.e., about one eighth of the queue depth, based on (CHEN et al., 2019)). The threshold for marking traffic as suspicious for causing contentions was set to 50% of the capacity of devices and links. For the performance evaluation, we considered an

Table 4.2: Metadata for the network topologies considered on the efficiency evaluation.

Network	Label	Nodes	Links	Total Demand (pps)	Avg. Path Length
Bell Canada	BC	48	130	831,790	5.3
US Signal	US	61	158	1,493,550	6.0
VTLWavenet	VW	92	192	994,565	13.1
TATA	TT	145	388	1,849,113	9.9
Cogent	CG	197	490	1,602,675	10.5
Sprintlink	SL	315	1944	19,253,769	4.0

epoch length of $10\,\mathrm{ms}$[4] (100 epochs per second).

Adaptating Existing Approaches

To enable the comparison to INTSIGHT, considering our problem constraints (i.e., no modification to end-hosts), we made the following adaptations to TPP (JEYAKUMAR et al., 2014) and SwitchPointer (TAMMANA; AGARWAL; LEE, 2018). We adapted TPP to extract the collected telemetry information at the end of a packet's path and send it directly to an analyzer server. Similarly, we adapted Swithpointer's approach by offloading the summarization task to the data plane and leaving the telemetry data analysis to the control plane. Forwarding devices, in this adapted SwitchPointer, report their summarized data at the end of each epoch. We refer to these adaptations as A-TPP and A-SwP, respectively. We emphasize that we have adapted the TPP and SwitchPointer approaches to our problem constraints because both were designed for a different problem setting. As such, our evaluation should not be viewed as a criticism of these algorithms. We include these modified variants to show that a straightforward extension of existing approaches to our problem setting offers room for improvement, as realized by INTSIGHT.

4.4.2 Functional Evaluation

In this section, we consider two use cases, one for the end-to-end delay and one for the bandwidth SLO. With these, we were interested in assessing the *functionality* of INTSIGHT and not its performance when executed on a large scale.

[4]Epoch lengths were selected considering the guidelines in Section 4.6.

End-to-End Delay

Consider the scenario presented in Figure 4.5a. The traffic between endpoints A and H (Red flow), which has an associated end-to-end delay SLO of $20\,\mathrm{ms}$, has its packets delayed by the bursty traffic from endpoint E to G. We carried out a measurement experiment lasting $60\,\mathrm{seconds}$ to reproduce this scenario. The flow from E to G changed its packet rate to about $6500\,\mathrm{pps}$ for $100\,\mathrm{ms}$ around the 30th second of the experiment.

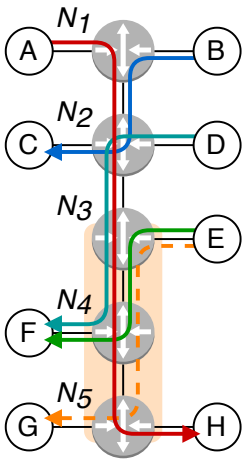
Figure 4.5b presents the reports generated by the forwarding devices during the experiment. Some report fields are omitted for space. From the first two rows in the table, we observe that INTSIGHT reported the end-to-end delay as violated for packets from the Red flow that traversed the network between epochs 464 and 466. During these epochs, the Red flow witnessed a contention in Node N_3. From the remaining rows in Figure 4.5b, we observe that other three flows (i.e., Orange, Green, and Teal) also witnessed a contention in Node N_3 during the same epochs (and in epoch 463). No report was generated for the Blue flow. Furthermore, the Orange flow was marked as a suspect by all nodes in its path (i.e., N_3, N_4, and N_5) in the epochs leading up to the one where only contention was observed. Note that the egress epochs (i.e., 465 and 466) match those for which the Red flow was severely delayed.

Figure 4.5c shows the arrival rate of traffic at Node N_3 in the moments before and after the reported violations and contentions. Considering the data shown in the figure, plotted by the control plane application based on the reports, INTSIGHT was able to observe the abnormal spike in the rate of the Orange flow, which caused the accumulated rate to grow above the link transmission capacity. This unexpected short-lived event caused buffering up on the output port of the forwarding device. Consequently, INTSIGHT was able to diagnose the SLO violation as being caused by the Orange flow.

Provided Bandwidth

In our second use case, we considered the scenario shown in Figure 4.6a, in which the Red flow between endpoints A and H had an associated provided bandwidth SLO of $25\,\mathrm{Mbps}$. It had a substantial number of its packets dropped due to a concurrent increase in demand by other flows sharing one of the nodes in the path. Similarly to the previous study, we carried out a measurement experiment lasting 60 seconds to reproduce this scenario. The Red flow had a constant rate of $30\,\mathrm{Mbps}$ throughout the whole experiment. The Blue flow also had a constant rate of $30\,\mathrm{Mbps}$. The remaining flows (Orange, Green,

Figure 4.5: End-to-End Delay Case Study. A delay-sensitive flow A–H has its SLO violated due to bursty flow E–G.

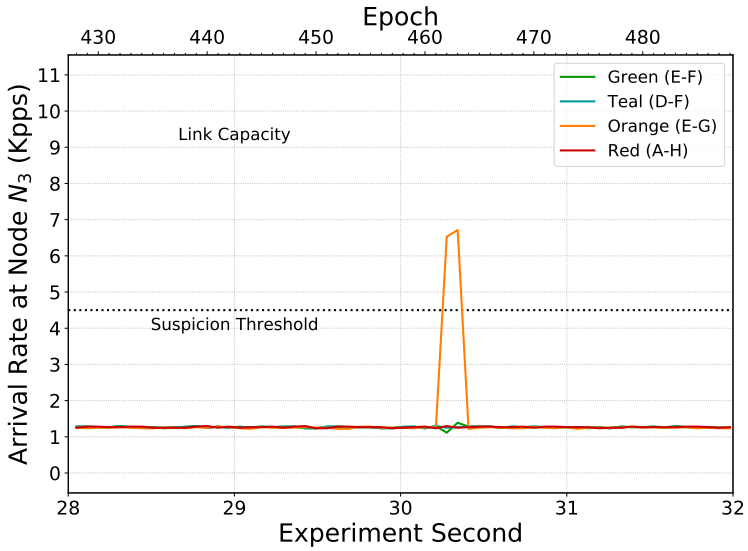


(a) Network topology and traffic routing for the case study.

Traffic	Epoch	Egress Epoch	CPs*	SPs*	High Delays
Red	464	465	N_3	-	39
Red	465	466	N_3	-	19
Orange	462	464	-	N_3, N_4, N_5	-
Orange	463	465	N_3	N_3, N_4, N_5	-
Orange	464	466	N_3	-	-
Green	463	465	N_3	-	-
Green	464	466	N_3	-	-
Teal	463	465	N_3	-	-
Teal	464	466	N_3	-	-

*CPs = Contention Points, SPs = Suspicion Points

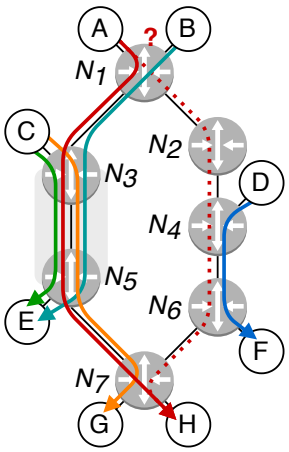
(b) Reports generated by IntSight for all traffic flows in the network. Four traffic aggregates witnessed a contention in Node N_3. The Orange traffic was marked as suspiciously high demanding.



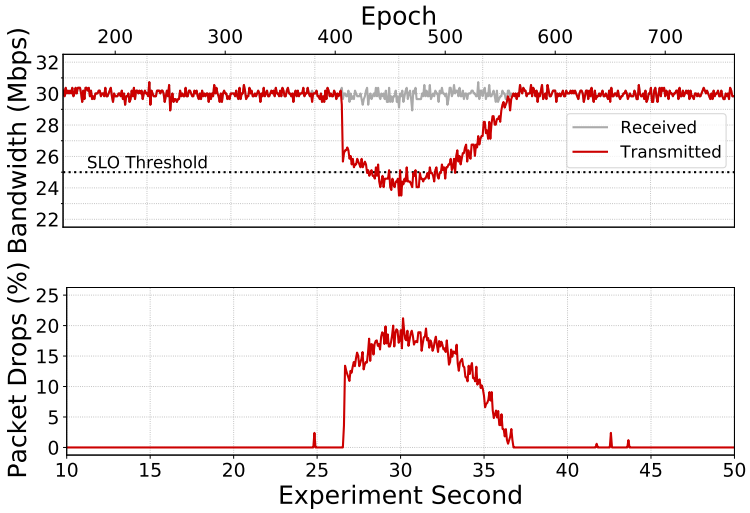
(c) Packet arrival rate for traffic being forwarded by Node N_3. A significant spike in demand is observed for the Orange traffic around the 30th second of the experiment.

and Teal) had each a specific initial rate of 10 Mbps, 15 Mbps, and 20 Mbps, respectively. Throughout the experiment, these flows progressively increased their demand, reaching 30 Mbps at about 30 seconds in the experiment, and then progressively decreased their demand back to the respective initial rates. Since all (but the Blue) flows converged in Node N_3 (directed to Node N_5) on a link with a nominal capacity equal to 100 Mbps, contentions and packet drops were expected.

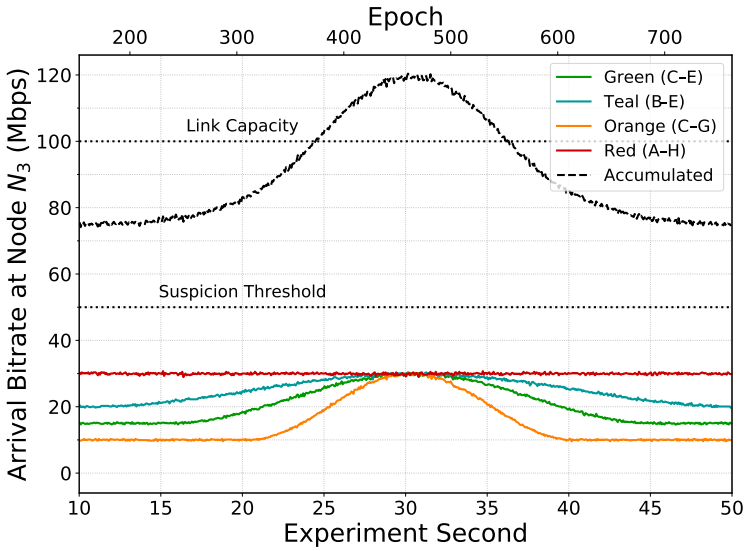
Figure 4.6: Bandwidth Case Study. A bandwidth-sensitive flow A–H suffers performance degradation from recurrent natural increases in demand for other traffic sharing link $N_3 \rightarrow N_5$.



(a) Network topology and traffic routing for the case study.



(b) Bandwidth and percentage of packet drops for Red Traffic. A substantial drop in bandwidth characterizing a violation is observed from the 27th to 37th second of the experiment.



(c) Arrival rate (in Mb/s) for traffic being forwarded by Node N_3. All flows present increases in demand resulting in the saturation of the outgoing link connected to the node.

Figure 4.6b presents the demanded and provided bandwidth of the Red flow and the percentage of dropped packets for each epoch. The figure shows a decrease in the provided bandwidth of up to $5\,\mathrm{Mbps}$ starting around the 27th second of the experiment. Furthermore, at around the 30th second of the experiment, the provided bandwidth falls below the SLO threshold of $25\,\mathrm{Mbps}$. The stable curve for demanded bandwidth along

with the accompanying curve for packet drops indicate that the decrease is not due to a decline in demand, but to a contention inside the network. This characterizes a provided bandwidth SLO violation, and, for such, INTSIGHT generated 59 reports, one for each epoch while the violation remained. All reports for the Red flow indicated a contention in Node N_3, and other three flows (Orange, Green, and Teal) witnessed a contention on the same node.

Figure 4.6c shows the packet arrival rate of traffic at Node N_3 during the experiment, generated by the control plane application based on the received reports. INTSIGHT was able to observe that all traffic was well behaved – as there was no individual flow above the suspicion threshold – but that their nearly simultaneous increase in demand exceeded the capacity of the outgoing link. Consequently, INTSIGHT was able to correctly diagnose the SLO violation as caused by typical demand fluctuations, and could suggest moving the Red traffic to the alternative path $(N_1, N_2, N_4, N_6, N_7)$.

4.4.3 Performance Evaluation

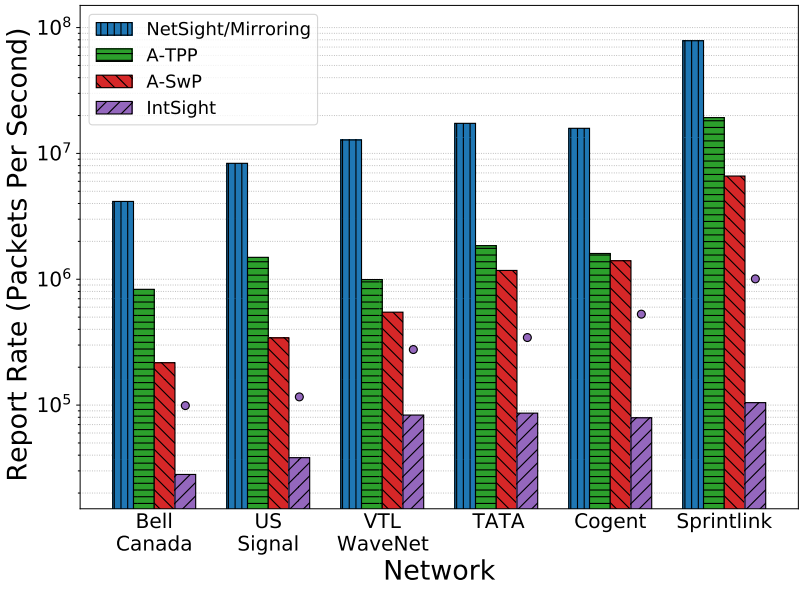
One of the main concerns when monitoring a system is minimizing the overhead imposed on it. In INT-based network monitoring, this process means a sensible use of four main resources: (a) network bandwidth, (b) memory space, (c) header space, and (d) computation time. In this part of the evaluation, we analyze the use of network resources by INTSIGHT and compare it, when appropriate, to existing monitoring approaches.

Network Bandwidth

Figure 4.7 presents the report rate of each approach, i.e., the number of monitoring report packets sent out each second. INTSIGHT[5] outperforms existing approaches by generating up to two orders of magnitude fewer reports (order of thousand packets per second) than Adapted SwitchPointer (A-SwP – i.e., the best among the contenders). Mirroring-based systems generate up to one order of magnitude more packets than the production traffic (one per hop of each production packet), making them impractical to monitor all traffic in the network. Adapted TPP (A-TPP) generates one report for each production packet, limiting its use to smaller networks. SwitchPointer achieves comparatively lower rates by reporting only once per epoch per monitored flow, but still reaches

[5]In the graph, we show the average (bar) and maximum (circle) number of reports that would be generated for an event of interest lasting for a second on any single device.

Figure 4.7: Report rate (packets per second).



the order of a tenth of millions of packets per second due to lacking a mechanism to filter reports in the data plane. INTSIGHT generates even less reports because it reports only when events of interest are observed and once per epoch per flow.

Device Memory

Table 4.3 compares the memory requirements of A-SwP and INTSIGHT (mirroring and A-TPP require a fixed amount of memory). We observe that A-SwP requires up to 4 times as much memory as INTSIGHT. SRAM is mainly used to store telemetry data. The difference comes mainly from the fact that A-SwP stores paths as a sequence of node IDs, while INTSIGHT only uses the small, fixed-size `PathID` tuple to trace paths. The remainder is used to hold SLO, contention, and suspicion thresholds as well as `FlowID` and `PathID` lookup tables. A-SwP and INTSIGHT consume the same amount of TCAM, as this resource is used by both to extend forwarding tables with information to distinguish flows. Thus, INTSIGHT's memory requirements are well within the available resources on current P4 programmable hardware[6].

[6]For example, the Barefoot Tofino has 370 Mb of SRAM and 40 Mb of TCAM (BAREFOOT NETWORKS, 2020; BOSSHART et al., 2013). The NetFPGA-SUME board has 216 Mb of SRAM (ZILBERMAN et al., 2014).

Table 4.3: Device memory usage (Mb).

Approach	**A-SwP**		**IntSight**	
Network	**SRAM**	**TCAM**	**SRAM**	**TCAM**
Bell Canada	30.42	0.29	12.78	0.29
US Signal	40.80	0.37	16.41	0.37
VTL WaveNet	112.85	0.65	26.84	0.65
TATA	164.86	1.18	42.78	1.18
Cogent	224.39	1.61	58.23	1.61
Sprintlink	175.25	2.89	89.53	2.89

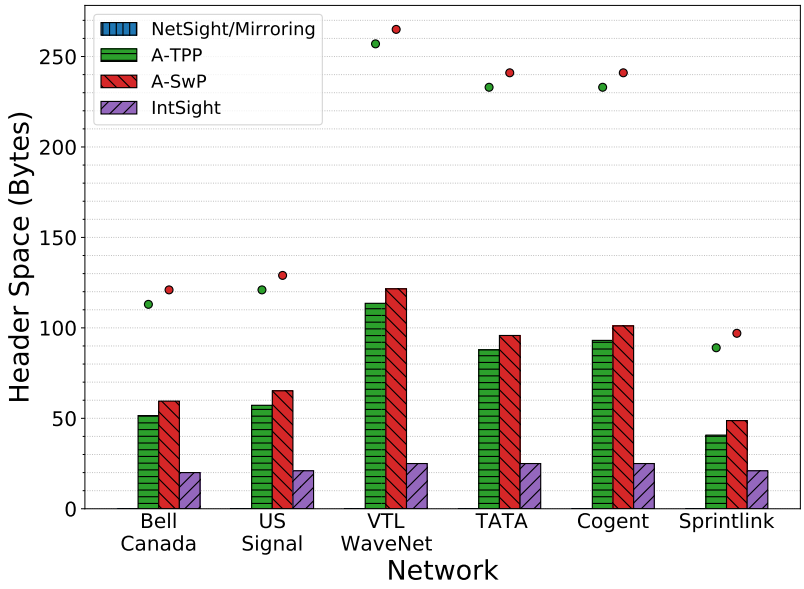
Header Space

We also evaluated how much header space is required to be reserved from the network MTU to allow packets to carry telemetry data. Figure 4.8 presents the header space demanded by each evaluated approach. NetSight and mirroring-based approaches do not use any header space from production packets. For A-TPP and A-SwP, the bar value shows the mean space required by all network packets instrumented with an INT header, while the circle indicates the maximum value. The header space usage by both approaches is proportional to the network diameter, and requires reserving up to $265\,\text{bytes}$ for telemetry data, what would amount to a $17.67\,\%$ decrease in the maximum goodput for the production traffic. Conversely, INTSIGHT adjusts its header to tailor each network, i.e., its header has a fixed-size for all flows in a network. For example, INTSIGHT sets the length of the `ContentionPts` field to the length of the longest path used in the network, and that of the `PathID` source field to $\lceil \log_2 \text{n} \rceil$, where n is the number of nodes in the network (since a unique ID needs to be assigned to each node). The maximum required size for the INTSIGHT header for all evaluated networks was $25\,\text{bytes}$, only $1.67\,\%$ of an Ethernet MTU.

Programmable Compute Resources

The computational resource requirements of INTSIGHT depend on the target platforms. Perhaps the most demanding are platforms with fixed resources, namely FPGA and ASIC switch implementations. To evaluate resource usage on such a fixed-resource target, we used our prototype for the NetFPGA-SUME development board (ZILBERMAN et al., 2014). Our prototype required $63{,}260$ look-up tables (LUTs) and $136{,}334$ registers. These numbers represent, respectively, $14.6\,\%$ and $15.7\,\%$ of the available resources, which we

Figure 4.8: Header space usage (bytes).



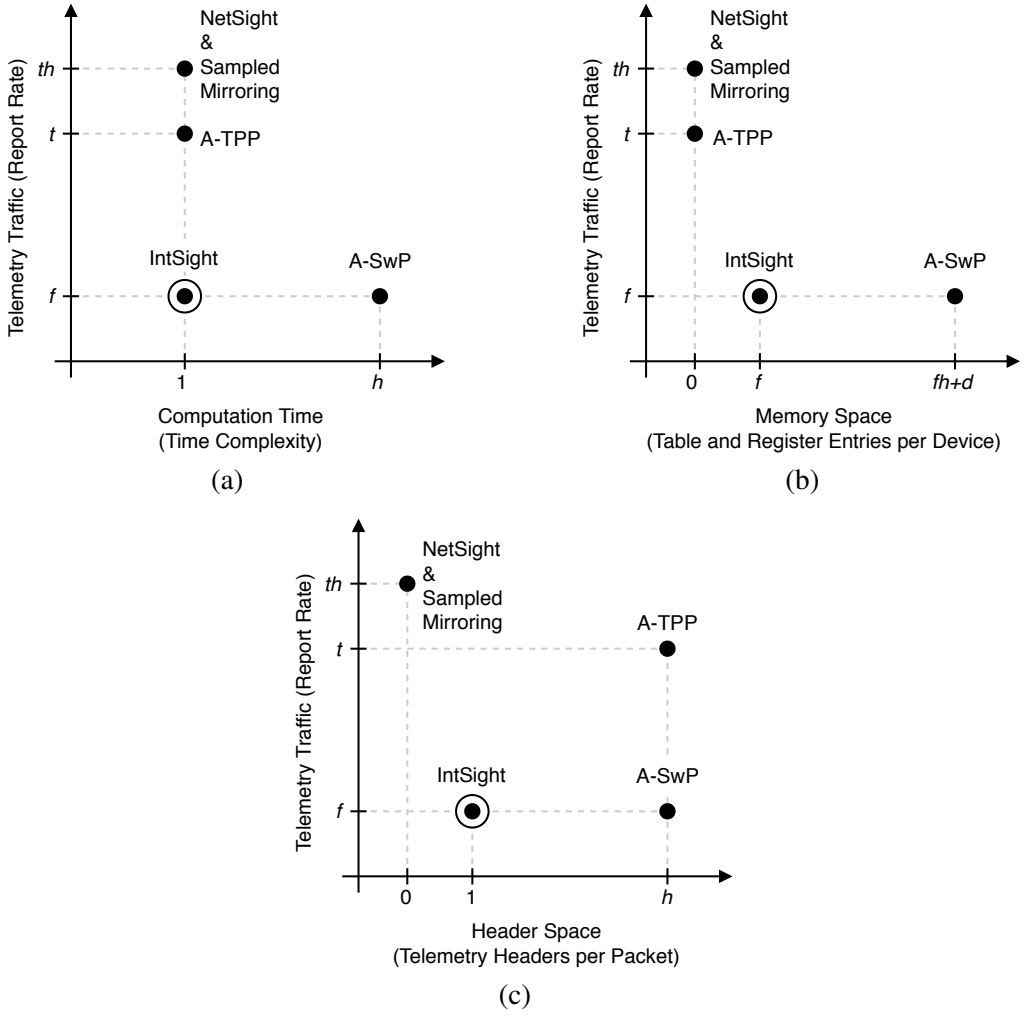
deem a modest requirement. We expect comparable processing resource requirements for similar targets.

Asymptotic Bounds for Resource Usage

To conclude this section, we analyze the trade-offs between the main resources by discussing the asymptotic bounds for their usage by each monitoring approach. Figure 4.9 compares INTSIGHT and the most representative approaches to network performance monitoring. Each graph in the figure presents the asymptotic amount of telemetry traffic generated as a function of (a) computation time, (b) memory space, and (c) header space. In the figure, d and f are, respectively, the number of forwarding devices and active flows in the network, t is the average total network throughput (pps), and h is the average path length between end-points. All values represent the asymptotic exact bound Θ. Figure 4.9a shows that Adapted SwitchPointer (A-SwP) involves processing (originally implemented on end-hosts) that is linear to the length of packet paths, making it challenging to implement even on novel programmable data plane platforms. All remaining techniques, including INTSIGHT, require constant-time processing. Figures 4.9b and 4.9c show that INTSIGHT, by carefully using both memory and header space, generates considerably less telemetry traffic than NetSight and Sampled Mirroring (mirroring-based techniques) and also Adapted TPP (A-TPP, which only uses header space). Compared to A-SwP, INTSIGHT requires less memory and header space to achieve the same telemetry

traffic rate.

Figure 4.9: Comparison of monitoring techniques considering generated telemetry traffic as a function of network resource usage.



4.5 Related Work

In this section, we present existing approaches that target at helping detect and diagnose network problems, and contrast them with INTSIGHT.

Traditional Tools and Techniques

Traditional passive monitoring tools, such as SNMP (CASE et al., 1989) and NetFlow/IPFIX (CISCO, 2005; CLAISE; TRAMMELL; AITKEN, 2013), lack the appropriate level of detail to detect and diagnose problems such as those described in our motivation Section 4.1. These tools are limited to providing traffic counters aggregated over

coarse timescales (in the order of dozens of seconds or higher). Furthermore, they do not monitor statistics crucial to detect and diagnose SLO violations, such as delays. In contrast, active measurement techniques (e.g., ping, traceroute, OWAMP (SHALUNOV et al., 2006), and TWAMP (HEDAYAT et al., 2008)) can be used to estimate the mentioned statistics. However, they are limited by the fact that the network may not necessarily route and prioritize their probes in the same way as the traffic of interest. Moreover, achieving granularity close to per-packet for these techniques requires injecting substantial measurement probes into the network.

Measurement Based on Packet Mirroring

Recent research (HANDIGOL et al., 2014; RASLEY et al., 2014; ZHU et al., 2015; TILMANS et al., 2018) explored the use of packet mirroring to estimate delays and trace paths of production packets. Handigol *et al.* (HANDIGOL et al., 2014) proposed NetSight, a monitoring system that has mirroring as its fundamental measurement mechanism. NetSight configures the network so that every forwarding device in the path of a flow creates payload-stripped copies (called postcards) of the packets it processes and sends them to analysis equipment. All of the postcards generated for a single packet are combined by the analysis servers to form a packet history reporting network state, behavior, and performance as observed by the packet.

The main challenge when using NetSight is that meeting the requirements introduced in Section 4.1 would involve generating postcards on every forwarding device for every packet. Except for small networks, the monitoring load added to the control plane (towards analysis equipment) would be prohibitively high, inhibiting timely detection, diagnosis, and, consequently, troubleshooting. To address the challenge faced by NetSight, subsequent techniques also based on packet mirroring (RASLEY et al., 2014; ZHU et al., 2015; TILMANS et al., 2018) rely on sampling methods to alleviate the monitoring overhead w.r.t. both bandwidth and processing. Since sampling inherently misses events, these techniques have as their challenge keeping monitoring granularity and accuracy high. For example, if a packet is not sampled, then any problem it has witnessed will not be detected (or diagnosed).

Monitoring Assisted by End-Hosts

TPP (JEYAKUMAR et al., 2014) and SwitchPointer (TAMMANA; AGARWAL; LEE, 2018), techniques proposed in the context of data center networks, investigated involving end-hosts in monitoring tasks such as analysis and storage of measurement data. Since hosts lack visibility into the network, to pinpoint and diagnose problems, these techniques build upon the concept of in-band network telemetry (INT) (KIM et al., 2015). They instruct forwarding devices to annotate production packets with metadata such as device IDs, queue IDs, processing delays, and matched forwarding rules. The metadata contained in the packets is then used by the end-hosts to detect problems with their flows. The detection then triggers the exchange of information between end-hosts to diagnose the problem.

As mentioned, these techniques were designed for data center networks, and within this setting, they can be used to detect and diagnose the SLO violations we discussed in Section 4.1. In our work, we target not only those but also enterprise and internet service provider networks. We assume operators either do not have control or do not want to modify the networking stack of end-hosts. Adapting TPP and SwitchPointer to this requirement is not straightforward. Furthermore, even when adapted (as we exercised in Section 4.4), they do not fully exploit the capabilities and advantages possible with programmable data planes, i.e., they are not able to detect SLO violations directly in the data plane and selectively report information to the control plane to minimize overheads.

4.6 Additional Remarks

To close our presentation of INTSIGHT, we discuss a few aspects that can be potentially revisited and expanded upon for future work.

Selecting an Epoch Length

Since there is no silver bullet for epoch length, as mentioned in Section 4.3.1, in this section, we aim at providing some guidance regarding the selection of an appropriate α value for a network. Three main factors should be considered when selecting the epoch length. Next, we describe each one of them. The first is related to the *granularity of SLOs*. Along with a target value for a performance metric of interest, SLOs also establish how often this target should be met (or, for how long it is reasonable it does not). For example,

consider the following SLO (also presented in the introduction): "Provided bandwidth has to be higher than $1\,\mathrm{Gbps}$ at least 99% of the time (within a minute) for a video streaming traffic aggregate". This indicates that the provided bandwidth can fall below the target value for 1% of one minute with no penalty to the network provider. This percentage of time imposes an upper bound on the epoch length, $0.6\,\mathrm{seconds}$ in the example. As the specified SLOs become stricter (requiring more nines), the upper bound becomes smaller. The upper bound is given by $\alpha \leq t \cdot (1 - p)$, where t is the SLO interval (e.g., one minute) and p is the percentage (e.g., 99% or 0.99). Given the upper bound, in practice, the network operator may want to select an epoch length that is a fraction of it so that INTSIGHT has time to react to any problems after an epoch-wise violation report (and before a penalty-inducing violation happens).

The second factor that needs to be considered when selecting the epoch length is the *bandwidth and processing capacity of links and nodes*. As previously described in Section 4.3.1, discretizing time into epochs enables INTSIGHT to correlate events in time and view the network and its events through the lens of a group of packets. The main goal is to diagnose congestions (i.e., pinpoint congested devices and identify the competing flows). The faster a network is, the more short-lived congestions can be, and, as a result, the shorter epochs may need to be to zero-in on the culprits of these congestions. Naturally, at some point, making the epoch shorter on a given network may not lead to significant improvements (while also risking saturating control plane analyzer servers; see the next topic). With this rationale in mind, we propose that the epoch length be at least smaller than the typical time necessary to transmit a full device buffer. For example, a $12.5\,\mathrm{MB}$ buffer transmitting via a $10\,\mathrm{Gbps}$ link would suggest an epoch length of at most 10 milliseconds. Buffer size choices and implications are extensively discussed in (MCKEOWN; APPENZELLER; KESLASSY, 2019).

The third, and last, factor involves *control plane resources and capabilities*. The control plane capacity, i.e., the bandwidth of links connecting data and control planes and the report processing capacity of analyzer servers, contrasts with the other factors since it establishes a lower bound on the epoch length. For example, considering INTSIGHT's reports have the minimum frame size of 64 bytes, a $1\,\mathrm{Gbps}$ connection between each device and the control plane would enable at most about two million reports to be transmitted within a given second. This suggests that, for the given example, the epoch length should not be less than $\frac{R}{B} = \frac{64\,\mathrm{B}}{1\,\mathrm{Gbps}} = 512$ nanoseconds, where R is the report length and B is the data-to-control plane bandwidth. Regarding the processing capacity of analyzer servers,

for contention diagnosis to be in real-time, no more reports should be sent in a second than the number the server can process. For example, on a small 25-node network, with a single $50\,\mathrm{Kpps}$-capable server dedicated to monitoring, and considering the worst-case scenario where all nodes are generating reports, the epoch length should be no less than $\frac{n}{m \cdot C} = \frac{25}{1 \cdot 50\,\mathrm{Kpps}} = 500$ microseconds, where n is the number of nodes in the topology, m, the number of monitoring servers, and C, the unit processing capacity of each server.

Pinpointing Contentions and Suspects with Thresholds

INTSIGHT currently uses thresholds to pinpoint device contentions and suspect flows. This approach brings with it the challenge of setting the correct values to adequately detect and report events of interest when they happen. A threshold is a logical fit for detecting contentions since they consist of queue buildups in devices. However, for suspicion, using a threshold on the traffic demand may not be enough to tell the whole story. We consider INTSIGHT's method for pinpointing suspect flows could be improved by integrating a mechanism such as ConQuest (CHEN et al., 2019), which enables programmable devices to track how much of the queue occupancy is due to each different flow being buffered on it. A more sophisticated approach to suspicion would consider those flows representing a significant fraction of the packets in a congested queue. In practice, these thresholds should be set empirically based on the desired volume of reporting traffic.

Fine-grained SLO Tracking and Violation Diagnosis

As previously presented in Section 4.3.2, INTSIGHT's design enables traffic flows to be defined as any set of packets sharing the same values for an arbitrary subset of header fields. In theory, this approach allows monitoring flows with very fine granularity (e.g., TCP sessions). Nevertheless, currently existing forwarding devices do not have sufficient memory (i.e., SRAM) to store metadata about *all* application-level flows traversing them, especially in a backbone network, for example. To manage memory usage, we recommend configuring INTSIGHT using the following method: discretize SLO-subject traffic using high levels of detail (e.g., 5-tuple) while discerning between the remaining traffic using broader definitions (e.g., UDP traffic from a source to a destination node). This method allows precisely detecting SLO violations while pinpointing culprits with a (potentially) coarser granularity. After initial detection, the network operator may install

additional finer-grained discretizing rules to zoom in on the culprit traffic and determine more specific culprit flows. The limitation of this method is that installing additional rules requires interaction with the control plane. Depending on how short-lived a problem is, rules may not be installed in time to pinpoint the specific culprit. Nonetheless, in the general case, we expect traffic to have its performance impacted by higher priority flows, which would already be identified with fine granularity.

Dealing with Packet Drops and Out-of-order Arrival at Egress Nodes

When packets are dropped because of a link or node failure, INTSIGHT could mistake it for a critical contention (although if that were the case, we would still expect that some of the surviving packets would signal a contention). Nevertheless, upon a failure, we expect the control plane operating system to communicate it to applications such as INTSIGHT, which can use this extra information when diagnosing performance-impacting events. Regarding out-of-order packet arrival, it can happen within an epoch or across epochs. To deal with the first case, INTSIGHT updates registers using commutative operations (addition, bitwise OR, max), so that the order of packets does not change the result. For the second case, nodes can be optionally configured to generate an exceptional report when an old packet arrives after a new epoch has started. In the control plane, INTSIGHT then uses this extra report to update the information previously reported for the epoch.

Identifying the Next Hop of a Packet

As described in Section 4.3.3, INTSIGHT does not decide the next hop of packets. It simply reads the device's metadata field that indicates the output port (chosen by normal forwarding logic) of a packet and updates the `PathID` field accordingly. For targets supporting link aggregation groups (LAGs), INTSIGHT can discern between links depending on the level of abstraction of the target's output port metadata field. For example, in the P4 Portable Switch Architecture (P4 ARCHITECTURE WORKING GROUP, 2018), the output port metadata field indicates the actual device port, even if the forwarding decision for the packet was made considering a LAG. Similarly, when a fast-failover mechanism picks an alternate output port for packets, INTSIGHT is able to record the new path and trigger a report for path change (as mentioned in Section 4.2.1).

Deploying INTSIGHT *Gradually*

For the design of INTSIGHT, we assumed all network devices are equipped with monitoring capabilities (e.g., through data plane programmability). Its approach leverages the visibility into network operation offered by in-band network telemetry to detect and diagnose performance problems accurately. In this context, legacy devices would represent grey boxes, and their operation would have to be estimated by the monitoring-capable devices. This approach translates into correspondingly less accurate data being made available to INTSIGHT, making both detection and diagnosis more challenging tasks. A detailed analysis of the partial deployment of INTSIGHT (and its effects on monitoring coverage) is left as future work.

4.7 Chapter Summary

In this chapter, we proposed INTSIGHT, a system that explores in-band network telemetry to monitor SLO compliance. End-to-end delay and bandwidth SLO violations are detected directly in the data plane, whereas diagnosis (i.e., when, where, and affected/offender flows) occurs in the control plane, using reports sent by the former. As part of this work, we have introduced an in-network, distributed path-aware mechanism for monitoring network traffic capable of fine-grained, highly-accurate, and timely detection and diagnosis of problems impacting performance. We also designed and implemented efficient data plane procedures for gradually computing path-wise metadata such as path IDs, contention points, and end-to-end delays. Finally, we have demonstrated the benefits (regarding functionality, performance, and resource footprint) of path-wise in-band network telemetry compared to state-of-the-art approaches, considering six representative network topologies and existing programmable data plane platforms. We understand our work as an important contribution that can be positioned in the intersection of (*i*) down to packet-level metadata SLO monitoring, (*ii*) modest resource footprint, and (*iii*) noninterference of end-hosts in monitoring tasks.

5 RESPONDING TO NETWORK FAILURES AT DATA-PLANE SPEEDS

Following our work on INTSIGHT, in this chapter, we present another step taken towards answering Question 3 described in Section 1.2, i.e., we further investigate offloading part of the analysis and reaction logic to the data plane. We shift our focus to network equipment failures and investigate ways in which data plane programmability can improve how networks recover from such events. We propose an approach to failure recovery that explores the question of where the line should be drawn to divide responsibilities between the control and the data planes. We start this chapter by introducing, in Section 5.1, the state-of-the-art in failure recovery and the drawbacks of existing approaches. In Section 5.2, we present an overview of FELIX, our proposed system for network rerouting to bypass failures. In Section 5.3, we describe the design and implementation of FELIX. In Section 5.4, we present our evaluation, where we test the effectiveness and overheads of FELIX's approach and compare it with state-of-the-art approaches. In Section 5.5, we describe the related work. In Sections 5.6 and5.7, we discuss limitations as well as possible extensions to FELIX. We conclude the chapter with our main takeaways in Section 5.8.

5.1 Motivation

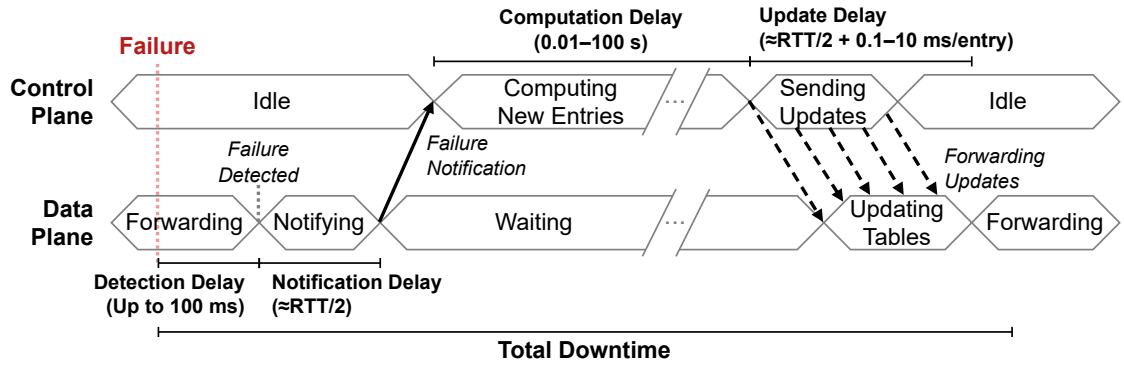
As previously described in Chapter 1, communication networks need to be resilient to provide high levels of availability. Modern services and applications rely on that as they are increasingly distributed into microservices and spread not only across one but multiple networks, their edges, and end-points (BALAKRISHNAN, 2021). Among events that may impact network operation are link and device failures. Recent measurement studies show that such failures happen quite frequently (a few failures per hour) and in all of the different types of network (e.g., enterprise (TURNER et al., 2012), data center (GILL; JAIN; NAGAPPAN, 2011; GOVINDAN et al., 2016), backbone (MARKOPOULOU et al., 2008), and WAN (TURNER et al., 2010; GOVINDAN et al., 2016)). Traditionally, network operators could simply wait for the routing protocol to converge and reroute around these failures. However, experience has shown that these protocols take considerable time to fully recover from a failure (YEGANEH; TOOTOONCHIAN; GANJALI, 2013). This delay in reacting to failures leads to massive packet drops, especially as networks grow larger and faster. In this section, we motivate the case for FELIX by describing the drawbacks and limitations of state-of-the-art approaches for network failure

resiliency.

SDN with OpenFlow (SDN-OF)

In the classic SDN model, the controller learns of network failures from switches. It then recomputes the forwarding tables and updates them on each switch. Between the time when a link fails and the controller updates the forwarding tables on an affected switch, packets simply "fall of the wire." Fig. 5.1 illustrates all of the delays that factor into a recovery from a failure in a traditional SDN with OpenFlow scenario. Typically, the downtime time is dominated by the time to compute the updated paths in the controller, which can take from milliseconds to minutes depending on the network size. The controller thinking time may be reduced by computing and caching forwarding entries for possible failure scenarios ahead of time. Nevertheless, the delay to get the failure notifications to the controller, propagate the new forwarding entries to data plane devices, and install such entries on the fast path of these devices is still significant. Consequently, even in the best-case scenario, where the necessary updated forwarding entries are already computed and ready to go, there will be significant packet loss, especially as networks are progressively upgraded to have higher bandwidth capacity.

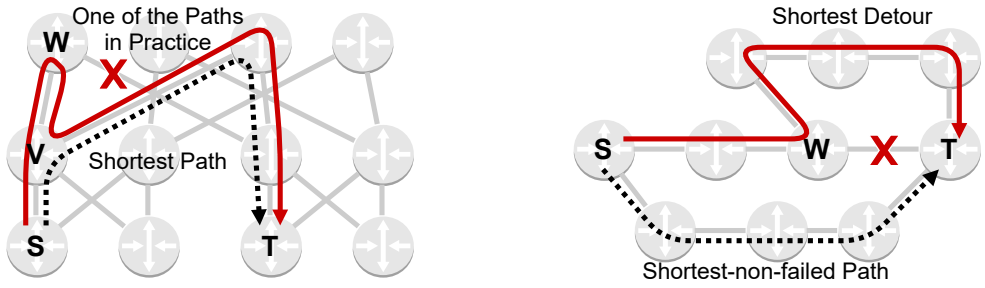
Figure 5.1: Failure recovery delay factors for SDN with OpenFlow approaches.



Switch-local failover (SLF) mechanisms

Mechanisms such as OpenFlow Failover Groups (MCKEOWN et al., 2008; ONF, 2015) and PURR (CHIESA et al., 2019) seek to reduce packet loss. These mechanisms enable the controller to configure switches with alternative next hops to be used in case the interface to the primary next hop is down. Failover mechanisms are especially useful for scenarios in which ongoing failures can be completely bypassed by alternative forwarding carried out solely by switches local to these failures. Unfortunately, not all failures can

Figure 5.2: Limitations to existing failover mechanisms and failure-inferencing approaches in the data plane.



(a) FIFR approaches

(b) TBFF approaches

be bypassed in such a way, unless the network is engineered with that goal in mind (LIU et al., 2013b). As previously mentioned, in general topologies, traffic may need to be forwarded backwards for a few hops before it can resume moving towards the destination through an alternative route.

Failure-inferencing fast reroute (FIFR) approaches

This line of work aptly represented by FIFR (LEE et al., 2004; ZHONG et al., 2005; NELAKUDITI et al., 2007) executes forwarding based not only on the destination of packets but also on their incoming interface. The main idea of this forwarding method is that if a packet arrives a device through an interface that was actually supposed to be the next hop of the packet, the device can infer that a failure is present down the main path (making it be sent back). The packet should then be forwarded through an alternative next hop. FIFR is more resilient to failure scenarios than switch-local failover mechanisms. For example, in Fig. 5.2a, under FIFR, whenever forwarding device V receives a packet with destination to T from W, it can infer that the shortest path (V→W→T) is not available and forward the packet through one of the alternative viable paths to T. The main limitation of FIFR is that no state is stored on devices, so every packet would have to go back and forth (once) on devices V and W before using an alternative path. In some situations, this causes an imbalance when trying to perform equal-cost multi-path routing, which can be detrimental to the flows relying on those paths. For example, in Fig. 5.2a, the lack of state in devices makes some of the packets of the traffic from S to T subject to a longer path than other packets.

Tagging-based fast failover (TBFF) approaches

Yet another line of work, represented by MRC (KVALBEIN et al., 2009) and SPIDER (CASCONE et al., 2017), seeks to achieve better resiliency and path guarantees (with data plane reaction) than their counterparts. These approaches pre-install a small set of routing configurations on forwarding devices and tag packets whenever they need to be forwarded through an alternative path due to a locally detectable failure. Devices receiving tagged packets forward them (and future packets in the flow) according to the respective pre-installed configuration. MRC and SPIDER can guarantee only single-node-or-link failure resiliency in bi-connected topologies (i.e., those that remain connect when removing any single node). Furthermore, they generally select the shortest detour from the point of failure to forward packets. Fig. 5.2b shows a case where shortest detour rerouting results in unnecessary extra hops when a shorter, more direct path from S to T is available in the network. An analysis we carried out using real-world wide-area networks (WAN) from the REPETITA dataset (GAY; SCHAUS; VISSICCHIO, 2017) suggests that path stretching in shortest detours is frequent and can be critical in typically sparse WAN topologies.

We close this motivation section with Table 5.1, which summarizes this conceptual review of the state of the art and compares it to FELIX. Our system FELIX seeks to fill the gap left by existing approaches. By introducing a failure-aware packet processing pipeline and a network-wide coordination protocol, we can send traffic along the shortest-non-failed path as long as the network is a single connected component. This, in connection with pre-planning control plane algorithms, allows FELIX to respond to failures in an optimal way at data plane time scales.

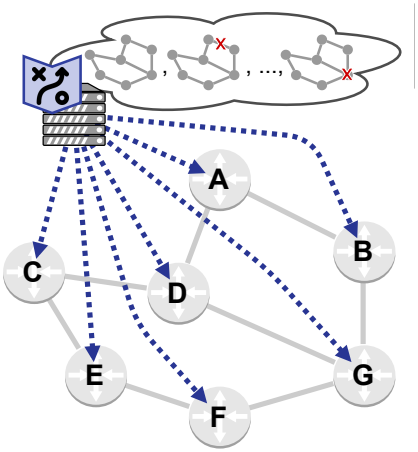
Table 5.1: Summary of the conceptual comparison of existing approaches for failure recovery and FELIX.

Approach	**Timescale**	**Resiliency**	**Path Optimality**
SDN-OF	Control Plane ○	Any Failure ●	Shortest Path ●
SLF	Data Plane ●	Local Failure ○	No Guarantee –
FIFR	Data Plane ●	Single Failure ◐	No Guarantee –
TBFF	Data Plane ●	Single Failure ◐	Shortest Detour ○
FELIX	Data Plane ●	Any Failure ●	Shortest Path ●

5.2 FELIX

In this section, we give an overview of how FELIX responds to network failures. Next, in Section 5.3, we delve into the design and implementation details of FELIX. The proposed architecture for FELIX is composed of two main components: (a) a failure-aware routing (and strategist) application in the control plane and (b) a custom packet processing pipeline running on forwarding devices. We note that although we focus on a link failure in our example, FELIX is not limited to failures on single links, but can also deal with switch failures, fiber cuts, and other failures involving multiple links in a shared-risk group (see Section 5.3.3 for more details). We use Fig. 5.3 to illustrate the following discussion.

Figure 5.3: Example of FELIX's approach to safeguard against and reroute around an example link failure.



(a) Planning for normal and *one-more-failure* scenarios.

Node A

tactic	dst	nhop
Normal	G	B, D
...	...	...
D–G	G	B
...	...	...

Node B

tactic	dst	nhop
Normal	G	G
...	...	...
...	...	...
...	...	...

Node C

tactic	dst	nhop
Normal	G	D
...	...	...
D–G	G	E
...	...	...

Node D

tactic	dst	nhop
Normal	G	G
...	...	...
D–G	G	A
...	...	...

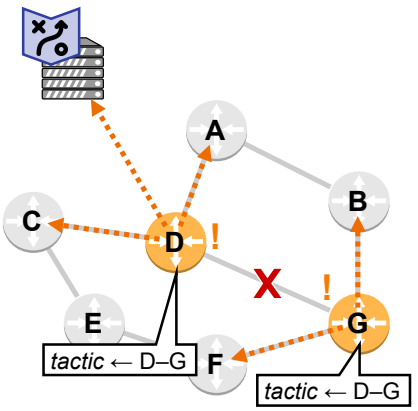
Node E

tactic	dst	nhop
Normal	G	F
...	...	...
...	...	...

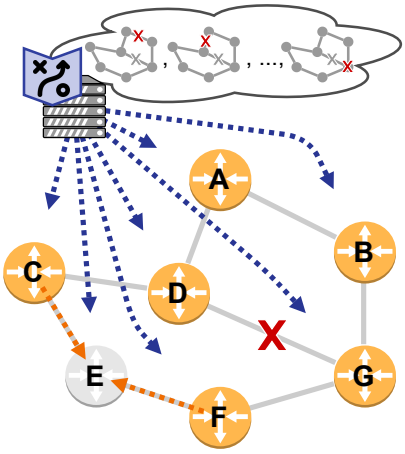
Node F

tactic	dst	nhop
Normal	G	G
...	...	...
...	...	...

(b) Forwarding entries for normal network state and for when link D–G has failed.



(c) Locally handling the failure of link D–G.



(d) Coordinating network-wide rerouting in the data plane and preparing for the future in the control plane.

Planning for failure scenarios

FELIX takes a proactive approach to dealing with failures. The first step in FELIX's recovery workflow is carried out before any failures strike the network, when the network is operating normally (Fig. 5.3a). In this step, the routing application computes how packets should be forwarded by each switch in the network during normal operation and installs the necessary forwarding entries on them. It also computes how packets would need to be forwarded should any single network element (e.g., link, switch, optical cable) fail and installs alternative forwarding entries on the switches (See Fig. 5.3b where entries are shown to deal with the failure of link D–G). We refer to this approach to forwarding entry computation as *one more failure* protection. Section 5.3.4 goes into further detail why that is both effective and essential in real networks. Finally, the application also installs instructions for transitioning between the different forwarding tactics, where a tactic is a set of forwarding entries across multiple switches that handles a particular failure scenario.

Handling local failures

The packet processing pipeline running inside switches is designed to detect failures to locally connected links and transition to the appropriate forwarding tactic (§5.3.2). When a switch detects a failure, besides transitioning to another tactic, it will also notify its neighbors and the routing application of it. Switch notification is done through a specially designed lightweight protocol that enables switches to both communicate observed failures and synchronize their active forwarding tactic. Fig. 5.3c depicts a case where the link between nodes D and G fails. Nodes D and G both transition to forwarding tactic D–G (see call-outs) and notify their neighbors and the routing application of the failure (see orange arrows).

Coordinating network-wide rerouting

Switches relay notifications in the network until all switches are aware of the failure, enabling the entire network to transition to a forwarding tactic appropriate to the new state of the network (§5.3.3). We highlight that both of the procedures just described, failure notification and tactic transition, are executed *entirely in the data plane* based on pre-configured match-action tables. This approach frees the data plane from having to wait for the control plane to make decisions upon a failure. No communication between

control and data planes is required when a new failure arises. The data plane is aware of the actions that should be carried out ahead of time, enabling it to react to failures independently and immediately.

Going back to our description of the failure recovery workflow, upon receiving a notification, a neighbor switch will update its forwarding tactic and disseminate the notification to its own neighbors (see orange arrows in Fig. 5.3d). This procedure takes place in the neighbor switch unless its is already aware of the failure, which can be due to having detected the failure locally or having received the same notification previously from other switches. In case the switch is already aware of the failure, it simply discards the notification. In some situations, when updating its forwarding tactic, the neighbor switch may need to make an additional tactic transition. These situations are due to a failure involving multiple links in a shared-risk group (e.g., the set of links connected to a switch). Whenever an additional transition is necessary, the switch updates the notification before sending it to its neighbors. In Section 5.3.3, we describe these coordination procedures in more detail.

Preparing for future failure scenarios

When the routing application is notified of a failure, it will start planning for the event of *one more failure* to happen so that it can pre-configure the data plane to act appropriately if and when the time comes. Fig. 5.3d exemplifies the routing application planning for more failures in the new current network state (where link D–G has failed). In this context, we highlight that the control plane has a strategic role, looking ahead and preparing for failures, while the data plane has a tactical role, acting according to failures when they happen, as proactively instructed by the routing application.

5.3 Design and Implementation

FELIX relies on multiple complementary components to prepare for, detect, and reroute around failures in the data plane efficiently and effectively. In Section 5.2, we provided an overview of how these mechanisms interact and how abstractions are applied. In this section, we describe each of these elements individually in greater detail following a bottom-up approach. For the data plane procedures, we consider P4-programmable devices as the target platform and provide design and implementation details in this context.

We start by describing how packets are forwarded under FELIX (§5.3.1). Next, we show how data plane devices detect and handle local failures (§5.3.2) as well as coordinate network-wide rerouting around these failures (§5.3.3). Finally, we describe how FELIX's routing application plans ahead for multiple failure scenarios (§5.3.4).

5.3.1 Forwarding Packets in the Data Plane

Packet forwarding is based on two complementary lookup tables, NORMALFWD and ALTFWD, as well as a state variable, *fwd_tactic*. Algorithm 5 presents pseudocode of the packet forwarding procedure. The variable *fwd_tactic* represents, by way of a number, the current network operation state and forwarding tactic. A unique *fwd_tactic* number is assigned for each failure scenario (see more in §5.3.4). The procedure starts by identifying the normal output port for the packet (Line 2). NORMALFWD represents a normal forwarding lookup table with a simple extension to indicate the switch that is expected to be the last to forward the packet before it leaves the network (i.e., traffic egress point). Next, the switch will check if the current network *fwd_tactic* is not normal (L.3, i.e., a failure scenario is ongoing), in which case it will also check, by looking up table ALTFWD, if an alternative output port must be used to forward the packet considering the current *fwd_tactic* and *dst_node* (L.4). In case both checks pass, the switch replaces the value of *output_port*, initially set by table NORMALFWD, with that defined by ALTFWD (L.5) and then forwards the packet accordingly.

In our P4-based prototype, *fwd_tactic* is stored in a stateful register and each lookup table is implemented as an independent match-action table. Assuming an IP network, entries in NORMALFWD have the form $\langle$*dst_addr*, *output_port*, *dst_node*$\rangle$, where *dst_addr* is the key in a longest-prefix match and *output_port* is the port connected to the next hop during normal network operation. The additional field *dst_node* indicates the target switch to send packets to in the case of one or more failures in the network. We note that FELIX only extends the action part of the normal forwarding table that would already be present in the switches. ALTFWD carries out alternative forwarding at the topology-graph level. Entries have the form $\langle$*fwd_tactic*, *dst_node*, *output_port*$\rangle$, where *fwd_tactic* and *dst_node* are the lookup keys in an exact match. Table ALTFWD has an entry $\langle$*ft*, *dn*, *op*$\rangle$ if and only if an alternative next hop *op* is necessary to be able to reach destination node *dn* under forwarding tactic *ft*. We describe, in Section 5.3.4, how these tables are populated.

Algorithm 5 Forward a data packet.

Input: Data *pkt*

```
1: state fwd_tactic
2: output_port, dst_node ← NORMALFWD.get(pkt.dst_addr)
3: if fwd_tactic is not NORMAL then
4:   if ALTFWD.hit(fwd_tactic, dst_node) then
5:     output_port ← ALTFWD.get(fwd_tactic, dst_node)
```

Output: Send *pkt* via port *output_port*

Packet forwarding in FELIX was designed considering the type and availability of memory resources in modern programmable targets. For example, RMT (BOSSHART et al., 2013) makes available Ternary Content-Addressable Memory (TCAM) and Static Random-Access Memory (SRAM) units. SRAM is usually present in larger quantity since it is cheaper than TCAM. The authors of RMT propose it to have a total of $370\,\mathrm{Mb}$ of SRAM and $40\,\mathrm{Mb}$ of TCAM. When programming in P4, TCAM is used to store lookup keys when match-action tables require ternary matching, while action data is stored in SRAM. Tables with exact matching are mapped completely to SRAM (BOSSHART et al., 2013). With that background in mind, and to minimize memory costs, FELIX's design focuses on utilizing solely SRAM. More specifically, the ALTFWD is an exact-match table totally mapped to SRAM, both for keys and for action data. FELIX extension to the NORMALFWD table only changes the action data part of it, limiting the additional memory usage to SRAM. We evaluate FELIX memory usage in Section 5.4.3. In Section 5.6, we also discuss the mapping of match-action tables to memory in further detail.

5.3.2 Handling Local Failures

In addition to a procedure to forward packets, in FELIX each switch is also programmed with procedures to deal with local failures and coordinate network-wide rerouting with other switches. We will describe rerouting coordination in the following section. In this section, we focus on how switches detect and reroute around local failures. Algorithm 6 presents a pseudocode for this procedure.

The procedure is executed for every data packet received by switches and starts by checking if the switch port status has changed (Line 4). The metadata field *port_status*[1] is a bitstring that indicates at each bit i, whether the i-th switch port is active (1) or not (0). If

[1]In line with the work by Chiesa et al. (2019), we assume the port status to be readily available as standard metadata to the P4 program.

Algorithm 6 Detect and reroute around a local failure.

Input: Data *pkt*

1: **metadata** *port_status*, *switch_ID*
2: **state** *prev_port_status*, *fwd_tactic*,
3: *n_transitions*, *transition_TS*
4: **if** *port_status* $\neq$ *prev_port_status* **then**
5: *element* $\leftarrow$ ELEMENT.get(*prev_port_status*, *port_status*)
6: *fwd_tactic* $\leftarrow$ TACTICS.get(*fwd_tactic*, *element*)
7: *prev_port_status* $\leftarrow$ *port_status*
8: *n_transitions* $\leftarrow$ *n_transitions* $+1$
9: *transition_TS* $\leftarrow$ CURRENTTIMESTAMP()
10: Create and send a new failure *notif*:
11: $\langle$*fwd_tactic*, *n_transitions*, *switch_ID*, *element*$\rangle$

Output: Continue *pkt* processing

the *port_status* has changed since the last check, the switch will make a forwarding tactic transition. To do so, first, it uses a special lookup table ELEMENT to determine the link which is connected to the port that is now down, breaking the communication to one of its neighbors (L.5). Next, the switch will update its tactic by way of table TACTICS (L.6), which considers the current *fwd_tactic* and the locally connected failed *element*. We note that the broken communication can be due to a failure of the link connected on the node port or of the neighbor device on the other side of the link. However, at this point, the switch cannot discern between the two with its local view but needs to coordinate with other switches in the network to do so, which we describe in Section 5.3.3.

To conclude the procedure, Lines 7-9 update the stateful variables needed for future calls (packets) as well as to create a new failure notification to be sent to neighbor switches and the routing application (L.10-11). The original data *pkt* is processed and forwarded normally and, in our prototype, also cloned to create the failure notification packet (since P4 only has replication but no packet creation primitives). Failure notifications have the form $\langle$*new_fwd_tactic*, *n_transitions*, *announcer*, *element*$\rangle$, where *new_fwd_tactic* is the new forwarding state after the failure, *n_transitions* is the number of state transitions so far, *announcer* is the switch ID of the node creating the notification, and *element* is the element ID of the link (or shared-risk group) that is now down.

Algorithm 7 Reroute around a remote failure.

Input: Failure *notif*

```
 1: metadata switch_ID
 2: state fwd_tactic, n_transitions, transition_TS
 3: if notif.new_fwd_tactic = fwd_tactic
 4:     or notif.n_transitions < n_transitions then
 5:   Discard the failure notification notif
 6: else
 7:   elapsed ← (notif.ingress_TS − transition_TS)
 8:   if elapsed ≤ THRESHOLD and
 9:       SFTACTICS.hit(fwd_tactic, notif.element) then
10:     notif.new_fwd_tactic, transitions ←
11:                         SFTACTICS.get(fwd_tactic, notif.element)
12:     notif.n_transitions ← notif.n_transitions + transitions
13:     notif.announcer ← switch_ID
14:   fwd_tactic ← notif.new_fwd_tactic
15:   n_transitions ← notif.n_transitions
16:   Propagate the failure notification notif
```

Output: Either discard (L5) or propagate (L16) *notif*

5.3.3 Coordinating Network-Wide Rerouting

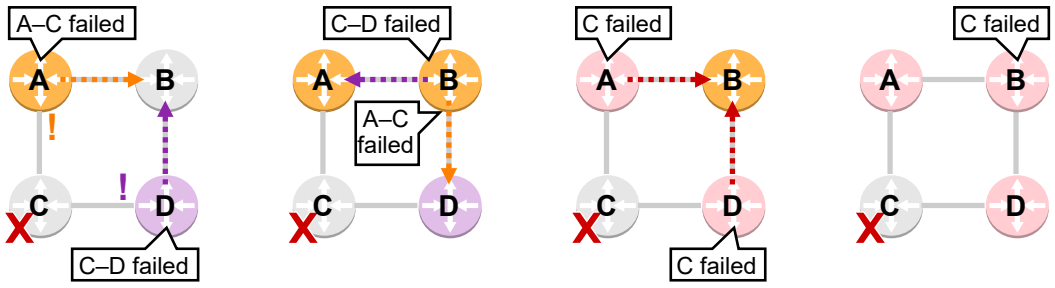
In this section, we detail how FELIX-enabled devices coordinate among themselves using failure notifications to reroute around failures. We start by describing how they communicate to deal with single-link failures and then show how to extend the approach to deal with multi-link failures under shared-risk events.

Single-link failures

As described in Section 5.3.2, when a switch detects a failure to a link directly connected to one of its ports, it sends a failure notification message to its neighbor switches. Algorithm 7 presents the procedure run by switches when they receive a failure notification.

The procedure starts by checking whether the switch is already aware of the failure (Line 3) or whether the notification is outdated (L.4), in which cases the notification is simply discarded (L.5). Otherwise, the switch will update its state variable *fwd_tactic* and propagate the notification to its neighbors, as shown in Lines 14-16. This procedure results in switches in the network progressively disseminating the notification until all of

Figure 5.4: Steps to reroute around a shared-risk failure.



them are aware of the failure and, thus, can correctly and efficiently reroute traffic to avoid it.

Switch and other shared-risk multi-link failures

The procedure as just presented enables data plane switches to handle single link failures. Next, we describe how FELIX reroutes around shared-risk failures involving multiple links (e.g., due to a node failure). The rerouting happens in two stages. In the first, switches deal with their locally-observable failures. In the second, switches communicate to handle all failures in the shared-risk group. Fig. 5.4 exemplifies the main steps in the rerouting coordination. Consider the failure of Switch C in the figure. In Fig. 5.4a, when Switch A can't reach C, it will assume link A–C failed, transition to the appropriate forwarding tactic, and notify its neighbor B. Similarly, switch D will assume link C–D failed and proceed accordingly. Switch B will receive the notifications sent from both A and D and propagate them to the respective other neighbor (Fig. 5.4b), after having updated its own tactic. Upon receiving the failure notification created by Switch D, since A has observed another local failure recently, it will conclude that Switch C failed, and not only link A–C. Consequently, it will transition to the appropriate tactic and send an updated notification to B indicating C as failed (Fig. 5.4c). Switch D will proceed similarly. After receiving the updated notifications, Switch B will update again its tactic as instructed by A and D (Fig. 5.4d).

Lines 7-13 in Algorithm 7 formalize how switches process failure notifications to detect shared-risk failures and transition to the appropriate forwarding tactic. As shown previously in Algorithm 6, Lines 5-6, when a switch detects a failure it transitions into a new *fwd_tactic* that assumes only a single link failure. Additionally, it also saves a timestamp of the transition (Alg. 6, L.9). In view of that, in Algorithm 7, whenever a valid notification is received, the procedure will also verify whether the notification was

received within a reasonably short amount of time of the previously detected local failure (Lines 7-8). According to measurement studies (GOVINDAN et al., 2016; TURNER et al., 2012; GILL; JAIN; NAGAPPAN, 2011; TURNER et al., 2010; MARKOPOULOU et al., 2008), the majority of times, when two links fail simultaneously, they belong to the same shared-risk group. With that observation in mind, whenever a switch is aware of two single-link failures happening in a short period in time, it will transition to a tactic that handles the shared-risk group to which the two links belong by looking up table SF-TACTICS (L.9-11). Finally, Lines 12-13 update the notification currently being processed, before propagating it to neighbor switches, which will transition to the new tactic.

5.3.4 Planning for Failure Scenarios

In this section, we start by presenting how FELIX plans for a single failure scenario and then describe its approach to efficiently plan for many possible future failure scenarios.

Single failure scenario

Algorithm 8 presents the pseudocode for computing the necessary table entries to handle a failure scenario (or event). The algorithm can be split in two parts. The first (Lines 1-7) calculates the alternative forwarding entries while the second (L.8-17) determines strategy entries to transition to the correct forwarding tactic for the scenario. The procedure starts by generating a new unique ID for the network state (and respective tactic) where elements in the set *event_elements* are those that would be failed in that event (in addition to those that failed previously, L.1). Next, it creates a graph representing such failure state and computes the (shortest-non-failed-path) next hops between all node pairs (L.2-5). FELIX only needs to save the alternative entries that are different from the normal forwarding (considering the non-failed network topology *graph*, L.6-7), since the latter can be used to forward packets correctly during failure. Consequently, for each failure scenario, FELIX installs only the necessary additional entries in the data plane, substantially saving data plane device memory.

In its second part, the algorithm checks whether the failure event involves only a single link (Line 8), in which case it will prepare tactic transition entries for the two nodes connected to such link (L.9-11). The tactic transition entries are populated in the

Algorithm 8 Plan for one failure scenario.

Input: Network topology *graph*, current *fwd_tactic*, currently *failed_elements*, would be failed *event_elements*

1: *fs* ← GETTACTICID(*failed_elements* ∪ *event_elements*)
2: *fgraph* ← *graph* − (*failed_elements* ∪ *event_elements*)
3: **for each** *src* **in** *fgraph.switches* **do**
4: **for each** *dst* **in** *fgraph.switches* **do**
5: *nh* ← SHORTESTPATHNEXTHOPS(*fgraph*, *src*, *dst*)
6: **if** *nh* ≠ SHORTESTPATHNEXTHOPS(*graph*, *src*, *dst*) **then**
7: ALTFWD[*src*].set(*fs*, *dst*, *nh*)
8: **if** |*event_elements*| = 1 **then** ▷ Single-link
9: *element* ← *failed_elements*.getSingleElement()
10: TACTICS[*element.nodeA*].set(*fwd_tactic*, *element*, *fs*)
11: TACTICS[*element.nodeB*].set(*fwd_tactic*, *element*, *fs*)
12: **else** ▷ Share-fate group
13: **for each** *ea* **in** *event_elements* **do**
14: *fsa* ← GETTACTICID(*failed_elements* ∪ {*ea*})
15: **for each** *eb* **in** *event_elements* −{*ea*} **do**
16: SFTACTICS[*ea.nodeA*].set(*fsa*, *eb*, *fs*, |*event_elements*| − 1)
17: SFTACTICS[*ea.nodeB*].set(*fsa*, *eb*, *fs*, |*event_elements*| − 1)

Output: ALTFWD, TACTICS

TACTICS table of data plane devices. Whenever the failure event involves multiple links, i.e., a shared-risk group, special SFTACTICS tactic transition entries are created (Lines 13-17). For every pair of links ⟨*ea*, *eb* ⟩ that are part of the failure event, entries are created on the nodes connected to *ea* such that they transition to the appropriate shared-risk tactic whenever they receive a notification of *eb* having failed after having detected *ea*'s failure locally. See Section 5.3.3 for more details on how this process takes place at runtime.

The worst-case computational complexity of Algorithm 8 is determined by the procedure to compute next hops between all switch pairs. For example, if implemented with Dijkstra's algorithm for shortest path, the algorithm complexity is given by $\mathcal{O}(m + n \cdot \log n)$ where n is the number of switches and m is the number of links in the network infrastructure. Other algorithms, such as (QIU et al., 2019), could be applied to achieve lower average complexity. After having computed the entries for a failure scenario, FELIX's routing application needs to install them into the data plane to prepare the network to tackle such scenario, which we will describe next. We evaluate, in Section 5.4.3, the time FELIX takes to prepare for possible future failures.

Algorithm 9 Continuously plan for failure scenarios.

Input: Network topology *graph*

1: Compute and populate entries for the normal forwarding tables NORMALFWD of data plane switches.
2: **while** *true* **do**
3: **for each** *fe* **in** the set of possible next failure events **do**
4: Plan for the failure of *fe.elements* (Algorithm 8)
5: Populate ALTFWD tables with new alternative entries
6: Remove outdated alternative entries from ALTFWD
7: Wait to receive a new failure *notif*
8: Remove failed element from *graph*

Many failure scenarios

Successfully planning for multiple failure scenarios involves tackling two challenges: the large number of possible such scenarios in real networks and the limited amount of memory available in programmable switches to store forwarding entries. More specifically, the number of possible failure scenarios in a network with n elements (e.g., links, switches, optical devices) is given by $\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$. Consequently, computing alternative forwarding entries for all possible failure scenarios would both take an impractically long time and require prohibitive amounts of memory. Despite that, network measurement studies (GOVINDAN et al., 2016; TURNER et al., 2012; GILL; JAIN; NAGAPPAN, 2011; TURNER et al., 2010; MARKOPOULOU et al., 2008) show that although failures happen frequently (a few minutes apart), it is very unusual for two elements (e.g., links, switches, optical-fiber cable) to fail at the "same time", unless they belong to the same shared-risk group. For example, two links connected to the same switch are perceived as failed whenever the switch itself fails.

FELIX's routing application leverages this fact about failure correlation to implement what we refer to as the *one-more-failure* approach to plan for failure scenarios. In other words, in any given point in time during network operation, FELIX pre-computes and installs only the entries necessary to handle a single additional failure event, which may consist of a single link or multiple links in shared-risk group. Algorithm 9 presents an example pseudocode for the one-more-failure approach. The algorithm starts by creating normal routing entries (Line 1) and, then, repeatedly plans for additional failures (L.3-6), receives failure notifications (L.7), updates the topology graph (L.8) and goes back to planning for failures.

5.4 Evaluation

This section seeks to answer two main questions: a) How much faster can networks recover from failures using FELIX when compared to traditional SDN with OpenFlow approaches? and b) How well does FELIX scale to large networks? We start by describing our experimental setup in Section 5.4.1. Next, we present and discuss our experimental results in Sections 5.4.2 and 5.4.3.

5.4.1 Experimental Setup

Scenarios

For the evaluation, we focused on two types of topologies: data center (DCN) and wide-area networks (WANs). For data center, we consider fat-tree topologies with progressively larger number of switches. The workload consists of symmetric traffic between all host pairs, which is load-balanced via Equal-Cost Multi-Path (ECMP) routing. For WAN, we use the REPETITA dataset (GAY; SCHAUS; VISSICCHIO, 2017), which consists of more than 260 real-world topologies and traffic matrices. We evaluated each approach considering all available networks. In the interest of clarity, we selected three of the most representative networks of each type with varying sizes. Table 5.2 presents the metadata (type, label, number of nodes, and number of links) for each of the selected networks.

Testbed

We developed prototypes[2] of FELIX targeting the reference P4 software switch (BMv2 – Behavioral Model version 2 (P4 LANGUAGEM CONSORTIUM, 2014)) and the NetFPGA-SUME[3] board (ZILBERMAN et al., 2014). We used as testbed for the functional evaluation Mininet-emulated networks running on a dedicated Linux 4.4 server with 2x Intel Xeon Silver 4208 $2.1\,\mathrm{GHz}$ 8-core 16-thread processors (32 total threads), 8x $16\,\mathrm{GB}$ $2400\,\mathrm{MHz}$ RAM, and $2\,\mathrm{TB}$ of NVMe SSD storage. The BMv2 prototype is mainly targeted at enabling us to evaluate FELIX in a high-fidelity environment via Mininet

[2] Our prototypes will be made available upon publication.

[3] This second prototype is not fully functional due to the (current) lack of support for port status metadata and packet replication in the NetFPGA P4 libraries.

Table 5.2: Summary of the networks used for evaluation.

Network	Type	Label	Nodes	Links
FatTree8	DCN	FT8	80	256
FatTree16	DCN	FT16	320	2048
FatTree32	DCN	FT32	1280	16,384
Bell Canada	WAN	BC	48	64
CogentCo	WAN	CG	197	243
SprintLink	WAN	SL	315	972

(LANTZ; HELLER; MCKEOWN, 2010), but limited to scaled-back scenarios. We also designed an analytical model of the system to enable us to evaluate FELIX in larger scale scenarios. To examine overheads on device memory, we assume the RMT/Tofino architecture (BOSSHART et al., 2013; BAREFOOT NETWORKS, 2020) as hardware targets. We use our NetFPGA-SUME prototype to evaluate the computational resource requirements of FELIX on fixed-resource platforms.

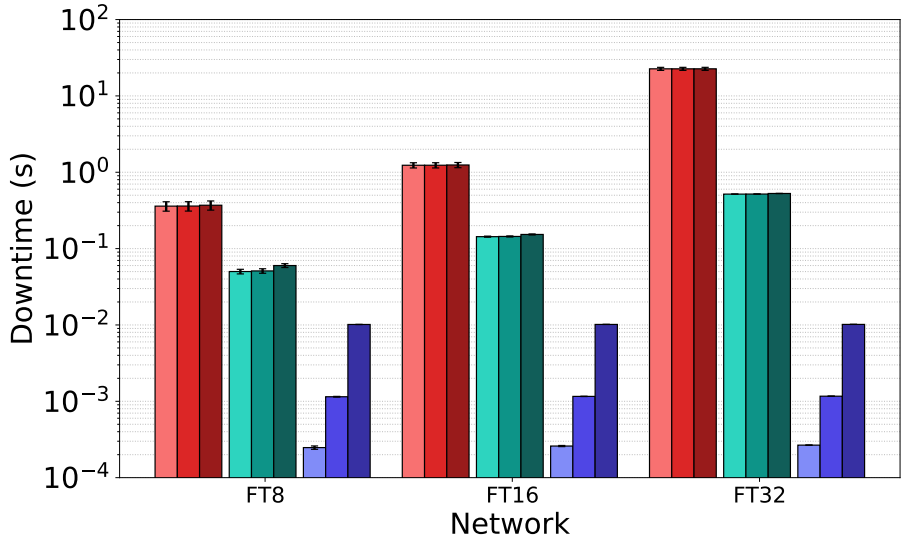
5.4.2 Performance

The main performance indicator for a failure rerouting approach is the time it takes to recover completely from a failure, i.e., its downtime. In this section, we compare FELIX's downtime with that of existing rerouting approaches. We limit the scope of our comparison to approaches that, like FELIX, provide shortest-non-failed routing. Namely, we consider the two SDN-OpenFlow approaches described in Section 5.1: one that computes alternative forwarding entries only upon failure and another that pre-computes and caches in the control plane the forwarding entries necessary for each failure scenario. We refer to the first approach as Standard SDN (S-SDN) and to the second as Pre-Compute SDN (PC-SDN). As shown previously in Fig. 5.1, the total downtime can be decomposed into four delay factors, which we now formalize in Equation 5.1.

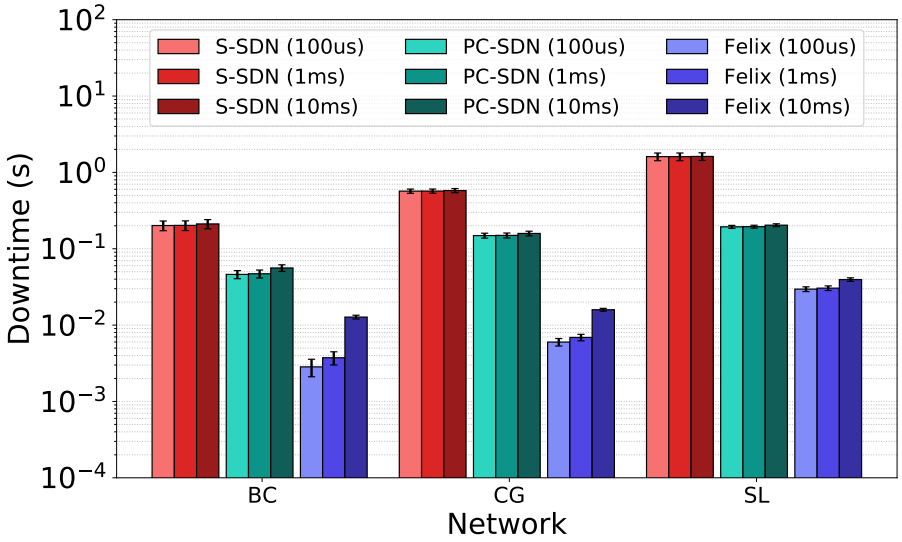
$$T = T_{detection} + T_{notification} + T_{computation} + T_{update} \tag{5.1}$$

Each one of the factors in Equation 5.1 can vary depending on the mechanisms and steps involved in their procedure. With regards to $T_{detection}$, existing mechanisms are able to detect failures in the order of milliseconds or lower (KATZ; WARD, 2010). In our evaluation, we consider three possible values for the expected detection delay: 0.1,

Figure 5.5: Downtime with varying detection delay values (entry installation delay fixed at 1 ms).



(a) DCN



(b) WAN

1, and 10 ms. $T_{notification}$ is defined by the communication delays between data plane devices and the SDN controller (or the other forwarding devices in the network in the case of FELIX). Consequently, this delay varies according to the network under study. The computation delay is a function of the network size and the controller processing capacity. In our analysis of S-SDN, we consider the runtime of a parallel implementation of Dijkstra's algorithm for shortest paths for all pairs, running with 30 parallel threads on our dedicated testbed server, to be the value of $T_{computation}$. For the same delay in PC-SDN, we consider only a fixed $10\,\mathrm{ms}$ delay for cache lookup. We consider this small, fixed lookup delay seeking to understand the best case scenario of what is possible with PC-SDN. For FELIX, the data plane coordinates failure recovery independently of the control

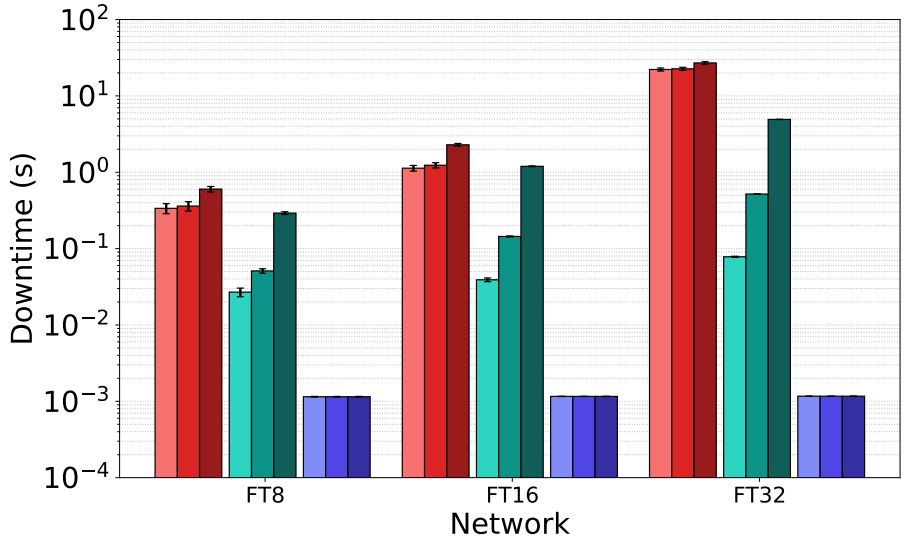
plane, so the computation delay does not factor into the downtime. Finally, T_{update} is influenced by communication delays and the time it takes to install entries on forwarding devices. Based on recent measurement studies (HUANG; YOCUM; SNOEREN, 2013; KUŹNIAR; PEREŠÍNI; KOSTIĆ, 2015), we use three different values for the update delay: 0.1, 1, and 10 milliseconds. Each of these three values represents an update rate of $10{,}000$, 1000, and 100 entries per second, respectively. We note that the third and fourth delays do not factor into FELIX's downtime, since it does not defer to the controller to install alternative entries at the time of failure.

We evaluate the effect of variations in the detection time in the total downtime. For this purpose, we vary the detection time while keeping the entry installation time fixed at the middle value of $1\,\mathrm{ms}$ (i.e., $1000\,\mathrm{entries/s}$). Fig. 5.5 presents the results for the two SDN-OpenFlow approaches and FELIX. Each bar[4] in the figure represents the average downtime among all possible network failures for a specific combination of recovery approach, detection delay, and network topology. In Fig. 5.5, we observe that the detection delay has a clear influence on FELIX's downtime. In DCN topologies (Fig. 5.5a), the downtime is closely approximated by the detection delay value. These results are expected since the remaining factor, $T_{notification}$, is dictated by the network communication delays, which in DCNs is at around the same order of magnitude of the minimum detection delay of $0.1\,\mathrm{ms}$. In WANs (Fig. 5.5b), this correlation is not as evident due to the communication delays being in the order of several milliseconds, slightly masking the effect of the detection delay. Compared to S-SDN, FELIX reduces downtime by a factor of at least 17 times in the BC network with $T_{detection}$ set to $10\,\mathrm{ms}$. The speedup can reach up to 5 orders of magnitude (in FT32 with $T_{detection}$ set to $0.1\,\mathrm{ms}$). Compared to PC-SDN, FELIX's downtime speedups range from about 4.4 times to about 2000 times, in the same scenarios as S-SDN, respectively. These speedup results show the benefits of FELIX's approach to failure recovery. The comparison to S-SDN demonstrates the benefits of eliminating the need to compute alternative entries in reaction to a failure, while the comparison to PC-SDN shows the additional gains possible by not having to install those entries at the time of failure.

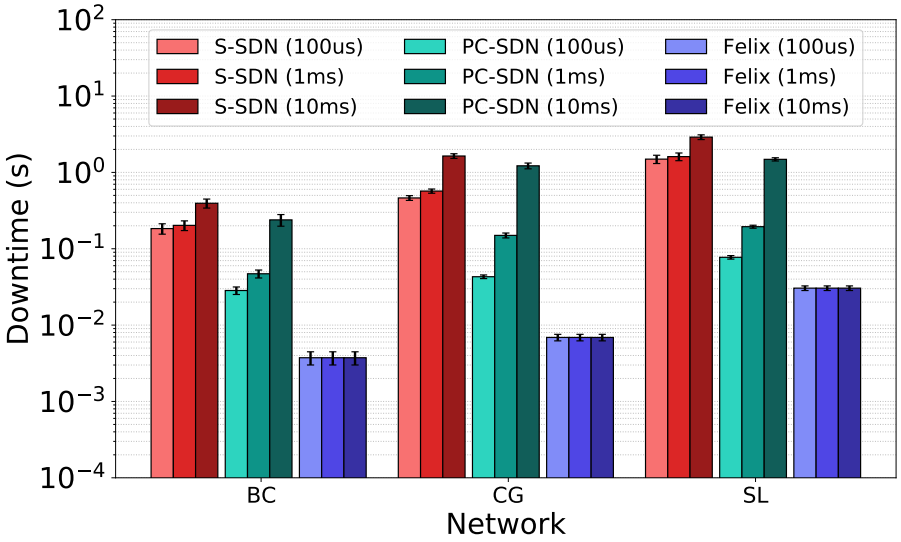
Next, we focus on the effect of the entry installation delay in the downtime. For this part of the evaluation, we fix the detection delay at $1\,\mathrm{ms}$. Fig. 5.6 presents the results obtained for these new scenarios. As expected, FELIX is not impacted by variations of the entry installation delay since no new entries need to be installed at the time of failure.

[4]Whiskers show the confidence interval with 99% confidence level.

Figure 5.6: Downtime with varying forwarding entry installation delay values (detection delay fixed at 10 ms).



(a) DCN



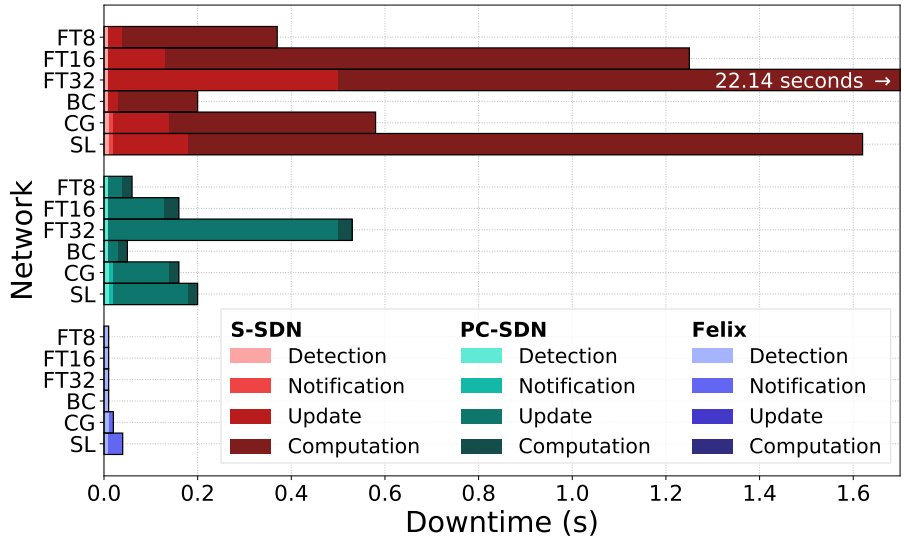
(b) WAN

We note that in contrast to FELIX, both of the SDN-OpenFlow approaches show variation in downtime as a result of the variation in installation delay, but especially PC-SDN. The downtime speedup of FELIX can reach up to four orders of magnitude when compared to S-SDN and three orders of magnitude when compared to PC-SDN. These perceived variations for the SDN-OpenFlow approaches suggest that the downtime for S-SDN is mainly dominated by the computation delay, while the update delay tends to represent a large fraction of the downtime for PC-SDN.

Seeking to further understand the effect of each delay factor on the downtime, in Fig. 5.7, we breakdown the downtime of each approach into its four factors[5]. For this

[5]Note that we clip the S-SDN-FT32 bar to avoid flattening the graph.

Figure 5.7: Downtime factor cost breakdown.



analysis we set the $T_{detection}$ to $10\,\mathrm{ms}$ and the installation delay to $1\,\mathrm{ms}$. We observe that, for FELIX, the detection delay accounts for the majority of the downtime for four of the six evaluated network topologies, namely, BC, FT8, FT16, and FT32. In the other two topologies, CG and SL, a significant portion of the downtime is also explained by the notification delay, which, in turn, is explained by the communication delays in these large WAN topologies. Fig. 5.7 confirms our suspicion that the downtime for S-SDN is mainly a function of the computation delay. It accounts from about 75% (CG) of the S-SDN downtime to about 98% (FT32) in the evaluated topologies. The figure also confirms our expectation that the update delay represents a large fraction of the PC-SDN downtime. This fraction ranges from about half of the PC-SDN downtime for the smallest topologies evaluated (FT8 and BC) to about nine-tenths (or 90%) in the largest topology (FT32).

Finally, we evaluate how this reduction in downtime provided by FELIX translates into fewer packet drops. In this analysis we focus our comparison to S-SDN to observe the benefits of joint pre-computation and data plane caching of alternative forwarding entries. We carried out experiments to measure the amount of packets lost by each approach on two network topologies. Representing DCNs, we chose the well known 4-port fat-tree topology containing 20 nodes and 32 links. Representing WANs, we chose the Abilene continental-wide backbone topology with 11 nodes and 14 links. In line with recent works in the area of programmable data planes (HSU et al., 2020), we choose smaller topologies to guarantee consistent performance in the emulated environment, without loss of generality. The measurement results underline our previous findings, with S-SDN losing on typically 6 times (Abilene) or 13 times (FatTree4) as many packets as FELIX.

In the worst case for S-SDN, a link failure between nodes in the aggregation and core layers in the FatTree4 topology, it loses about 112 times more packets than FELIX. The results and discussions presented in this section show the benefits of FELIX approach (bringing independence from the control plane in the time of a failure), which grows as networks get larger, faster, and with lower communication delays.

5.4.3 Scalability

Next, we evaluate how well FELIX scales on different network types and sizes.

Memory usage

One of the sources of overhead introduced by FELIX is the additional memory used to cache alternative forwarding entries that handle the failure scenarios. The overhead is mostly due to the alternative forwarding entries, but some overhead is also due to the NEIGHBOR and TACTICS tables. As previously described, FELIX implements the alternative forwarding table in SRAM, since it is both more plentiful and more energy efficient than TCAM. Table 5.3 shows FELIX memory usage when applying the one-more-failure approach to planning for future failures. The results show that FELIX makes sensible use of device memory resources. For the evaluated topologies, it allocates at most $7.44\,\mathrm{Mb}$ of SRAM (FatTree32 topology, Column "SRAM Usage Mbits"). This amounts to only $2\,\%$ (Column "SRAM Usage Util") of the SRAM available in the RMT/Tofino chips, which have $370\,\mathrm{Mb}$ of SRAM (BOSSHART et al., 2013). For the WAN topologies, the memory usage stays well under $1\,\%$, with the maximum utilization at $0.53\,\%$, or only about $2\,\mathrm{Mb}$ of SRAM for the SprintLink network.

Packet processing resources

FELIX's computational resource requirements depend on the target platforms, of which the most demanding are platforms with fixed resources (e.g., FPGAs and ASICs). To evaluate resource usage on such a fixed-resource target, we used our prototype for the NetFPGA-SUME development board (ZILBERMAN et al., 2014). Our prototype required $74{,}645$ lookup tables (LUTs) and $204{,}775$ registers. These numbers represent, respectively, $17.2\,\%$ and $23.6\,\%$ of the available resources. When compared to the resources demanded by a baseline IP-destination forwarding pipeline (which we implemented for

Table 5.3: Summary of the scalability results.

Network	SRAM Usage		# Alt Entries		Planning
Label	Mbits	Util	Total	/Node	Runtime
FT8	0.66	0.18%	28,288	484	0.426s
FT16	1.31	0.35%	914,432	4008	11.361s
FT32	7.44	2.00%	29,335,552	32,464	880.716s
BC	0.66	0.18%	6922	369	0.203s
CG	1.53	0.41%	201,676	4648	2.280s
SL	1.97	0.53%	636,312	6521	7.100s

the sake of comparison), the overhead observed is negligible. We expect comparable processing resource requirements for similar targets. From these estimates, we conclude that FELIX introduces a practicable level of overhead on packet processing.

Pre-compute runtime

We also evaluate the time it takes for FELIX's routing application (Algorithm 9) to pre-compute the forwarding entries needed to handle the next possible failures, as shown in Column "Runtime" of Table 5.3. FELIX's one-more-failure algorithm scales polynomially to the number of nodes and links in a network. For the selected WAN topologies, FELIX took at most $7.1\,\text{seconds}$ to compute all of the necessary entries. For the DCN topologies, FELIX took up to about $15\,\text{minutes}$ for the largest network (FatTree32). This represents a runtime of only about 50 milliseconds per failure scenario. We observe that the algorithm takes longer to run on DCN topologies, when compared to WANs, due to the higher number of possible paths between any pair of endpoints. We note that the one-more-failure algorithm is embarrassingly parallel, which enables its runtime to be substantially reduced by scaling up or out the computational resources assigned to the routing application. Another way to reduce runtime for DCN topologies would be to design an algorithm that exploits their regularity. We leave such design as future work.

Notification overhead

When a failure happens, the switches local to it send notifications to their respective neighbors. A neighbor switch propagates a notification packet to its own neighbors whenever the notification processing resulted in a change of forwarding tactics. If the processing does not result in a change of tactics, the neighbor does not propagate the notification. As a result, the number of notifications exchanged within the network λ is bound by $\lambda \leq (f+1) \cdot 2m$, where f is the number of links impacted by the failure and m

the total number of links in the network. In other words, a notification will be forwarded back and forth at most once in each link of the network for each link being part of the failure, which also means that the total collection of notifications is in essence uniformly distributed across the network.

5.5 Related Work

Programmable data planes enable novel, remarkably efficient solutions to failure recovery in communication networks. In this section, we highlight some of the existing approaches related to FELIX and position it within the state-of-the-art.

Qiu et al. (2019) proposed two algorithms for efficient recovery path computation upon link failures. Their work seeks to minimize the time it takes to compute the necessary updated forwarding entries to deal with a new failure, which accounts for the majority of the recovery delay (as we have discussed previously in Sections 5.1 and 5.4). In FELIX, we seek to take the computation delay out of the equation by pre-computing alternative entries and caching them on data plane devices during normal network operation, i.e., before failures happen. Inspired by (QIU et al., 2019), we designed new algorithms that efficiently compute recovery paths for not only one single link failure, but also for many links at a time and in shared-risk groups.

Pre-computing alternative forwarding entries can also require large amounts of memory, if not carefully managed, which is especially troublesome for data plane devices due to their limited memory resources. In view of that, some works have sought to minimize the amount of memory that is necessary in the data plane to store forwarding entries. For example, Plinko (STEPHENS; COX; RIXNER, 2016) was designed in the context of OpenFlow (MCKEOWN et al., 2008) switches and proposes a forwarding table compression algorithm to minimize Ternary Content-Addressable Memory (TCAM) usage, a type of memory that is both scarce and costly in existing hardware. Plinko's design and capacity to minimize memory usage is limited by the flexibility of the OpenFlow protocol and its forwarding model. In contrast, FELIX targets programmable data plane devices (e.g., with P4 (BOSSHART et al., 2014)) allowing for greater flexibility in defining forwarding model and table structures. Specifically, FELIX was designed to only use Static Random-Access Memory (SRAM) for alternative entries, incurring in lower overheads in terms of memory usage (since SRAM is available in larger amounts than TCAM) and operation costs (since SRAM is also more energy-efficient than TCAM). Furthermore,

FELIX's two-stage forwarding model enables it to minimize the usage of memory by only storing an alternative entry for a given failure scenario when it differs from the normal entry.

Similar to the division of responsibility between control and data planes in FELIX, DDC (LIU et al., 2013a) proposes moving the responsibility for network connectivity to the data plane. In a nutshell, DDC builds a Directed Acyclic Graph (DAG) for each destination and forwards packets based on this DAG and their incoming ports. Whenever failures strike the network, switches progressively recompute the DAG using a link-reversal algorithm triggered by packets arriving on unexpected ports. DDC guarantees reachability under any failure scenario as long as there exists a viable path. Alternative paths are computed in reaction to failures and there is no guarantee that these will be the shortest paths within the network. In contrast, in FELIX, the control plane is recruited to compute these paths ahead of time and can be given performance objectives for paths (such as using the shortest path). Moreover, data plane devices are programmed with a lightweight protocol to enable direct coordination to reroute around failures.

Considering the taxonomy proposed by Chiesa et al. (2020), alternative path setup in FELIX can be classified as *pre-planned* since the routing application plans for failure scenarios in advance. The recovery scope is *local* since it provides the shortest-non-failed paths around each of the network failures. Recovery resources are *shared* among primary and alternative paths. In conclusion, FELIX pushes the boundaries of failure recovery in three areas: (a) efficient planning ahead for many failure scenarios, (b) sensible use of data plane resources to cache alternative forwarding entries, and (c) control-plane-independent rerouting coordination in the data plane.

5.6 Additional Remarks

In this section, we discuss additional topics related to FELIX's design and performance.

Brown-Field Deployment

Can FELIX be deployed on SDN networks were not all switches are programmable? The short answer is yes, FELIX can be deployed partially on such networks. However, legacy OpenFlow-enabled switches restrict the space of failure scenarios that could be dealt with in the most efficient way by FELIX, i.e., by data-plane coordination only. In

those deployment cases, the best approach would be to use a hybrid of FELIX and the Pre-Compute SDN approach. The routing application can plan for failure scenarios normally and cache all forwarding tactics in the control plane but only installs in the data plane the tactics that can be implemented successfully without having to rely on the legacy switches to take any rerouting action. Whenever the data plane cannot deal with a failure independently, it will fallback to the control plane, which can promptly send the cached alternative forwarding entries.

Minimal Memory Usage

In Section 5.4.3, we have shown that FELIX makes judicious use of data plane device memory, using at most $2\,\%$ of the SRAM currently available in programmable hardware to fully protect a large network (i.e., with more than a thousand nodes and sixteen thousand links). However, one might still ponder whether the memory overhead could be further reduced. With that in mind, we sketch algorithms for forwarding entry minimization and briefly discuss the overall gains they achieve and their associated costs. Our main idea for reducing the number of entries is to group failure scenarios together and have one single forwarding tactic to handle all failures in each group. The set of feasible failure groups that can be formed in an arbitrary network is constrained by the reachability in each failure scenario. More specifically, any pair of nodes that can reach each other in any one of the scenarios in a group has also to be mutually reachable in the group scenario. Given this constraint, a strawman solution to minimizing the number of entries would be to analyze every possible combination of scenario groupings. However, this solution is not feasible since the number of possible groupings is a permutation on all of the possible failure scenarios in a network. Another solution would be to start with one group for each single element failure and greedily merge pairs of groups (e.g., considering the maximum decrease in number of entries) until no more groups can be joined (i.e., due to the reachability constraint). Unfortunately, this solution is still costly for medium to large topologies. Finally, we propose a probabilistic opportunistic merging procedure that starts with one grouping for each single failure scenario and at each iteration randomly picks two groups and merges them whenever the reachabiliity constraint is satisfied and the number of entries is reduced. The procedure stops when no two groups can be merged without violating the reachability guideline or doing so does not reduce the number of necessary entries.

We implemented a prototype of the opportunistic merging (OM) procedure. Ta-

ble 5.4 summarizes our findings. Regarding memory usage, the OM procedure was able to reduce the total number of forwarding entries in about $7\,\%$ in the selected WAN topologies. For the DCN, OM savings start at around $64\,\%$ for the smallest topology and quickly grows as the topologies become larger. We observe that for both types of network evaluated, more densely-connected topologies tend to yield higher savings, which is expected since there are more alternative paths to use before reachability is violated.

Table 5.4: Summary of the results for the opportunistic merging memory-minimization procedure.

	Alt Entries			**Path**
Network	**Total**	**Savings**	**Runtime**	**Stretching**
FT8	10133	64.18%	5.951s	1.0
FT16	244502	73.26%	251.991s	1.0
FT32	-	-	TLE	-
BC	6452	6.79%	47.914s	1.9
CG	184434	8.55%	202.492s	1.4
SL	588902	7.45%	601.022s	1.1

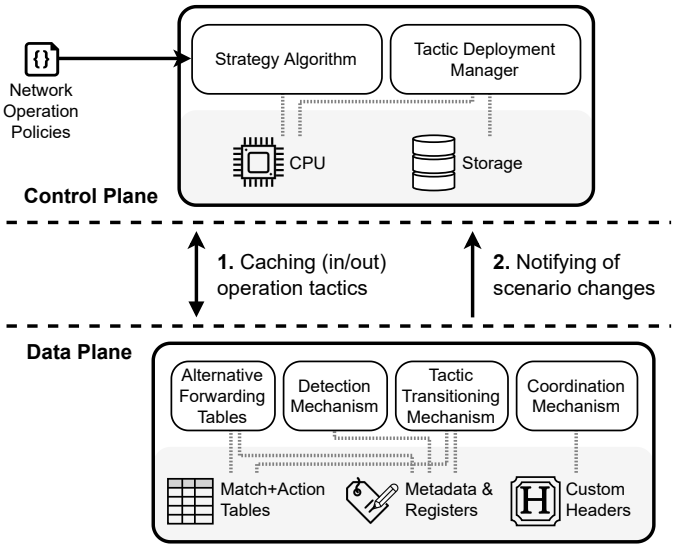
We also observe that the total runtime to achieve these memory savings grows substantially for larger and denser topologies. For example, The OM procedure did not finish within a 2-hour time limit for FT32. We highlight here that the OM procedure was designed seeking to gauge the amount of memory savings that could be expected from such procedure and leave the exploration of more time-efficient procedures as future work. Finally, any approach that seeks to group failure scenarios to reduce device memory costs may introduce path stretching in the network. From our results, we find that path stretching is independent of the network size. We observe, however, that DCN topologies are subject to lower stretching due to the availability of many equal-cost paths. For all networks, the OM procedure kept path stretching below 2 times in the median case.

5.7 On the Generality of the Strategy-Tactic Paradigm

In this chapter, we introduced the Strategy-Tactic (ST) paradigm for network operation, which is a fruit of our lessons learned with designing FELIX. This new paradigm plays on the strengths of the emerging software-defined networking architecture with programmable data planes. With FELIX, we instantiate one example application that shows the benefits of the ST paradigm: data-plane timescale reactions with control-plane deci-

sions based on a global understanding of the network. In this section, we argue that this paradigm could be applied to other network operation applications. We begin by briefly revisiting and presenting the Strategy-Tactic paradigm by way of Figure 5.8, which illustrates the main components of this paradigm. In the control plane, the Strategy Algorithm uses the available compute capacity to consider different potential network operations scenarios. More importantly, this algorithm can create operation tactics, each to handle a specific scenario in the best possible way, i.e., satisfying the network operation policies. Furthermore, the control plane servers also have ample storage capacity, enabling them to cache a large number of these tactics, which are handled by the Tactic Deployment Manager. In the data plane, programmable forwarding devices can be programmed with Alternative Forwarding Tables, which are populated by the control plane to implement the pre-computed operation tactics. Moreover, a set of simple yet powerful mechanisms and protocols enables these devices: (*i*) to detect whenever the network and traffic transition between different operation scenarios (Detection Mechanism); (*ii*) to quickly move to the appropriate tactic upon detection (Tactic Transitioning Mechanism); and (*iii*) to keep the active tactic consistent across the entire network (Coordination Mechanism). All of these mechanisms make use of the new constructs (*e.g.*, match+action tables, metadata, registers, custom headers and parsing) made available with the emergence of data plane programmability. Next, we discuss the main challenges and design decisions faced when instantiating these types of applications according to the Strategy-Tactic paradigm.

Figure 5.8: Main components of the Strategy-Tactic paradigm.



Design Challenges

The first challenge in instantiating a network operation task in the ST paradigm is that of *exploring the space of possible operation scenarios*. Given an arbitrary network management task, there are usually many network and traffic operation scenarios that need to be considered and catered to. For example, as mentioned earlier in the chapter, in the case of FELIX, the total number of possible failure scenarios is given by $\sum_{i=1}^{n} \binom{n}{i} = 2^n - 1$, where n is the number of elements in the network subject to a failure (e.g., links and switches). Consequently, new algorithms may need to be designed to explore the space of possible scenarios in a clever way, in the best case, preparing for a scenario just in time to handle it but before it actually comes to fruition. This is important to avoid both wasting compute power to plan for highly improbable scenarios as well as saturating resources to store the resulting tactics. In FELIX, based on the observed behavior of failures in recent measurement studies, the strategic application plans for the scenarios with one more failure than the current one, efficiently constraining the scenario search space at any given point.

Comparable to the previous challenge, another is that of *computing many tactics efficiently*. In the traditional implementation, for most operation tasks like failure rerouting, the control plane only computes one "tactic" at a time. In other words, the control plane only computes the forwarding entries necessary to handle the traffic given the current state of the network. When instantiating applications to run these tasks in the ST paradigm, many tactics need to be pre-computed at a time. In view of that, it becomes important to design algorithms that can compute multiple tactics in parallel to minimize the pre-planning runtime. In the context of FELIX, multiple instances of Algorithm 8, which plans for a single failure scenario, can be run in parallel.

After computing the tactics, a related challenge is that of *efficiently caching these tactics in the data plane devices*. Modern programmable forwarding devices have limited memory to implement match-action tables, thus, they may not have enough space to cache all of the necessary tactics. This challenge calls for the compression of tactics. There are two types of compression for a set of operation tactics. The first is that of minimizing the space required by each tactic independently. In FELIX, this is done in Algorithm 8 by storing only those forwarding entries that differ from the ones in the normal operation scenario. This type of compression is lossless since each tactic remains the same, and may be sufficient by itself to meet memory constraints, as observed in Section 5.4.3. The second type is that of tactic grouping, where multiple compatible tactics are combined

into one, minimizing the number of distinct tactics and forwarding entries. This type of compression can create side effects (such as path stretching, §5.6) due to the merging of tactics and is usually more time consuming than the first type.

Moving to challenges related to the runtime of the application, a natural challenge is that of *detecting changes in the network*. Since forwarding devices compose the data plane of a network and are responsible for handling traffic, they are in a prime position to detect when there is a change in the operation scenario. With the proposal of programming languages such as P4, the idea is for programs to have access to metadata and counters regarding the underlying hardware and the packets (along with their respective flows) being processed. However, the necessary information for a particular monitoring task may no be readily available and require some preprocessing. Furthermore, the type of computation enabled on these devices is still constrained (e.g., no support for division, floating-point operations), limiting the scope of detection mechanisms that can be devised. In the case of FELIX, the *port_status* field is not currently available as standard metadata in our evaluated platforms, so additional work was required to make such information available to our P4 programs.

Another aspect to be considered in that context is when to *transition between tactics*, or in other words, in which type of network event to focus on. This is necessary since even a single operation task may have multiple distinct events related to it. For example, when dealing with failures, there are events such as links going down (or coming back up), devices going offline (or coming back online), links flapping up between states, etc. As an example, in the design of FELIX, for simplicity, we focused the detection mechanism on the type of events that are clearly observable by forwarding devices, i.e., port down and port up events, leaving more involved events to appropriate device-operation-level mechanisms. Our experiments confirmed our expectation that this simplification is reasonable and effective in minimizing packet losses due to equipment failures. A similarly limited scope may need to be found for different operation tasks to keep mechanisms simple and effective.

One final important design decision to discuss is that of how to *coordinate reaction to scenario changes across forwarding devices in the network*. More specifically, there is the need to identify which subset of network devices need to be aware of a certain scenario change and which subset is necessary to implement the respective tactic. Depending on these subsets, different coordination patterns may be applied. In the case of FELIX, for each individual failure scenario, not all devices are involved in implementing

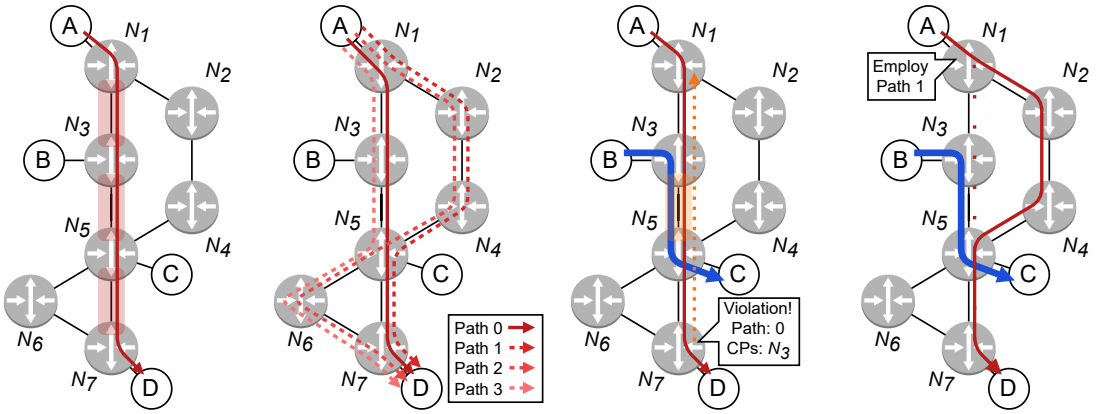
the necessary detours. However, any and all devices in the network, to be able to transition to the appropriate tactic upon locally observed failure, need to be aware of all other ongoing failures. Consequently, the designed coordination protocol broadcasts failure notifications to all forwarding devices, implementing what is effectively one network-wide tactic. We note, nevertheless, that there may exist situations where not all devices need to be aware of a certain event and multiple tactics (applied selectively to a subset of flows) may be in use at the same time. Figuring out what is the most appropriate coordination model is, thus, one of the challenges involved in instantiating an operation task to the ST paradigm. To continue our discussion, in the following, we present a case study of instantiating an operation task in the Strategy-Tactic paradigm.

Case Study

In this case study, we return our focus to performance impairing events (i.e., contentions) as previously described in Chapter 4 and seek to reroute traffic to avoid these events. Consider a network serving flows of interest with associated SLOs (along with other flows). Each flow of interest has an assigned primary path inside the network, i.e., the best path out of the ones meeting operator-defined routing policies. With that in mind, whenever ongoing contentions in the primary path of a flow of interest cause its SLO to be violated, it may be desirable for the flow to take an alternative path that avoids such contentions. Next, we start by describing a system (henceforth referred to as INTREACT) modeled after the Strategy-Tactic paradigm to perform SLO- and contention-aware rerouting at data-plane timescales. Afterwards, we contrast INTREACT's design with that of FELIX, describing the different decisions made to deal with the aforementioned instantiation challenges.

To further define the objective of INTREACT, we present an example scenario shown in Figure 5.9, in which a Red flow between endpoints A and D has an associated bandwidth SLO. Figure 5.9a depicts the primary path for the flow. Regarding this path, there are three possible points of contention that may impact the performance of the Red flow, namely the hops from N_1 to N_3, from N_3 to N_5, and from N_5 to N_7. Furthermore, alternative paths for the Red flow are shown in Figure 5.9b, along with the primary path (Path 0). Path 1 (right-side detour through nodes N_2 and N_4) enables the flow of interest to avoid contentions in either nodes N_1 or N_3, while Path 2 (left-side detour through node N_6) enables it to avoid contentions leaving in N_5. Finally, Path 3 (zigzag detour) enables the Red flow to bypass contentions in all the hops in the primary path. Given

Figure 5.9: Steps to reroute around a shared-risk failure.



this scenario, the main objective of INTREACT is to, at any given point in time, forward packets of the Red flow via a path that avoids all ongoing contentions.

INTREACT is designed in the following way, in line with the architecture in Figure 5.8. First, the Strategy Algorithm in the control plane computes the alternative paths avoiding possible points of contention for the flows of interest. In our example scenario, this algorithm would compute Paths 0 through 3 in Figure 5.9b. The alternative paths define distinct forwarding tactics to deal with contentions wherever they arise in the primary path of the Red flow. These paths are cached in the data plane by the Tactic Deployment Manager. During the normal operation of the network (i.e., no contentions) Red packets are forwarded normally via the primary path (Tactic Zero). Furthermore, INTREACT employs INTSIGHT to detect SLO violations and contention points in the path of the flow. In Figure 5.9c, a concurrent Blue flow is shown to cause a contention in the link between nodes N_3 and N_5 and consequently leads to an SLO violation for the Red flow.

Whenever a violation is detected by INTREACT (via INTSIGHT), one of the alternative tactics is put in practice to circumvent contentions and restore adequate performance. Figures 5.9c and 5.9d exemplify this transition procedure. First, the egress node N_7 detects the violation and sends a notification to the ingress node for the flow, in this case node N_1 (Fig. 5.9c). This notification includes the `PathID` (i.e., path identifier) and `ContentionPts` (i.e., contention points) for the flow as monitored by INTSIGHT (among other information). Second, with the notification in hand, node N_1 decides (using a custom match+action table) on the appropriate tactic to use considering the detected contention points in the path (Fig. 5.9d). In the case of this example, Path 1 is chosen since it avoids the saturated link between N_3 and N_5. From this point, node N_1 will mark the flow's packets (using an extra field in the INTSIGHT telemetry header) such that devices

down the path know to use Path 1. Finally, besides that, node N_1 will also periodically (at the beginning of each INTSIGHT epoch) send query packets via the primary path to check whether the contention has ended, in which case it can transition the Red flow back to the primary path.

Next, we contrast INTREACT with FELIX. Both systems have a similar high-level goal: they seek to route traffic via paths that avoid troubled network elements. Still, there are differences between their events of interest, in the manner that these events happen, and how they are best handled that elicit distinct design decisions. First, events of interest in FELIX constitute failures and those need to be avoided at all costs to stop packet loss, while in INTREACT the events are contentions and they should be avoided only if they cause SLO violations and there are alternative paths that provide the adequate performance. This results in necessary adjusts having to be made to the Strategy Algorithm. In FELIX, elements are removed from the network representation graph to compute alternative paths, whereas in INTREACT one should adjust the weight of the affected elements to signal that its use is undesirable but still leave it as an option for the case of no other appropriate path being found. We believe that exploring the space of possible scenarios can be simplified since, from our experience with the REPETITA dataset (GAY; SCHAUS; VISSICCHIO, 2017), paths tend to have an average length of around 5 hops, restricting the number of possible contention scenarios for each flow. Furthermore, the generally low connectivity degree of nodes imposes further constraints on the number of available alternative paths. This restriction would also enable naturally compressing forwarding tables since a single path may be used to circumvent many contention points at once.

As a second example of distinct design decisions, the Detection Mechanisms for INTREACT and FELIX are naturally different since they deal with distinct events. INTREACT employs INTSIGHT to detect violations and contentions. Third, and for the same reason, the Tactic Transition Mechanism in INTREACT considers the current path and contention points of the flow to transition, instead of the port status. Finally, contention events are generally more frequent and ephemeral than failures. Contentions may last a short time, impact flows unevenly, and even cause other contentions. Consequently, it is hard to keep a single current tactic consistent across the network. INTREACT applies flow-aware tagging-based forwarding, enabling it to have multiple tactics in use at the same time. Devices forward packets based on how they are marked by the ingress node, making tactic coordination part of the forwarding procedure.

In this section, we discussed the generality of the Strategy-Tactic paradigm. We

also presented a case study that indicates its applicability to enable networks to react to performance impairing events at data plane timescales. We leave the implementation of a system according to INTREACT's high-level design as future work.

5.8 Chapter Summary

In this chapter, we proposed FELIX, a novel approach for network resiliency to link and device failures that builds on top of programmable network devices to operate at data-plane speeds and uses optimal alternative paths. FELIX provides mechanisms to enable data plane devices to handle failures locally, as well as a lightweight protocol that allows these devices to coordinate network-wide rerouting around failures. As mentioned, the data plane can reconfigure forwarding at the time of failure by itself, without the need for control plane intervention, which significantly reduces network downtime and packet drops. We also proposed the Strategy-Tactic paradigm, which seeks to generalize how different operation tasks can be implemented to perform at data-plane timescales. We evidence its generality by sketching INTREACT, a system for quickly and effectively reacting to network contentions and SLO violations by migrating flows of interest to alternative paths.

6 FINAL CONSIDERATIONS

In this chapter, we present the conclusions and contributions obtained from work developed in the context of this thesis. We also outline directions for future work. We conclude the chapter by presenting the achievements obtained during this research.

6.1 Conclusions

The work conducted throughout this thesis suggests that our hypothesis that "In-band Network Telemetry (INT) along with Data Plane Programmability (DPP) can successfully be applied to monitor and operate networks with per-packet granularity as well as practicable overheads" is correct. As discussed next, the work provided us with satisfactory answers to our research questions, namely: (1) how can INT collection actions be orchestrated across devices in a network to maximize measurement quality while minimizing network and traffic overheads?; (2) can monitoring data be pre-processed or consolidated by forwarding devices before being reported in a way to further reduce overhead and with no loss to measurement quality?; and (3) can part of the analysis and reaction logic (traditionally placed in the control plane) be offloaded to the data plane to enable detecting and reacting to network problems in shorter timescales?

As the first piece of work toward improving network monitoring by leveraging the flexibility introduced by data plane programmability, we explored strategies to orchestrate network-wide In-band Network Telemetry (INT). We began our study by understanding what are the main constraints that apply to INT, as well as what are the main performance impacting factors associated with it. Next, we formalized the problem under research as an optimization model, the In-band Network Telemetry Orchestration (INTO) problem. We introduced two variations of the INTO problem (Concentrate and Balance) focusing on different aspects of the optimization solution space. INTO Concentrates focuses on minimizing the number of flows participating in INT actions. INTO Balance focuses on minimizing the average telemetry load across flows. We presented mathematical programming models to solve both of the optimization problems. We also proved that they belong to the NP-Complete class of problems by way of polynomial computing time verifiers and the reduction of known NP-Complete problems. Through our evaluation using realistic network setups, we confirmed that solving both variations of the problem takes considerable time.

To address the scalability problem related to optimally solving the INTO problem, we designed two polynomial-time heuristic algorithms. Through our evaluation, we observed that the proposed heuristics produce close to optimal solutions up to three orders of magnitude faster than mathematical programming models. Compared to the optimal solution of each instance of the problem, the heuristic for INTO Concentrate (CH) produced solutions using at most 49 additional flows. The heuristic for INTO Balance (BH) assigned at most five additional items to be transported by a flow. Thus, we observed that both of the proposed heuristics generate high-quality solutions to the problem. Continuing our evaluation, we also observed that CH performs best in minimizing the overhead on forwarding devices and monitoring sinks, making it the most scalable of the strategies and the most recommended for medium to large networks. In turn, BH achieves the best distribution of the telemetry load among forwarding devices and links in a network, making it the best option for monitoring networks serving traffic that is highly sensitive to packet size changes.

Next, in view of the additional capabilities enabled by data plane programmability, we designed and implemented INTSIGHT, a novel system for monitoring network performance SLOs related to both bandwidth and end-to-end delay guarantees. As part of this system, we devised efficient data plane procedures that gradually compute path-wise metadata (including paths, contention points, end-to-end delays, and provided bandwidth). Furthermore, we were also able to create mechanisms that take the aforementioned path-wise metadata into consideration to detect SLO violations directly in the data plane. In our evaluation, all of these mechanisms integrated into INTSIGHT made it possible to detect SLO violations as well as other performance impairing events (e.g., microbusts) in a manner that is both fine-grained and timely. Our evaluation further showed that INTSIGHT uses monitoring bandwidth sparingly, generating up to two orders of magnitude fewer reports than the state-of-the-art approach, and requires practical amounts of device memory, header space, and compute resources from the programmable forwarding devices.

Continuing our investigation on the frontiers of programmable data plane capabilities, we also developed FELIX, a system that quickly reroutes traffic in response to network failure events. FELIX is composed of elements such as a just-in-time alternative routing algorithm, a tactic-based packet forwarding pipeline, an event of interest (failure) detection mechanism, and a tactic coordination protocol. As part of our work on FELIX, we also proposed the Strategy-Tactic paradigm, which enables rerouting systems to react

to events of interest in data-plane timescales by following tactics pre-computed by control plane (while taking networking policies into account). FELIX was shown, in our evaluation, to substantially reduce the downtime observed by traffic due to equipment failures, since it operates at such small timescales. When compared to existing SDN approaches, FELIX presents average downtime that is several orders of magnitude lower, since the existing approaches are slowed down by communication delays between control and data planes. Moreover, we have shown that FELIX approach scales well with network size, making judicious use of data plane resources (e.g., about 2% memory usage for the largest evaluated network). Finally, by way of a case study, we showed that FELIX's approach to rerouting (the Strategy-Tactic paradigm) can be adapted to handle different events of interest and with different deployment scopes. We discuss the challenges involved in this adaptation process and the main necessary design decisions.

We conclude by highlighting that the contributions of this thesis touch both theoretical and practical aspects of network management. We mathematically formalized problems and proposed architectural paradigms as well as designed heuristic algorithms and monitoring and operation systems. Throughout, we give special focus to experimentation by considering real world scenarios. Our hope is that with this transversal investigation of the hypothesis, the research questions and their related challenges, we were able to not only tackle important problems, but also sensibly inform future work in the area.

6.2 Directions for Future Research

We envision that this research can be extended in several ways in future investigations. In the context of In-band Network Telemetry orchestration, it may be interesting to explore other assignment and reporting modes as well as optimization goals and guarantees. Regarding the assignment mode, one interesting line of investigation is to allow some redundancy in assignment, with multiple flows being assigned to the same telemetry item. This could improve the measurement granularity and accuracy at the cost of higher bandwidth requirements. With regards to reporting mode, in our work, we considered the INT Embed Data operation mode, in which the telemetry data collected by each packet is reported at most once. One could envision other modes to be used, such as, for example, one where each packet collects telemetry data until it is full, reporting that data, clears the telemetry header and goes back to collecting more data. One challenge in this scenario is how to reconcile the multiple reports generated along the way of pack-

ets. Another clear potential area for future work is exploring other optimization goals and algorithms for the assignment problem. With relation to exploring other goals, it could enable achieving other compromises between coverage, telemetry load, flow usage, and information correlation and freshness. Related to other algorithms, better or stricter guarantees on optimality (or closeness to it) could be given with approximation algorithms, for example.

Another relevant research direction consists in further exploring other applications for the path-wise telemetry computation model in the data plane introduced with INTSIGHT. More specifically, we believe this computation model could be adapted to implement runtime verification of operator-defined network properties (or invariants), such as reachability between end-points, waypointing, path preference and disjunction, and loop freedom. We observe that INTSIGHT's algorithm for path tracing naturally lends itself to the types of checking required to verify these properties. For example, a waypointing property can be verified by looking at the computed `PathID` field of the telemetry header of packets. We are currently exploring this opportunity for extending INTSIGHT as part of an ongoing master dissertation work.

Related to FELIX, we intend to investigate other operation and routing tasks that could be modeled following the Strategy-Tactic (ST) paradigm. We discussed one such modeling with the sketching of INTREACT, a system to respond to SLO violations by rerouting the traffic of interest out of paths with contentions. As future work, we intend to mature this initial concept of INTREACT by further considering the involved challenges, introducing additional cases based on real world scenarios, and developing functional prototypes. Furthermore, we also envision the ST paradigm could also be used to implement the following additional operation tasks. Regarding network property verification, the extension of INTSIGHT described in the previous paragraph (to detect network property violations) could be applied to enable the network to respond appropriately to property violations. For example, it could trigger route adjustments or even packet dropping when a waypointing violation represents a security breach. Another operation task could be contemplated by designing a system to help mitigate Distributed Denial-of-Service (DDoS) attacks via rerouting mechanisms. The idea would be for the traffic to be selectively rerouted via additional inspection and mitigation systems whenever the network believes it is under attack. In our research group, we have carried out work related to the detection and mitigation of DDoS attacks (as listed in the next section) and have built mechanisms running in the data plane that help with the detection process. In addition, and finally,

another interesting direction for future work is that of investigating the implementation of machine learning models directly in the data plane that could be used instead of deep packet inspection middleboxes, offloading as much of the DDoS attack response capability as possible to the data plane. We deem this line of work crucial to release the full potential of data plane programmability.

6.3 Achievements

The work carried out in the context of this Ph.D. course has led to the publication of the following peer-reviewed papers:

- **Data Plane Programmability Beyond OpenFlow: Opportunities and Challenges for Network and Service Operations and Management.**
  Weverton Luis da Costa Cordeiro, Jonatas Adilson Marques, and Luciano Paschoal Gaspary.
  Journal of Network and Systems Management (JNSM), 2017.
- **Explorando Estratégias de Orquestração de Telemetria em Planos de Dados Programáveis. (Best Paper Award)**
  Jonatas Adilson Marques and Luciano Paschoal Gaspary.
  Brazilian Symposium on Computer Networks and Distributed Systems (SBRC), 2018.
- **An optimization-based approach for efficient network monitoring using in-band network telemetry.**
  Jonatas Adilson Marques, Marcelo Caggiani Luizelli, Roberto Irajá Tavares da Costa Filho, and Luciano Paschoal Gaspary.
  Journal of Internet Services and Applications (JISA), 2019.
- **IntSight: Diagnosing SLO Violations with In-band Network Telemetry.**
  Jonatas Adilson Marques, Kirill Levchenko, and Luciano Paschoal Gaspary.
  Proceedings of the 16th International Conference on Emerging Networking Experiments and Technologies (CoNEXT), 2020.

In addition to the aforementioned primary outcomes of this work, we actively contributed to and further coauthored other studies on correlated problems in the context of Software-Defined Networking, Data Plane Programmability, and network monitoring and operation. These publications are listed next.

- **Identificação de Fluxos Elefantes em Redes de Ponto Troca de Tráfego com Suporte à Programabilidade P4.**
  Marcus Vinicius Brito da Silva, Jonatas Adilson Marques, Luciano Paschoal Gaspary, and Lisandro Zambenedetti Granville.
  Brazilian Symposium on Computer Network and Distributed Systems (SBRC), 2018.
- **Offloading Real-time DDoS Attack Detection to Programmable Data Planes. (Best Student Paper Award)**
  Ângelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary.
  IFIP/IEEE International Symposium on Integrated Network Management (IM), 2019.
- **Identifying Elephant Flows using Dynamic Thresholds in Programmable IXP Networks.**
  Marcus Vinicius Brito da Silva, Jonatas Adilson Marques, Luciano Paschoal Gaspary, and Lisandro Zambenedetti Granville.
  Journal of Internet Services and Applications (JISA), 2020.
- **Euclid: A Fully In-network, P4-based Approach for Real-time DDoS Attack Detection and Mitigation.**
  Alexandre da Silveira Ilha, Ângelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary.
  IEEE Transactions on Network and Service Management (TNSM), 2020.
- **BUNGEE: An Adaptive Pushback Mechanism for DDoS Detection and Mitigation in P4 Data Planes.**
  Libardo Andrey Quintero González, Lucas Castanheira, Jonatas Adilson Marques, Alberto Schaeffer-Filho, and Luciano Paschoal Gaspary.
  IFIP/IEEE International Symposium on Integrated Network Management (IM), 2021.
- **A Systematic Review on Distributed Denial of Service Attack Defense Mechanisms in Programmable networks.**
  Bruno L Dalmazo, Jonatas A Marques, Lucas R Costa, Michel S Bonfim, Ranyelson N Carvalho, Anderson S da Silva, Stenio Fernandes, Jacir L Bordim, Eduardo Alchieri, Alberto Schaeffer-Filho, Luciano Paschoal Gaspary, Weverton Cordeiro.
  International Journal of Network Management (IJNM), 2021.

REFERENCES

ALEXANDER, D. S. et al. The SwitchWare active network architecture. **IEEE Network**, v. 12, n. 3, p. 29–36, May 1998. ISSN 0890-8044.

ALEXANDER, D. S. et al. Active bridging. In: **Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: ACM, 1997. (SIGCOMM '97), p. 101–111. ISBN 0-89791-905-X.

ALIZADEH, M. et al. Conga: Distributed congestion-aware load balancing for datacenters. In: **Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2014. (SIGCOMM '14), p. 503–514. ISBN 978-1-4503-2836-4. Available from Internet: <http://doi.acm.org/10.1145/2619239.2626316>.

BALAKRISHNAN, H. Mind the app! SIGCOMM Lifetime Achievement Award (SIGCOMM'21 Keynote). 2021. Available from Internet: <https://conferences.sigcomm.org/sigcomm/2021/files/SIGCOMM-Award-Talk-2021.pdf>.

BALL, T.; LARUS, J. R. Efficient path profiling. In: **Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture**. Washington, DC, USA: IEEE Computer Society, 1996. (MICRO 29), p. 46–57. ISBN 0-8186-7641-8. Available from Internet: <https://dl.acm.org/doi/10.5555/243846.243857>.

BAREFOOT NETWORKS. **Tofino: World's Fastest P4-Programmable Ethernet Switch ASICs**. 2020. Available from Internet: <https://barefootnetworks.com/products/brief-tofino/>.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2656877.2656890>.

BOSSHART, P. et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In: **Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM**. New York, NY, USA: ACM, 2013. (SIGCOMM '13), p. 99–110. ISBN 978-1-4503-2056-6. Available from Internet: <http://doi.acm.org/10.1145/2486001.2486011>.

BROCKNERS, F. et al. **Requirements for In-situ OAM**. [S.l.], 2017. Accessed on May, 2022. Available from Internet: <https://datatracker.ietf.org/doc/html/draft-brockners-inband-oam-requirements-03>.

BRUNNER, M.; STADLER, R. The impact of active networking technology on service management in a telecom environment. In: **Integrated Network Management VI. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management. (Cat. No.99EX302)**. [S.l.: s.n.], 1999. p. 385–400.

CALVERT, K. L. et al. Directions in active networks. **IEEE Communications Magazine**, v. 36, n. 10, p. 72–78, Oct 1998. ISSN 0163-6804.

CASADO, M. et al. Ethane: Taking control of the enterprise. In: **Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications**. New York, NY, USA: ACM, 2007. (SIGCOMM '07), p. 1–12. ISBN 978-1-59593-713-1.

CASADO, M. et al. Sane: A protection architecture for enterprise networks. In: **Usenix Security**. [S.l.: s.n.], 2006.

CASCONE, C. et al. Fast failure detection and recovery in sdn with stateful data plane. **International Journal of Network Management**, v. 27, n. 2, p. e1957, 2017. E1957 nem.1957. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.1957>.

CASE, J. D. et al. **Simple Network Management Protocol (SNMP)**. IETF, 1989. RFC 1098. (Request for Comments). Available from Internet: <https://tools.ietf.org/html/rfc1098>.

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: ACM, 2019. (CoNEXT '19), p. 15–29. ISBN 9781450369985. Available from Internet: <https://doi.org/10.1145/3359989.3365408>.

CHENG, G.; YU, J. Adaptive sampling for openflow network measurement methods. In: **Proceedings of the International Conference on Future Internet Technologies**. New York, NY, USA: ACM, 2017. (CFI'17), p. 4:1–4:7. ISBN 978-1-4503-5332-8. Available from Internet: <http://doi.acm.org/10.1145/3095786.3095790>.

CHIESA, M. et al. A survey of fast recovery mechanisms in the data plane. **TechRxiv Preprint**, TechRxiv, 2020. Available from Internet: <https://www.techrxiv.org/articles/preprint/Fast_Recovery_Mechanisms_in_the_Data_Plane/12367508>.

CHIESA, M. et al. Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst. In: **Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies**. New York, NY, USA: Association for Computing Machinery, 2019. (CoNEXT '19), p. 1–14. ISBN 9781450369985. Available from Internet: <https://doi.org/10.1145/3359989.3365410>.

CHOLE, S. et al. drmt: Disaggregated programmable switching. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 1–14. ISBN 978-1-4503-4653-5.

CHOWDHURY, S. R. et al. Payless: A low cost network monitoring framework for software defined networks. In: **Proceedings of the IEEE Network Operations and Management Symposium**. [S.l.: s.n.], 2014. (NOMS'14), p. 1–9. ISSN 1542-1201.

CISCO. **Cisco IOS NetFlow**. 2005. Http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

CISCO. **In-band OAM (iOAM)**. S.a. Accessed on May, 2022. Available from Internet: <https://github.com/CiscoDevNet/iOAM>.

CLAISE, B.; TRAMMELL, B.; AITKEN, P. **Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information**. IETF, 2013. RFC 7011. (Request for Comments). Available from Internet: <https://tools.ietf.org/html/rfc7011>.

CORDEIRO, W. L. d. C.; MARQUES, J. A.; GASPARY, L. P. Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management. **Journal of Network and Systems Management**, v. 25, n. 4, p. 784–818, Oct 2017. ISSN 1573-7705. Available from Internet: <https://doi.org/10.1007/s10922-017-9423-2>.

CORMODE, G. et al. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In: **Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2005. (SIGMOD '05), p. 25–36. ISBN 1-59593-060-4. Available from Internet: <http://doi.acm.org/10.1145/1066157.1066161>.

DABBAGH, M. et al. Software-defined networking security: pros and cons. **IEEE Communications Magazine**, v. 53, n. 6, p. 73–79, June 2015. ISSN 0163-6804.

DANG, H. T. et al. Paxos made switch-y. **SIGCOMM Comput. Commun. Rev.**, v. 46, n. 2, p. 18–24, may 2016.

FEAMSTER, N. et al. The case for separating routing from routers. In: **Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture**. New York, NY, USA: ACM, 2004. (FDNA '04), p. 5–12. ISBN 1-58113-942-X.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The Road to SDN: An intellectual history of programmable networks. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 44, n. 2, p. 87–98, abr. 2014. ISSN 0146-4833.

FOSTER, N. et al. Frenetic: A high-level language for openflow networks. In: **Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow**. New York, NY, USA: ACM, 2010. (PRESTO '10), p. 6:1–6:6. ISBN 978-1-4503-0467-2.

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.

GAY, S.; SCHAUS, P.; VISSICCHIO, S. REPETITA: repeatable experiments for performance evaluation of traffic-engineering algorithms. **arXiv preprint arXiv:1710.08665**, 2017.

GILL, P.; JAIN, N.; NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. In: **Proceedings of the ACM SIGCOMM 2011 Conference**. New York, NY, USA: Association for Computing Machinery, 2011. (SIGCOMM '11), p. 350–361. ISBN 9781450307970. Available from Internet: <https://doi.org/10.1145/2018436.2018477>.

GOVINDAN, R. et al. Evolve or die: High-availability design principles drawn from googles network infrastructure. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: Association for Computing Machinery, 2016. (SIGCOMM '16), p. 58–72. ISBN 9781450341936. Available from Internet: <https://doi.org/10.1145/2934872.2934891>.

GRAY, R. S. Agent tcl: A flexible and secure mobile-agent system. In: **Proceedings of the Fourth Annual USENIX Tcl/Tk Workshop**. Berkeley, CA, USA: USENIX, 1996. (Fourth Annual Tcl/Tk Workshop).

GREENBERG, A. et al. A clean slate 4d approach to network control and management. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 35, n. 5, p. 41–54, oct. 2005. ISSN 0146-4833.

GROUP, P. A. W. **In-band Network Telemetry (INT) Dataplane Specification**. [S.l.], 2018.

GUDE, N. et al. Nox: Towards an operating system for networks. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 38, n. 3, p. 105–110, jul. 2008. ISSN 0146-4833.

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**. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 357–371. ISBN 978-1-4503-5567-4. Available from Internet: <http://doi.acm.org/10.1145/3230543.3230555>.

HANDIGOL, N. et al. I know what your packet did last hop: Using packet histories to troubleshoot networks. In: **Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2014. (NSDI '14), p. 71–85. ISBN 978-1-931971-09-6. Available from Internet: <https://dl.acm.org/doi/10.5555/2616448.2616456>.

HE, K. et al. Presto: Edge-based load balancing for fast datacenter networks. In: **Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 465–478. ISBN 978-1-4503-3542-3. Available from Internet: <http://doi.acm.org/10.1145/2785956.2787507>.

HEDAYAT, K. et al. **A Two-Way Active Measurement Protocol (TWAMP)**. [S.l.], 2008. Available from Internet: <https://tools.ietf.org/html/rfc5357>.

HONG, C.-Y. et al. Achieving high utilization with software-driven wan. In: **Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2013. (SIGCOMM '13), p. 15–26. ISBN 978-1-4503-2056-6. Available from Internet: <http://doi.acm.org/10.1145/2486001.2486012>.

HSU, K.-F. et al. Contra: A programmable system for performance-aware routing. In: **17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)**. Santa Clara, CA: USENIX Association, 2020. p. 701–721. ISBN 978-1-939133-13-7. Available from Internet: <https://www.usenix.org/conference/nsdi20/presentation/hsu>.

HUANG, D. Y.; YOCUM, K.; SNOEREN, A. C. High-fidelity switch models for software-defined network emulation. In: **Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: Association for Computing Machinery, 2013. (HotSDN '13), p. 43–48. ISBN 9781450321785. Available from Internet: <https://doi.org/10.1145/2491185.2491188>.

JAIN, S. et al. B4: Experience with a globally-deployed software defined wan. In: **Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2013. (SIGCOMM '13), p. 3–14. ISBN 978-1-4503-2056-6. Available from Internet: <http://doi.acm.org/10.1145/2486001.2486019>.

JEYAKUMAR, V. et al. Millions of little minions: Using packets for low latency network programming and visibility. In: **Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2014. (SIGCOMM '14), p. 3–14. ISBN 978-1-4503-2836-4. Available from Internet: <http://doi.acm.org/10.1145/2619239.2626292>.

JURCA, D.; STADLER, R. H-gap: estimating histograms of local variables with accuracy objectives for distributed real-time monitoring. **IEEE Transactions on Network and Service Management**, v. 7, n. 2, p. 83–95, June 2010. ISSN 1932-4537.

KATZ, D.; WARD, D. **Bidirectional Forwarding Detection (BFD)**. IETF, 2010. RFC 5880. (Request for Comments). Available from Internet: <https://tools.ietf.org/html/rfc5880>.

KIM, C.; LEE, J. **P4: Programming the Network Data Plane**. 2016. ACM SIGCOMM Tutorial. Accessed on May, 2022. Available from Internet: <http://conferences.sigcomm.org/sigcomm/2016/tutorial-p4.php>.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: **Proceedings of the 2015 ACM Symposium on SDN Research**. New York, NY, USA: ACM, 2015. (SOSR'15). Available from Internet: <http://git.io/sosr15-int-demo>.

KNIGHT, S. et al. The internet topology zoo. **IEEE Journal on Selected Areas in Communications**, v. 29, n. 9, p. 1765–1775, October 2011. ISSN 0733-8716.

KUMAR, A. et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In: **Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 1–14. ISBN 978-1-4503-3542-3. Available from Internet: <http://doi.acm.org/10.1145/2785956.2787478>.

KUŹNIAR, M.; PEREŠÍNI, P.; KOSTIĆ, D. What you need to know about sdn flow tables. In: MIRKOVIC, J.; LIU, Y. (Ed.). **Passive and Active Measurement**. Cham: Springer International Publishing, 2015. p. 347–359. ISBN 978-3-319-15509-8.

KVALBEIN, A. et al. Multiple routing configurations for fast ip network recovery. **IEEE/ACM Transactions on Networking**, v. 17, n. 2, p. 473–486, 2009.

LAKSHMAN, T. et al. The softrouter architecture. In: CITESEER. **Proc. ACM SIGCOMM Workshop on Hot Topics in Networking**. [S.l.], 2004. v. 2004.

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**. New York, NY, USA: Association for Computing Machinery, 2010. (Hotnets-IX). ISBN 9781450304092. Available from Internet: <https://doi.org/10.1145/1868447.1868466>.

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

LEE, M.; DUFFIELD, N.; KOMPELLA, R. R. Not all microseconds are equal: Fine-grained per-flow measurements with reference latency interpolation. In: **Proceedings of the ACM SIGCOMM 2010 Conference**. New York, NY, USA: ACM, 2010. (SIGCOMM '10), p. 27–38. ISBN 978-1-4503-0201-2. Available from Internet: <http://doi.acm.org/10.1145/1851182.1851188>.

LEE, S. et al. Proactive vs reactive approaches to failure resilient routing. In: **IEEE INFOCOM 2004**. [S.l.: s.n.], 2004. v. 1, p. 186.

LIU, J. et al. Ensuring connectivity via data plane mechanisms. In: **10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)**. Lombard, IL: USENIX Association, 2013. p. 113–126. ISBN 978-1-931971-00-3. Available from Internet: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/liu_junda>.

LIU, V. et al. F10: A fault-tolerant engineered network. In: **10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)**. Lombard, IL: USENIX Association, 2013. p. 399–412. ISBN 978-1-931971-00-3. Available from Internet: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/liu_vincent>.

MACEDO, D. F. et al. Programmable networks: From software-defined radio to software-defined networking. **IEEE Communications Surveys Tutorials**, v. 17, n. 2, p. 1102–1125, Secondquarter 2015. ISSN 1553-877X.

MARKOPOULOU, A. et al. Characterization of failures in an operational ip backbone network. **IEEE/ACM Transactions on Networking**, v. 16, n. 4, p. 749–762, 2008.

MARQUES, J.; LEVCHENKO, K.; GASPARY, L. Intsight: Diagnosing slo violations with in-band network telemetry. In: **Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2020. (CoNEXT '20), p. 421–434. ISBN 9781450379489. Available from Internet: <https://doi.org/10.1145/3386367.3431306>.

MARQUES, J.; LEVCHENKO, K.; GASPARY, L. **IntSight Repository on GitHub**. 2020. Available from Internet: <https://github.com/jonadmark/intsight-conext>.

MARQUES, J. A.; GASPARY, L. Explorando estratégias de orquestração de telemetria em planos de dados programáveis. In: **Anais do XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Porto Alegre, RS, Brasil: SBC, 2018. p. 1299–1312. ISSN 2177-9384. Available from Internet: <https://sol.sbc.org.br/index.php/sbrc/article/view/2495>.

MARQUES, J. A. et al. An optimization-based approach for efficient network monitoring using in-band network telemetry. **Journal of Internet Services and Applications**, Springer, v. 10, n. 1, p. 1–20, 2019. Available from Internet: <https://doi.org/10.1186/s13174-019-0112-0>.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1355734.1355746>.

MCKEOWN, N.; APPENZELLER, G.; KESLASSY, I. Sizing router buffers (redux). **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 49, n. 5, p. 69–74, 2019.

MOSHREF, M. et al. Dream: Dynamic resource allocation for software-defined measurement. In: **Proceedings of the 2014 ACM Conference on SIGCOMM**. New York, NY, USA: ACM, 2014. (SIGCOMM '14), p. 419–430. ISBN 978-1-4503-2836-4. Available from Internet: <http://doi.acm.org/10.1145/2619239.2626291>.

NELAKUDITI, S. et al. Fast local rerouting for handling transient link failures. **IEEE/ACM Transactions on Networking**, v. 15, n. 2, p. 359–372, 2007.

ONF. **ONF Technical Library**. 2015. Accessed on May, 2022. Available from Internet: <https://opennetworking.org/sdn-resources/technical-resources/>.

OZDAG, R. **Intel® Ethernet Switch FM6000 Series-Software Defined Networking**. [S.l.], 2012. Available from Internet: <https://people.ucsc.edu/~warner/Bufs/ethernet-switch-fm6000-sdn-paper.pdf>.

P4 ARCHITECTURE WORKING GROUP. **P4_16 Portable Switch Architecture (PSA)**. 2018. Accessed on May, 2022. Available from Internet: <https://p4.org/p4-spec/docs/PSA-v1.1.0.html>.

P4 LANGUAGEM CONSORTIUM. **Behavioral Model (BMv2)**. 2014. Accessed on May, 2022. Available from Internet: <https://github.com/p4lang/behavioral-model>.

PRIETO, A. G.; STADLER, R. A-gap: An adaptive protocol for continuous network monitoring with accuracy objectives. **IEEE Transactions on Network and Service Management**, v. 4, n. 1, p. 2–12, June 2007. ISSN 1932-4537.

QIU, K. et al. Efficient recovery path computation for fast reroute in large-scale software-defined networks. **IEEE Journal on Selected Areas in Communications**, v. 37, n. 8, p. 1755–1768, 2019.

QUINN, P.; ELZUR, U.; PIGNATARO, C. **Network Service Header (NSH)**. [S.l.], 2018. 1-39 p. Available from Internet: <https://tools.ietf.org/html/rfc8300>.

QUITTEK, J.; BRUNNER, M. Applying and evaluating active technologies in distributed management. **Journal of Network and Systems Management**, v. 11, n. 2, p. 171–197, 2003. ISSN 1573-7705.

RASLEY, J. et al. Planck: Millisecond-scale monitoring and control for commodity networks. In: **Proceedings of the 2014 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2014. (SIGCOMM '14), p. 407–418. ISBN 978-1-4503-2836-4. Available from Internet: <http://doi.acm.org/10.1145/2619239.2626310>.

REKHTER, Y.; HARES, S.; LI, D. T. **A Border Gateway Protocol 4 (BGP-4)**. RFC Editor, 2006. RFC 4271. (Request for Comments, 4271). Available from Internet: <https://www.rfc-editor.org/rfc/rfc4271.txt>.

ROUGHAN, M. Simplifying the synthesis of internet traffic matrices. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 35, n. 5, p. 93–96, oct. 2005. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1096536.1096551>.

SALIM, J. et al. **Linux Netlink as an IP Services Protocol**. RFC Editor, 2003. RFC 3549. (Request for Comments, 3549). Accessed on May, 2022. Available from Internet: <https://www.rfc-editor.org/rfc/rfc3549.txt>.

SCHOENWAELDER, J.; QUITTEK, J. **Script MIB Extensibility Protocol Version 1.1**. RFC Editor, 2001. RFC 3179. (Request for Comments, 3179). Accessed on May, 2022. Available from Internet: <https://www.rfc-editor.org/rfc/rfc3179.txt>.

SHALUNOV, S. et al. **A One-way Active Measurement Protocol (OWAMP)**. [S.l.], 2006. Available from Internet: <https://tools.ietf.org/html/rfc4656>.

SIVARAMAN, V. et al. Heavy-hitter detection entirely in the data plane. In: **Proceedings of the 2017 Conference of the ACM Symposium on SDN Research**. New York, NY, USA: ACM, 2017. (SOSR '17), p. 164–176. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3063772>.

SMITH, J. M. et al. Activating networks: a progress report. **Computer**, v. 32, n. 4, p. 32–41, Apr 1999. ISSN 0018-9162.

SONCHACK, J. et al. Turboflow: Information rich flow record generation on commodity switches. In: **Proceedings of the Thirteenth EuroSys Conference**. New York, NY, USA: Association for Computing Machinery, 2018. (EuroSys '18). ISBN 9781450355841. Available from Internet: <https://doi.org/10.1145/3190508.3190558>.

STEPHENS, B.; COX, A. L.; RIXNER, S. Scalable multi-failure fast failover via forwarding table compression. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2016. (SOSR '16). ISBN 9781450342117. Available from Internet: <https://doi.org/10.1145/2890955.2890957>.

TAMMANA, P.; AGARWAL, R.; LEE, M. Distributed network monitoring and debugging with switchpointer. In: **Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation**. USA: USENIX Association, 2018. (NSDI '18), p. 453–466. ISBN 9781931971430. Available from Internet: <https://dl.acm.org/doi/10.5555/3307441.3307480>.

TANGARI, G. et al. Decentralized monitoring for large-scale software-defined networks. In: **Proceedings of the IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. [S.l.: s.n.], 2017. p. 289–297.

TENNENHOUSE, D. L. et al. A survey of active network research. **IEEE Communications Magazine**, v. 35, n. 1, p. 80–86, Jan 1997. ISSN 0163-6804.

THOMAS, J.; LAUPKHOV, P. **Tracking Packets' Paths and Latency via INT (In-band Network Telemetry)**. 2016. 3rd P4 Workshop. Available from Internet: <https://2016p4workshop.sched.com/event/6otq/using-int-to-build-a-real-time-network-monitoring-system-scale>.

TILMANS, O. et al. Stroboscope: Declarative network monitoring on a budget. In: **Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation**. USA: USENIX Association, 2018. (NSDI '18), p. 467–482. ISBN 978-1-939133-01-4. Available from Internet: <https://www.usenix.org/conference/nsdi18/presentation/tilmans>.

TOOTOONCHIAN, A.; GANJALI, Y. Hyperflow: A distributed control plane for openflow. In: **Proceedings of the 2010 internet network management conference on Research on enterprise networking**. [S.l.: s.n.], 2010. p. 3–3.

TURNER, D. et al. On failure in managed enterprise networks. **HP Labs HPL-2012-101**, Citeseer, 2012.

TURNER, D. et al. California fault lines: Understanding the causes and impact of network failures. In: **Proceedings of the ACM SIGCOMM 2010 Conference**. New York, NY, USA: Association for Computing Machinery, 2010. (SIGCOMM '10), p. 315–326. ISBN 9781450302012. Available from Internet: <https://doi.org/10.1145/1851182.1851220>.

WETHERALL, D. Active network vision and reality: lessons from a capsule-based system. In: **Proceedings DARPA Active Networks Conference and Exposition**. [S.l.: s.n.], 2002. p. 25–40.

WETHERALL, D. J.; GUTTAG, J. V.; TENNENHOUSE, D. L. ANTS: a toolkit for building and dynamically deploying network protocols. In: **1998 IEEE Open Architectures and Network Programming**. [S.l.: s.n.], 1998. p. 117–129.

WHITE, J. E. **Telescript: The Foundation for the Electronic Marketplace**. [S.l.], 1993.

YAN, H. et al. Tesseract: A 4d network control plane. In: **NSDI**. [S.l.: s.n.], 2007. v. 7, p. 27–27.

YANG, L. et al. **Forwarding and Control Element Separation (ForCES) Framework**. RFC Editor, 2004. RFC 3746. (Request for Comments, 3746). Accessed on May, 2022. Available from Internet: <https://www.rfc-editor.org/rfc/rfc3746.txt>.

YEGANEH, S. H.; TOOTOONCHIAN, A.; GANJALI, Y. On scalability of software-defined networking. **IEEE Communications Magazine**, v. 51, n. 2, p. 136–141, 2013.

ZHANG, Q. et al. High-resolution measurement of data center microbursts. In: **Proceedings of the 2017 Internet Measurement Conference**. New York, NY, USA: ACM, 2017. (IMC '17), p. 78–85. ISBN 9781450351188. Available from Internet: <https://doi.org/10.1145/3131365.3131375>.

ZHONG, Z. et al. Failure inferencing based fast rerouting for handling transient link and node failures. In: **Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.** [S.l.: s.n.], 2005. v. 4, p. 2859–2863 vol. 4.

ZHU, Y. et al. Packet-level telemetry in large datacenter networks. **ACM SIGCOMM Computer Communication Review (CCR)**, ACM, New York, NY, USA, v. 45, n. 4, p. 479–491, aug. 2015. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2829988.2787483>.

ZILBERMAN, N. et al. NetFPGA SUME: Toward 100 gbps as research commodity. **IEEE Micro**, v. 34, n. 5, p. 32–41, 2014.

APPENDIX A — SUMMARY IN PORTUGUESE

Nos dias atuais, redes de comunicação operam sob altas expectativas de desempenho (*e.g.*, latência, largura de banda, disponibilidade), especialmente com o surgimento e proliferação de novas aplicações (*e.g.*, negociação algorítmica, telecirurgia e streaming de vídeo de realidade virtual) com arquiteturas baseadas em componentes espalhados por múltiplos nodos e hospedeiros (BALAKRISHNAN, 2021). Essas aplicações e seus usuários demandam requisitos rígidos, que para serem atendidos exigem a definição de metas claras de desempenho da rede, os chamados objetivos de nível de serviço (SLOs), além da solução de problemas que podem impedir o alcance de tais metas. Infelizmente, há uma infinidade de problemas que podem afetar a operação correta e eficiente de uma rede, desde congestionamentos de tráfego até falhas de hardware. Nesse contexto, monitorar o estado, o comportamento e o desempenho dos dispositivos de rede e seu tráfego é essencial para a operação das infraestruturas de rede atuais. No entanto, o monitoramento de rede é uma tarefa inerentemente difícil, às vezes comparada a procurar uma agulha em um palheiro (ZHU et al., 2015; GUPTA et al., 2018).

As ferramentas e técnicas existentes não são projetadas para monitorar redes e solucionar seus problemas com o nível necessário de detalhes e precisão demandado. Por exemplo, e em relação à coleta de metadados e estatísticas, as ferramentas tradicionais de monitoramento passivo (*e.g.*, SNMP (CASE et al., 1989) e NetFlow/IPFIX (CISCO, 2005; CLAISE; TRAMMELL; AITKEN, 2013)) operam em escalas de tempo longas (dezenas de segundos ou mais) e, portanto, não oferecem a granularidade adequada para detectar eventos como rajadas de tráfego de curta duração (*e.g.*, microbursts) que podem ser desastrosas para aplicações modernas. Técnicas de medição ativa (*e.g.*, ping, traceroute, OWAMP (SHALUNOV et al., 2006) e TWAMP (HEDAYAT et al., 2008)) também não fornecem resolução de tempo satisfatória, e além disso, não há garantia de que a rede roteará e priorizará os pacotes de medição da mesma forma que os pacotes de produção. O movimento consistente em direção à heterogeneidade no tratamento do tráfego (JEYAKUMAR et al., 2014; HONG et al., 2013), roteamento multi-caminhos (JAIN et al., 2013; KUMAR et al., 2015) e balanceamento de carga de *flowlets* (ALIZADEH et al., 2014; HE et al., 2015) exacerba essa limitação. Como segundo exemplo, e em relação à solução de problemas de violação de SLO, abordagens baseadas em *mirroring* de pacotes (*e.g.*, NetSight (HANDIGOL et al., 2014), Planck (RASLEY et al., 2014), Everflow (ZHU et al., 2015) e Stroboscope (TILMANS et al., 2018)) podem ajudar a dar

visibilidade para entender como os pacotes estão sendo processados na rede e encaminhados pelos dispositivos. Essas abordagens encontram seu principal desafio em manter a sobrecarga de monitoramento (ou seja, largura de banda e processamento necessários para este fim) em níveis razoáveis e ao mesmo tempo coletar dados com granularidade fina (TILMANS et al., 2018). A amostragem de pacotes, o método mais comum para enfrentar esse desafio, inerentemente leva essas abordagens a perder eventos importantes. Isso, pois decidir o que e quando amostrar é difícil.

Outro desafio no atendimento aos SLOs nos tempos modernos é a dependência de comunicação, e o atraso causado por esta, entre os mecanismos que detectam problemas e os que encontram a solução para esses problemas. Considere, por exemplo, o caso de falhas de equipamentos e seu impacto na disponibilidade da rede. As soluções existentes dependem de algum tipo de computação no plano de controle no momento da falha e a posterior reconfiguração das tabelas de encaminhamento. Calcular as novas entradas de encaminhamento para dispositivos ou, em termos mais amplos, computar a solução do problema pode levar um tempo considerável. Como o atraso na reação a falhas leva a um número significativo de quedas de pacotes, no caso geral, o atraso causado pelo uso de loops de controle para resolver problemas pode levar a uma queda substancial no desempenho e a perdas financeiras. Idealmente, o plano de dados deve ser capaz de reagir imediatamente no momento da falha. Notamos que as limitações e desvantagens apresentadas pelas ferramentas e técnicas de monitoramento existentes resultam do baixo nível de flexibilidade na definição de como os pacotes devem ser processados pelo plano de dados em redes tradicionais. Mesmo com Rede Definida por Software (SDN) e OpenFlow (MCKEOWN et al., 2008), há suporte apenas para protocolos padronizados; não há liberdade para definir protocolos e procedimentos personalizados.

A.1 Definição do Problema

Como resultado do cenário apresentado, a comunidade de redes tem buscado maior flexibilidade no plano de dados, o que culminou recentemente na proposta de programabilidade do plano de dados (do inglês *Data Plane Programmability* – DPP) (BOSSHART et al., 2013; CHOLE et al., 2017; OZDAG, 2012). O DPP remodela o cenário das redes definidas por software (*Software-Defined Networking* – SDN), permitindo que os operadores de rede reprogramem os seus dispositivos de encaminhamento mesmo após sua instalação para implantar novos protocolos de comunicação, personalizar

o comportamento da rede e, consequentemente, desenvolver e oferecer suporte a aplicações serviços inovadores. Protocolos e procedimentos de processamento de pacotes, neste novo paradigma, são definidos por meio de linguagens específicas de domínio – por exemplo, P4 (BOSSHART et al., 2014) – com suporte a abstrações para especificar cabeçalhos de protocolos, lógica de análise e tabelas de *lookup* personalizados, por exemplo. Um conceito interessante que ganhou força com a introdução de planos de dados programáveis é a Telemetria de Rede *In-band* (*In-band Network Telemetry* – INT) (JEYAKUMAR et al., 2014; KIM et al., 2015; THOMAS; LAUPKHOV, 2016). Dentro desse conceito, os dispositivos de encaminhamento são programados para anotar os pacotes de produção com metadados sobre o estado destes, seu comportamento e seu desempenho (*e.g.*, percentual de utilização de portas, regras de encaminhamento aplicadas e tempo de atraso de encaminhamento). As informações anotadas são acumuladas em um pacote ao longo de seu caminho e, em algum ponto da rede, extraídas e exportadas para servidores analisadores. Esses servidores reúnem as informações recebidas (conforme necessário) para construir uma visão global e precisa da rede, conforme o observado pelo seu tráfego.

As técnicas baseadas em INT mostraram produzir dados de monitoramento com um nível de precisão e granularidade sem precedentes (KIM et al., 2015; ZHANG et al., 2017). Isso porque, em vez de depender de pacotes específicos para medição (método ativo), que podem estar sujeitos a comportamentos de encaminhamento e roteamento diferentes daqueles do tráfego de interesse, os próprios pacotes de produção podem ser usados para investigar a rede. Além disso, a coleta de metadados pode ser feita precisamente durante os instantes em que pacotes individuais de interesse estão sendo processados em um dispositivo. Como consequência, o INT torna possível detectar e identificar eventos de rede que antes eram imperceptíveis, como congestionamentos que duram microssegundos.

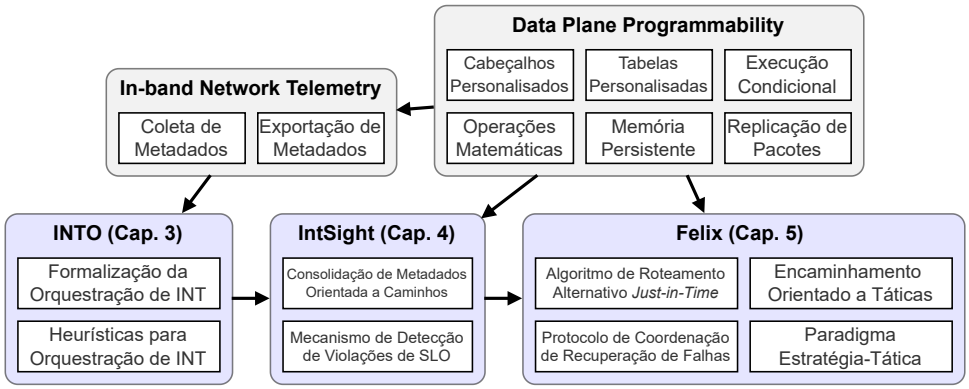
Embora a DPP traga maior flexibilidade para o desenvolvimento de mecanismos de monitoramento, para operar em *linerate* em links de alta velocidade, os programas de plano de dados são sujeitos a restrições no tempo de execução (dezenas de nanossegundos) e um espaço de memória limitado (por exemplo, centenas de megabits de SRAM e dezenas de megabits de TCAM) (BOSSHART et al., 2013). Em relação ao INT, por envolver a modificação de pacotes de produção que atravessam a rede, a quantidade de metadados que podem ser coletados por um pacote é limitada pelo seu tamanho original e pela unidade máxima de transmissão da rede (MTU). Além disso, algumas das ações

executadas podem aumentar a carga da rede e afetar o desempenho. Por exemplo, a incorporação de dados de telemetria em pacotes aumenta a carga nos links de rede e a geração de pacotes de relatório aumenta a carga nos dispositivos de encaminhamento e canais de controle. Nesta tese, é argumentado que essas restrições e fatores precisam ser cuidadosamente considerados para despertar todo o potencial da programabilidade do plano de dados para a disciplina de monitoramento de rede.

A.2 Hipótese, Questões de Pesquisa e Contribuições

Esta tese busca preencher a lacuna de concretização das novas oportunidades de monitoramento e operação de redes trazidas pela programabilidade de plano de dados (DPP) e telemetria de rede *in-band* (INT). A sua hipótese é que INT juntamente com DPP podem ser aplicados com sucesso para monitorar e operar redes com granularidade por pacote ao mesmo tempo que mantendo os *overheads* praticáveis. Esta tese traz oito contribuições valiosas que respondem a questões importantes sobre como gerenciar redes de forma eficaz e eficiente neste novo paradigma. A Figura 1.1 resume este trabalho, mostrando seus principais fundamentos originados do surgimento da DPP e INT, bem como suas principais contribuições. Essas contribuições são posicionadas como resultado de três questões de pesquisa que são descritos a seguir.

Figure A.1: Sumário da tese com suas principais contribuições e fundamentos.



Questão 1. *Como devem as ações de coleta de Telemetria de Rede In-band ser orquestradas entre os dispositivos em uma rede para maximizar a qualidade da medição e minimizar as sobrecargas de rede e tráfego?*

Uma análise inicial sobre a Telemetria de Rede In-band (INT) mostrou que quando tal conceito é aplicado de forma não sistemática para coletar metadados de cada um dos dispositivos visitados e por cada um dos pacotes de tráfego que atravessam a rede, um

encargo substancial é imposto sobre o tráfego, os recursos de rede e os servidores de análise. A sobrecarga imposta por essa tarefa pode chegar ao ponto de interromper a operação da rede ou seu efetivo monitoramento.

No Capítulo 3, como a **primeira contribuição** desta tese, é formalizado o problema de orchestração de telemetria de rede *in-band* (do inglês, *In-band Network Telemetry Orchestration* – INTO) por meio de programação linear inteira. O objetivo final do problema INTO é minimizar a sobrecarga de monitoramento, mas ainda assim obter dados de alta qualidade. São formalizadas duas variações do problema INTO como modelos de programação matemática, cada uma delas focando na otimização do uso de um recurso específico: a capacidade de processamento de pacotes dos dispositivos e a largura de banda dos links, respectivamente. Ambas as variações do problema de orquestração são provadas como pertencendo a classe de problemas NP-Completos. Através de uma extensa avaliação usando topologias de rede reais, é confirmado que a geração de soluções ótimas leva um tempo proibitivo.

Como **segunda contribuição**, também no Capítulo 3, é abordada a limitação de escalabilidade dos modelos de programação matemática projetando algoritmos heurísticos. Esses algoritmos são capazes de computar soluções de alta qualidade em tempo de computação polinomial para as duas variações do problema INTO. Através de uma avaliação, observa-se que as heurísticas propostas são capazes de gerar soluções próximas do ótimo para todas as topologias consideradas em menos de um segundo. Também é avaliada a qualidade e os custos associados às heurísticas propostas sob diferentes aspectos e comparados os seus resultados para identificar quais tipos de redes cada heurística é mais adequada para monitorar.

Na parte seguinte desta tese, continua-se este estudo sobre INT, colocando a DPP em foco e respondendo à seguinte pergunta intrigante.

Questão 2. *Dada a flexibilidade no processamento de pacotes fornecida pela Programabilidade do Plano de Dados, podem os dados de monitoramento ser pré-processados ou consolidados por dispositivos de encaminhamento antes de serem exportados ao plano de controle para reduzir ainda mais a sobrecarga e sem perda (ou mesmo melhoria) na qualidade da medição?*

Uma das observações resultantes do estudo de orquestração de INT é que os planos de dados programáveis têm recursos que podem permitir ações de monitoramento mais além da coleta de metadados brutos. A partir desta observação, no Capítulo 4, apresenta-se o INTSIGHT, um sistema que explora totalmente as capacidades de programação de

rede (BOSSHART et al., 2013; BOSSHART et al., 2014) para monitorar SLOs relacionados ao atraso fim-a-fim e a garantias de largura de banda. Em poucas palavras, INTSIGHT discretiza o tempo em janelas de duração fixa na ordem de subsegundos, chamadas aqui de épocas. O tráfego de rede e dos fluxos de interesse são monitorados pacote-a-pacote pelo plano de dados para observar seu estado, comportamento e desempenho (*e.g.*, roteamento, contenção, atraso, quedas de pacotes e largura de banda fornecida). Cada pacote de produção é instrumentado para transportar informações essenciais (em um cabeçalho de telemetria), que tem seus valores atualizados sistematicamente (por meio de computação na rede) à medida que o pacote se move em direção ao destino. O dispositivo de encaminhamento de egresso de cada pacote consolida essas informações temporariamente em sua memória. Ao final de cada época, as informações consolidadas mantidas para cada fluxo de interesse permitem detectar e diagnosticar eventos de violação de SLO, suas causas, vítimas e culpados. Como uma **terceira contribuição** desta tese, foram projetados e implementados procedimentos eficientes no plano de dados para computar gradualmente metadados relacionados aos caminhos, pontos de contenção, atrasos fim-a-fim e largura de banda fornecida. Através de uma avaliação aprofundada, demonstra-se os benefícios (em relação à funcionalidade, desempenho e demanda de recursos) da telemetria de rede in-band orientada a caminhos em comparação com abordagens do estado da arte.

A resposta positiva à segunda pergunta – ou seja, o sucesso no pré-processamento e consolidação dos dados de monitoramento diretamente no plano de dados – motiva a terceira e última pergunta.

Questão 3. *Pode parte da lógica de análise e reação (tradicionalmente posicionada no plano de controle) ser transferida para o plano de dados para permitir detectar e reagir a problemas de rede em mais rapidamente?*

A seguir, são descritos casos onde indentificou-se que a resposta para este pergunta é positiva. Primeiro, como mencionado brevemente, as informações consolidadas armazenadas em dispositivos de encaminhamento em INTSIGHT permitem detectar e diagnosticar eventos de violação de SLO. Durante o projeto do INTSIGHT, observou-se que a forma como as informações são consolidadas permitiria que os próprios dispositivos de encaminhamento detectassem épocas em que ocorreram violações de SLO. Como resultado, a **quarta contribuição** desta tesa é um mecanismo de rede, distribuído e orientado a caminhos para monitorar o tráfego de rede capaz de detectar violações de SLO e outros problemas que afetam o desempenho dos fluxos de interesse (*e.g.*, microbursts). No Capí-

tulo 4, ao apresentar o INTSIGHT, descreve-se como, no final de cada época, os desvios do que é esperado da rede (*e.g.*, violações de SLO) são detectados pelos dispositivos de encaminhamento de egresso, o que aciona a geração de relatórios. Os servidores do plano de controle recebem e analisam os relatórios gerados para identificar o tráfego culpado que tem afetado a operação normal da rede. Na avaliação do INTSIGHT, sua abordagem para geração de relatórios de dados de telemetria mostrou-se fazer o uso sensato da largura de banda do plano de controle e dos recursos dos servidores de análise sem perda significativa de precisão e nível de detalhes.

Seguindo o trabalho sobre o INTSIGHT, no Capítulo 5, é apresentado um passo adicional dado para responder à Questão 3. Move-se o foco do estudo para falhas de equipamentos de rede e são investigadas formas de redirecionamento de tráfego mais eficiente e resiliente, isto para atender aos SLOs relacionados à disponibilidade de serviço que surgem no contexto de SDN com planos de dados programáveis. Propõe-se FELIX, um novo sistema que calcula proativamente táticas de encaminhamento de pacotes tanto para o estado normal da rede bem como para cenários de falha, e programa essas táticas em dispositivos de plano de dados junto com um protocolo de coordenação leve para reagir imediatamente a falhas. De certa forma, o plano de controle atua como um estrategista que cria táticas de recuperação para lidar com falhas, enquanto o plano de dados executa a tática apropriada quando necessário. Essa abordagem para a solução do problema elimina a necessidade de esperar que o plano de controle calcule e instale novas regras de encaminha no momento em que uma falha ocorre, e ao mesmo tempo permite o uso dos melhores caminhos alternativos para contornar falhas. São feitas quatro contribuições principais com o projeto do FELIX. Como **quinta contribuição** desta tese, desenvolve-se um pipeline de processamento de pacotes com tabelas match-action personalizadas que encaminham os pacotes de acordo com o estado atual da rede. A **sexta contribuição** é o projeto de um protocolo leve rodando no *fastpath* dos switches para permitir a coordenação da recuperação de falhas inteiramente no plano de dados. A **sétima contribuição** é o desenvolvimento de algoritmos que calculam e instalam regras de encaminhamento alternativas bem a tempo de lidar com falhas possivelmente iminentes. Através de uma extensa avaliação do FELIX, observou-se que este reduz consideravelmente o tempo de reação a falhas ao mesmo tempo que este faz uso sensato da memória nos dispositivos do plano de dados. A **oitava**, e última, **contribuição** desta tese é a proposta do paradigma Estratégia-Tática. Esse paradigma abstrai os elementos de FELIX para formar uma arquitetura geral que pode ser instanciada para executar outras tarefas de operação da rede.

Exemplificamos a generalidade desse paradigma combinando as lições aprendidas com FELIX e INTSIGHT para esboçar INTREACT, um sistema para reencaminhamento em escalas de tempo de plano de dados orientado a SLOs.