BALANCED PATH GENERATION AND RELIABILITY EXTENSION FOR IN-BAND NETWORK TELEMETRY

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

GÖKSEL ŞIMŞEK

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

MAY 2021

Approval of the thesis:

BALANCED PATH GENERATION AND RELIABILITY EXTENSION FOR IN-BAND NETWORK TELEMETRY

submitted by **GÖKSEL ŞIMŞEK** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Ertan Onur
Supervisor, **Computer Engineering Department, METU** _____

Assist. Prof. Dr. Hande Alemdar
Co-supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. İbrahim Körpeoğlu
Computer Engineering Department, Bilkent University _____

Prof. Dr. Ertan Onur
Computer Engineering Department, METU _____

Assist. Prof. Dr. Pelin Angın
Computer Engineering Department, METU _____

Date: 03.05.2021

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Göksel Şimşek

Signature :

ABSTRACT

BALANCED PATH GENERATION AND RELIABILITY EXTENSION FOR IN-BAND NETWORK TELEMETRY

Şimşek, Göksel
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Ertan Onur
Co-Supervisor: Assist. Prof. Dr. Hande Alemdar

May 2021, 119 pages

Network monitoring is one of the key aspects to ensure communication reliability in case of failures and malicious activities and has several design issues depending on the system characteristics. As traditional monitoring solutions usually rely on periodic updates between the network controller and ordinary nodes, scalability remains a challenge, especially for large-scale systems. A recent solution, the In-Band Network Telemetry (INT) framework, allows data packets to probe the nodes while traversing the network. Accordingly, INT allows special packets to carry accumulated performance information of multiple switches, reducing the overhead between the controller and other nodes. Even though INT may significantly reduce the communication overhead, there are several design problems to achieve effective usage of the INT framework. These design problems are (i) minimization of the control overhead, (ii) guaranteeing the freshness of telemetry information and (iii) minimization of the redundancy. In this work, we formulate requirements as an optimization problem, Balanced Simple INT path generation Problem (BSIP), to generate balanced, simple INT paths. Due to the optimization problem's search space complexity, we propose a

heuristic, Graph Partitioned INT (GPINT), to find balanced paths to forward in-band telemetry information to satisfy these three requirements. Furthermore, we customize the INT framework to support custom and dynamic measurement ranges to achieve flexible monitoring. With this customization, the controller gains the ability to adapt measurement requests according to the network conditions. We present an extensive analysis of our approach, GPINT, and compare it with a recent study that uses Euler's method for path generation. Our numerical results show that GPINT outperforms its opponent in terms of all three requirements. To verify our claims made in numerical analysis, we deploy path generator approaches on a simulation environment and test with various settings. The simulation results show the importance of the defined requirements and verify GPINT's performance observed in numerical analysis. During the simulations, we realize that the INT framework is prone to packet losses and may cause partial information blackouts while obtaining a holistic view. Therefore, we propose a data recovery architecture as an auxiliary module to monitoring systems. We thoroughly test the recovery module in our simulations and measure its efficiency.

Keywords: in-band network telemetry, network monitoring, data recovery, programmable data plane, p4

ÖZ

BANT-İÇİ TELEMETRİ İÇİN DENGELİ PATİKA ÜRETİMİ VE GÜVENİRLİK KAZANDIRILMASI

Şimşek, Göksel
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ertan Onur
Ortak Tez Yöneticisi: Dr. Öğr. Üyesi. Hande Alemdar

Mayıs 2021 , 119 sayfa

Şebekenin düzenli bir şekilde gözlemlenmesi, veri düzlemi içerisinde oluşabilecek hatalara karşı güvenli veri aktarımını gerçekleştirebilmek veya şebekenin saldırılara karşı güvenliğini sağlayabilmek için en önemli temel ihtiyaçlardan biridir. Geleneksel veri düzlemi gözlem yöntemleri, şebeke içerisindeki tüm cihazlardan gelecek periyodik güncellemelere dayalı çalışmakta olup bu yöntemler merkezi kontrolcü ve cihazlar arasında yoğun bir trafiğe sebep olmaktadır. Bu sebepten dolayı geleneksel gözlemleme yöntemlerinin büyük ve karmaşık sistemlerde uygulanabilirliği kısıtlıdır. Bant-içi telemetri (in-band telemetry (INT)) bu kısıtlamaya karşı ortaya çıkan bir çözüm olup trafikteki özel paketlerin ağ anahtarlarında tutulan verilerine erişimini sağlamaktadır. Bu sayede özel bir paket birden fazla ağ anahtarının verisi taşıyabilmekte ve merkezi kontrolcü ile ağ cihazları arasındaki trafiğin azalmasını sağlayabilmektedir. Her ne kadar bant-içi telemetri iletişim yoğunluğunu azaltmayı sağlayabilse de bu yöntemin etkili olarak kullanılabilmesi için bazı tasarım sorunlarının ele alınması gerekmektedir. Bu sorunlar (i) merkezi kontrolcü üzerindeki trafik yükünü azaltmak,

(ii) uzölçümlerinin güncelliğinin korunmasını sağlamak, ve (iii) taşınacak tekrar eden bilgi miktarını azaltmak olarak sıralanabilir. Bu çalışma, sıralanmış olan üç problemi ele alan bir optimizasyon problemi, dengeli basit bant-içi patika üretimi problemi (BSIP), tanımı yapıyor ve problemin çözümü olarak dengeli bant-içi telemetri patikaları üretiyor. Optimizasyon probleminin çözüm üretmek için ihtiyaç duyduğu değer arama uzayının büyüklüğü sebebiyle büyük veri düzlemleri için kullanılması çok fazla zaman ve kaynak gerektiriyor. Bu sebepten dolayı, aynı problemi ele alan ağ parçalama tekniği üzerine kurulmuş ağ diyagramı bölerek INT yolu oluşturma (GPINT) algoritması sunulmuştur. Bunun yaninda, INT protokolüne yeni bir modül eklenip, özel paketler ile dinamik bir şekilde ayarlanabilen ölçüm aralıkları verilmesi sağlanmıştır. Dinamik aralıklar sayesinde merkezi kontrolcü kendi ihtiyacı ve ağ düzleminin durumuna göre ölçümlerin tamamını almak yerine istediği aralığı ve miktarı alabilmesi sağlanmıştır. Böylelikle INT protokolünün ağ şartlarına adapte olabilme özelliği arttırılmıştır. Önerilen GPINT algoritmasının performansını hem sayısal, hem de simülasyon ortamında detaylı bir şekilde inceleyerek daha önceden kabul edilmiş Euler tekniği ile karşılaştırılmıştır. Sayısal karşılaştırmalarımız sonucunda GPINT algoritmasının BSIP çözümüne oldukça yakın kalitede yollar ürettiği görülmüş ve Euler tekniğini karşısında üstün bir performans sağlamıştır. Simülasyon ortamında da tespit ettiğimiz bu üç ihtiyacın önemli olduğunu doğrulanmış ve GPINT algoritmasının etkili bir ağ gözlemi yapmak için uygun patikalar üretebildiği gösterilmiştir. Simülasyon sırasında INT protokolünün paket kayıplarına karşı hassas olduğu gözlemlenmiş ve bu kayıpların ağ gözlemi yapmayı zorlaştıracak seviyelere ulaşabileceği tespit edilmiştir. Buna karşılık INT protokolü için veri kaybını engelleyici bir modül önerilmiş ve önerilen modül simülasyon ortamında test edilmiştir.

Anahtar Kelimeler: Bant-içi telemetri, ağ gözlemi, veri kurtarılması, programlanabilir veri tabanı, p4

To my family and close friends

ACKNOWLEDGMENTS

I would like to offer my sincere gratitude to Prof. Dr. Ertan Onur for accepting me as his student and giving me an opportunity to work with him. Throughout this journey, I had many moments where I was clueless, but in those moments, my dear friend Doganalp Ergenc answered all of my questions and led me in the right direction with his critizim to improve the quality of this work. Thanks to their genuine guidance, I have learned and grew a lot. Thank you.

I am profoundly grateful to my parents and brother for their love and caring. They have constantly supported my education and been there whenever I needed them. I wish my father could see me write and finish this thesis, for which he would have been the happiest and the proudest.

Finally, this thesis is supported by Vodafone under BTK Graduate Scholarship Program. I would like to thank the authorities who started this program to support students in their academic careers.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

BSIP	Balanced Simple In-band Network Telemetry Problem
DLV	Device-level
DSCP	Differentiated Services Field Codepoints
FBCK	Feedback
FT	Fine-tuned
GPINT	Graph Partitioned In-band Network Telemetry
ID	Identifier
INT	In-band Network Telemetry
IoT	Internet of Things
MAC	Media Access Control
MEA	Measurement
MTU	Maximum Transmission Unit
P4	Programming Protocol-independent Packet Processors
PRB	Probe
SDN	Software-Defined Networking
SNMP	Simple Network Management Protocol
SR	Source Routing
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

CHAPTER 1

INTRODUCTION

As a result of the integration of various new technologies and services, modern networks have become highly-complex ecosystems with a multitude of components. While data centers are enlarging to meet the need for exploding data-driven services, heterogeneity in the Internet of Things (IoT) is forcing the change of networking paradigms with new challenges [1]. Accordingly, the increasing complexity of networks requires efficient monitoring and management mechanisms to guarantee (i) reliable communication to detect and mitigate network failures and attacks, (ii) to reconfigure the system in case of dynamically changing resource and traffic demands, especially for critical components and services.

In the traditional networks, the control plane, which handles how the traffic should be forwarded, and the data plane, which applies the decisions made in the controller plane, are bundled together inside the networking device. As a consequence, the network operators have to configure each individual network device separately using vendor specific commands when they want to adjust control plane policies or introduce new protocols [2]. Due to the configuration complexity, dealing with faults and adapting to load changes become daunting task to achieve. Even though traditional networks guaranteed network resilience to some extent, the resulting network architectures were relatively complex and static [3–7]. Another side effect of such complex networks is the lack of efficiency in gathering monitoring information [8,9]. For instance, traditional network management protocols like the Simple Network Management Protocol (SNMP) [10], NetFlow [11] and sFlow [12] were the first steps to define required interfaces and methods to design network monitoring and management systems, but they cannot offer a full-fledged solution for today's complex

networks [13, 14], such as 5G networks and cloud-based data centers [8].

Software-Defined Networking (SDN) is a more recent and popular solution to cope with such complexity. SDN separates the network's control logic, control plane, from the underlying network devices (switches, routers), data plane. With the separation of control and data planes, SDN simplifies the network management [15, 16] by facilitating creating and introducing new abstractions in networking. As a result, SDN introduces the ability to program the network via a centralized controller to orchestrate network nodes (e.g., switches) and provides network-wide visibility for flexible configuration [2]. With the introduction of the next generation mobile network, 5G, the flexibility SDN brings can help managing large number of connected devices, especially when they expected to run different services [17, 18]. To achieve such management, there has to be efficient and scalable monitoring system leveraging and complementing the flexibility of the SDN. In the SDN, the controller can request various measurements and statistics such as link utilization and port-meters from switches to monitor and reconfigure the network if required. However, polling those measurements from each network node does not scale well as it increases control traffic for measurement requests and responses [19, 20]. Additionally, there are several hindering blocks against designing efficient and scalable monitoring system for SDN. One of them is the fact that widely deployed OpenFlow [21] enabled SDN switches lack intelligence and depend on the controller for traffic monitoring/forwarding [22]. Another limitation of OpenFlow enabled SDN switches is they only support fixed set of headers and are not flexible enough to allow designing custom protocols [23, 24]. Consequently, the operators lack the ability to define new protocols according the network needs. With the latest advancements in switch design [23], the P4 language (Programming Protocol-independent Packet Processors) [25] has emerged, addressing these limitations of OpenFlow switches. The P4 is a data plane description language that facilitates the customization of packet processing and forwarding pipelines [24, 26]. With the level of customization it brings, it is easy to introduce new protocols or to customize actions that switch can take on each packet. Accordingly, the P4 Language Consortium proposes a scalable telemetry protocol, In-band Network Telemetry (INT) [27] to address the monitoring challenges introduced by poll and push-based methods which are vastly employed in SDN.

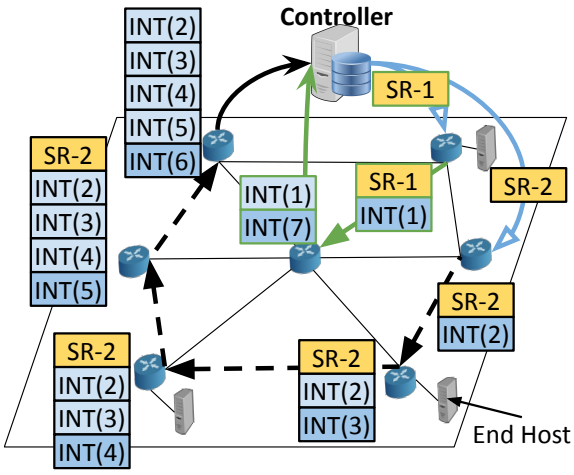


Figure 1.1: An example usage of INT to cover every switch in the network.

As described in the pioneering study by Pan et al. [28], using specific *INT packets*, it is possible to query telemetric information such as queuing latency, queue depth, and packet-metering from P4-programmable switches on the fly. That is, once an INT packet is triggered and forwarded through a predetermined path, *INT path* (e.g., using source routing (SR)), each node through that path extends the packet by appending its measurements. Eventually, the destination node of an INT path receives the collective information of several nodes within a single INT packet without probing each node separately.

In Figure 1.1, we depict an example use case of INT paths that cover all switches (nodes) in the network to collect their measurements to create a holistic view. In this example, the controller generates two probe packets, SR-1 and SR-2, and awaits both to arrive before constructing the global view. As one can notice, SR-2 takes a longer tour than SR-1. Consequently, SR-2 collects more measurements and consumes higher bandwidth.Moreover, as SR-1 reaches the controller earlier, it represents an older state of the network when SR-2 is received. That time-difference hinders the controller from having a recent and complete network-wide view of a certain network state. Even though the controller can utilize multiple shorter paths instead of a single long path, e.g., SR-2, an increasing number of paths also increases the number of INT packets to be processed, and eventually, the load of the controller. In the case of utilizing joint paths, i.e., traversing the same nodes, they carry redundant information for some nodes and induce extra latency. INT paths can also be deployed

to cover every edge to obtain link statuses, such as latency [28]. In this case, paths traverse the same switches, which is similar to employing joint paths. NetView [29] shows that it is possible to summarize link statuses by carrying additional telemetry data, hence ceasing the need to cover every edge.

1.1 Requirements and Problem Definition

Although INT enables us to design efficient monitoring mechanisms, there are several design requirements to maximize efficiency, as concluded below.

Requirement 1 *INT path(s) should traverse all nodes in the network.* As the monitoring system should receive measurements from all nodes in the network, each node should receive at least one INT packet through an INT path to write its measurements. Any number of paths can be defined to cover the network. However, while several shorter INT paths result in an increasing number of INT packets to be processed, a few long paths would impose a higher delay to deliver INT packets to the controller. Besides, as INT packets grow with new measurements after each hop, larger INT packets should be forwarded through the longer INT paths. When such an INT packet gets lost in the network due to failures, the controller will lose the information of a considerable portion of the network. Therefore, the goal should be **to find the optimum number of INT paths that cover the whole network**.

Requirement 2 *End-to-end latency of INT paths should be similar.* The delivery time of INT packets depends on the processing, queuing and propagation delay of INT paths. The deviation in the latency of different INT packets hinders a consistent network view as some measurements might reflect an older state of the network. Accordingly, the goal should be **to deliver INT packets concurrently** to have a timely and holistic view.

Requirement 3 *INT paths should be as disjoint as possible.* The INT packets that traverse intersecting INT paths carry duplicate information of the same network

nodes. While redundancy enlarges the packet size, traversing the same nodes also imposes an extra delay. Furthermore, under high-frequency monitoring requests, deploying joint paths may increase the load of the switches. Hence, the goal should be **to find disjoint paths to minimize redundant information**.

There can be many other complementary design requirements defined and enforced upon path generation procedure. For instance, there can be a reliability requirement to guarantee report delivery under any condition. In this work, we do not consider such a requirement but design a data recovery mechanism for the INT framework to make any INT probe generator reliable. Another requirement could be satisfying multiple monitoring requests, potentially with different frequencies. We lay foundations in the data plane to realize such a requirement in the future by enabling dynamically customizable data request ranges and types.

1.2 Contributions of the Thesis

This study's contributions can be listed in two categories: design of INT probe generator, and customization of data and control planes to improve INT protocol.

In this study, we design an effective INT probe generator, and our contributions are as follows:

1. We formulate the Balanced Simple INT path generation Problem (BSIP) to find optimal simple INT paths. We use the requirements listed in Section 1.1 as our objective functions, which determines the optimality of generated paths. To generate simple (i.e., loop-free) paths, we use Miller-Tucker-Zemlin's (MTZ) [30] constraints for the traveling salesman problem.

2. We realize that the search space required by the BSIP is vast, making the optimization models not suitable to be deployed on the controllers in real-life scenarios. Therefore, we propose a Graph Partitioning-based heuristic, GPINT, to target BSIP and find near-optimal INT paths, published in [31]. GPINT is an extension of Kernighan-Lin's algorithm [32], and a typical Kernighan-Lin-like algorithm contains three stages, initial partitioning, exchange, and repetition.

In this study, we modify both initial partitioning and exchange stages to target BSIP.

Our main contributions in data and control planes can be summarized as follows:

1. We leverage the flexibility of the P4 programmable data plane and customize INT protocol so that it can support dynamically changeable request types and ranges. With the dynamic ranges, we lay the foundations for the controller so that it can adapt monitoring requests to network conditions. For instance, in congested conditions, the controller can collect a small number of measurements or maximize the number of measurements in normal conditions.

2. We realize that the INT protocol suffers from packet losses quite considerably since a single probe packet carries multiple stacks of information obtained from several switches. For instance, a single INT probe packet loss can blackout a major portion of the network, which can hinder monitoring decisions and pinpoint the issue in the network. Therefore, in this study, we propose a data recovery module for the INT framework to recover packet failures seamlessly. Our proposal is influenced by SQR [33], which targets commercial packet losses. Accordingly, we reprogram the data plane from packet parsing to ingress and egress pipelines to deploy the data recovery module for the INT framework. Furthermore, we support various customizable parameters in the data recovery module and analyze them thoroughly to lay the foundations for future works where these parameters can be automatically determined.

Finally, we provide an extensive numerical analysis and simulation results of the proposed probe generators and compare them with Euler's method [28], a recently published INT path generator algorithm. Furthermore, we show the implications of our data and control plane modifications and lay the foundations for future works.

1.3 Outline of The Thesis

The rest of the thesis is organized as follows. In Chapter 2, we first provide background information on in-band network telemetry and source routing, which makes

probe generators possible. Then, we present our related work, which contains works from traditional network monitoring to different types of monitoring approaches, such as sketch- and query-based techniques. We conclude the chapter with INT-based works. In Chapter 3, we explain our control and data plane architecture designs in detail to realize reliable INT with dynamically customizable measurement ranges. Chapter 4 defines the formulation of the BSIP, from path generation constraints to objective functions. At the end of the chapter, we analyze the problem's search space and discuss why it is not feasible. In Chapter 5, we present GPINT heuristic. First, we start by providing the necessary information about the Kernighan-Lin's algorithm. Then, we explain our extensions and conclude the chapter with complexity analysis. In Chapter 6, we present a data recovery module for the INT framework to achieve seamless recovery. First, we explain how SQR recovers commercial packet losses in the data plane. Afterward, we present how we apply a similar idea but specific to the INT framework. In Chapter 7, we provide detailed numerical and simulation analyses of probe generators. Furthermore, we analyze the data recovery module on different network conditions and with various parameters. Moreover, we discuss how different request ranges affect measurement collection and compare it with corresponding frequency levels to achieve similar data collection. Finally, in Chapter 8, we conclude our study and discuss future directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we explain some of the key concepts in more detail to make this work more accessible and then review major works for the design of a monitoring system.

2.1 Background

We divide the background information into two subsections. In the first section, we review the fundamentals of major monitoring techniques and INT. By doing so, we explain how INT is different from other monitoring techniques. In the second part, we describe the source routing (SR), an enabler for deploying INT paths.

2.1.1 In-band Network Telemetry

In terms of technical definition, network telemetry refers to the network data collection, and consumption techniques [34]. That is, network telemetry is an enabler for entities to obtain data from network devices so that the collected data can be analyzed and turned into something meaningful to support network monitoring and operation. Hence, designing efficient and effective data collection techniques is crucial for maintaining and operating the network at its total capacity.

In traditional monitoring and management protocols, such as SNMP [10], the management entity (the controller) works in polling mode, in which case it requests statistics from hundreds of devices one by one in low frequencies [29]. However, it is more beneficial to monitor devices continuously and desirably at higher frequencies so that the running applications (i.e., traffic management, fault detection) can always access

information in real-time and, by doing so, achieve better decisions [34]. Hence, traditional protocols that work in polling mode generate communication overheads that cannot be overlooked to provide real-time network telemetry.

As the time progressed, traditional monitoring and management protocols (i.e., SNMP, sFlow [12]) were extended to support data pushes without intervention of management entity, where devices periodically send data to the entity. However, the pushed data offered by these protocols are limited to predefined warnings or sampled user packets. Due to limited data pool, network operators cannot collect the data with high granularity and precision to operate efficiently. With the introduction of programmable switches, the operators gained the ability to extend the available data pool. Hence, they continue supporting the main idea of periodically pushing data and implement it in some of the commercial devices [35–38].

On the other hand, the in-band network telemetry framework, proposed by the P4 organization, addresses the monitoring quite differently. INT leverages the programmable data plane's ability, allowing special probe packets to query telemetry as they visit network devices. There are various approaches to employ INT as a monitoring framework. In one of the applications, programmable switches can temporarily convert commercial packets to INT and push their telemetries without generating new packets. Just before those packets reach the destination, switches pop telemetry headers and send them to the measurement entity. In this application, INT is adjusted to work as a medium to collect telemetry from switches that can be considered working in push-mode. In another application, one can intelligently generate routes of switches on-demand to collect specific measurements. This method requires the management entity to deploy additional packets to the network, which resembles sending requests in poll-mode but with less overhead. In either case, the INT packets carry accumulated data and reduce the communication overhead significantly. Hence, the INT is an excellent choice to fetch information when applied correctly, even at high frequencies.

The telemetry data INT offers can include but are not limited to precise telemetry measurements such as queuing latency or queue occupancy. The maintainer can also implement custom measurements (i.e., number of packets that certain match/action

pair activated for) by utilizing flexibility and customizability P4 brings. Official specifications [39] define three different query types or so-called application modes for INT: INT-XD, INT-MX, and INT-MD. The INT-XD (export data) mode acts as an alert system. When such a packet arrives, switches send information back to the monitoring system by using instructions configured at their flow watchlists. In the INT-MX, embed instructions mode, the probe packets carry several instructions that the nodes generate reports based on and send them to the monitoring application. The packet size does not increase but may decrease as switches can pop instructions if they are specific to them. The last and the most popular mode is INT-MD, embed data. As the name implies, in this mode, every switch inserts INT measurements specified in the corresponding INT instructions and populates the packet. In other words, a single INT packet can carry accumulated data, unlike poll and push modes, where every information is carried in different packets. Consequently, when INT is correctly applied, it can reduce the communication overhead and the load of the controller as it will need to process fewer packets. However, the INT specifications do not define how to construct INT paths, a sequence of connected devices. In this work, we layout three requirements that should be addressed while constructing INT paths to obtain a holistic view of the network quickly, flexibly, and efficiently. Like other INT-based monitoring systems, we share a common problem of how to forward INT packets and follow the same solution, employing Source Routing (SR).

2.1.2 Source Routing

Normally, packets are forwarded based on the destination IP addresses. However, in the case of INT packets, we need to forward them over a special path that is likely to be irrelevant to IP addresses. It is possible to direct INT packets by entering unique rules to every network device. Unfortunately, doing so would impose a high overhead on the controller since it would need to be constantly adjusting rules on every device. The Source Routing [40] overcomes the scalability problem by introducing a special header that devices can read and forward packets based on the written forwarding rules. Typically, a sender stacks the relative port numbers of switches in the packet header, and every network device on the route follows them. This flexibility and its redundant overhead make SR a de facto technique to forward INT probe packets.

2.2 Related Work

In this section, we present different approaches and major works for the design of monitoring systems.

2.2.1 Sampling-based Approaches

Sampling-based approaches collect telemetry with a sampling rate of n of packets associated with a session. In NetFlow [11], network entities forward session-specific statistics to the centralized or distributed collector units periodically to be merged and summarized. Depending on the number of flows, it requires a significant amount of resources to keep track of session information. sFlow [41] addresses this issue by improving the sampling strategy. The sampling-based approaches are not limited to traditional networks but can also be employed in SDN based environments. For instance, OpenNetMon [42], introduces adaptive rate for polling switches. They increase or decrease the rate based on network events such as flow rate instabilities (sudden increase or decrease). Another sampling-based approach is OpenSample [43], which focuses on assessing link utilization. Similar to OpenSample, PayLess [44] proposes an alternative adaptive sampling algorithm to realize real-time network monitoring. Even though sampling-based techniques can identify certain network issues, they lack the scalability aspect, especially when network conditions demand monitoring updates in high frequency. The scalability aspect can be improved by leveraging the programmable data plane to its full capacity. For instance, Kucera et al. [45] propose a tailored data structure for data plane to detect events such as heavy hitter and super-spreader detection. Upon a network event detection, switches notify the controller to take necessary actions so that the controller intervention is not needed to fetch telemetry. Consequently, the controller's load is reduced significantly.

2.2.2 Sketch-based Approaches

Sketch is a compact data structure tailored to summarize streaming data and can prove highly detailed flow information with resource utilization [46,47]. One of the prelim-

inary works, OpenSketch [48], introduces customizable flow measurement collection based on sketches. UnivMon [49] enhances the sketch-based approach by introducing different monitoring requirements and balancing them across the network. Unlike OpenSketch and UnivMon, ElasticSketch [50] focuses on the monitoring system's adaptability to limited bandwidth, varying flow size, and packet rate distribution. NitroSketch [51] addresses the resource and computation limitations of software switches and proposes an acceleration scheme for sketch-based monitoring systems. Although sketch-based approaches offer highly detailed flow measurements, they do not provide switch-level performance indicators such as queuing delays, which might be essential to diagnose switch performance.

2.2.3 Query-based Approaches

Network query languages allow executing high-level functional constructs such as map, filter, reduce by utilizing data and/or control plane. The Marple query language [52] focuses on executing queries solely on the data plane with a new cache design to improve query performance. Additionally, it supports conversion from the Marple query language to switch language so that the maintainer execute queries without worrying too much about reprogramming the data plane. Sonata [53], on the other hand, utilizes both data and control planes. However, both Marple and Sonata require compiling queries and loading them on switches, which can interrupt packets forwarding and monitoring. *Flow [54] tackles this issue by placing the aggregation functions to control plane while keeping feature selection and flow identification functions in the data plane. Newton [55], on the other hand, argues that *Flow requires too much monitoring traffic and high computational power to analyze collected data. Lastly, HyperSight [56] focuses on monitoring packet behavior changes proposing Packet Behavior Query Language and Bloom Filter Queue (BFQ), an efficient algorithm for packet behavior recording in the data plane. Despite the improvements, the communication overhead between the control and data plane still presents a challenge to overcome for both sketch-based and network query languages.

2.2.4 INT-based Approaches

Throughout the literature, there are various approaches to how INT can be applied and employed as a monitoring framework. In this review, we focus on two categories, utilizing commercial traffic and introducing artificial probe packets to collect and deliver reports.

The first and initial approach leverages the idea of using business packets to carry INT headers. On its basis, switches insert INT headers to every received packet. Right before the packet being forwarded to its destination, switches extract the accumulated data and send it to the controller. In one of the earliest works, Kim et al. [27] employ INT to construct continuous latency plots specific to HTTP requests. However, inserting INT headers to every packet generates considerable overheads. There are two different sets of strategies for addressing the overheads.

The first strategy is to address the overheads at the server or the controller side. One of the earliest works, IntMon [57], designs a data collection and analysis architecture as a service in the controller. Prometheus INT exporter [58] offers a two-layered data extraction mechanism. First, they extract network information from the packets and send them to a gateway with a direct connection to the centralized database. In the database, they periodically update the stored information with the latest arrivals to the gateway. INTCollector [59] argues that both IntMon and Prometheus INT exporter lack historical information collection, resulting in missing critical network events or connections while analyzing the data. Accordingly, they propose a sophisticated data extraction architecture. Upon the arrival of raw INT telemetry reports, they first parse and filter the reports such that they are easy to analyze. Afterward, they identify important network events such as hop-by-hop delay fluctuation and queue congestions. They store the network events in a time-series database and achieve reduced storage cost while providing historical data.

In the second strategy, the main goal is designing complex data plane programs that can address induced overheads. Selective-INT [60], which shows that adaptive INT insertion rate can achieve similar results compared to inserting INT to every packet but with much lower overhead. In addition to adaptive INT insertion, the Flexible

Sampling-based INT [61] also utilizes an event-based INT insertion mechanism. The events are user-defined and configurable such as having a queue length exceeding a certain threshold. Different than previous works, INT-Label [62] considers labeling state of switches with an adaptive rate. Every switch maintains a labeling rate and accordingly inserts port statistics to the commercial traffic. PINT [63], on the other hand, argues that inserting the complete device information at every hop induces high overhead and designs a probabilistic variation of INT that spreads out the information onto multiple packets. They show that telemetry data approximation is sufficient for monitoring applications to perform at levels close to when they receive the complete telemetry data. The probabilistic part comes from the PINT's query engine, which assigns a likelihood of executing a query set on packets and notifies the switches. If the query set changes often, it may reduce the PINT's scalability aspect as it needs to notify every switch in the network. Hyun et al. [64], propose a combination of two strategies, an INT management architecture built on top of INTCollector. Their management system includes the flexibility of starting and stopping INT measurements and which information sets to collect during measurement sessions. However, to enable different information set collection, they notify every switch, which shares a similar consideration with PINT.

In the second approach, agents or the controller introduce additional probe packets to collect telemetry data from switches. The goal is to direct one or more probe packets so that the monitoring entity will obtain a holistic view. Compared to the aforementioned INT methods, probe-based INT requires the monitoring entity to introduce additional packets to the network. In return, it gains the flexibility to redirect them and query only the information mentioned in the probe packets rather than configuring switch tables to achieve so. NetVision [65] and INT-Path [28], the *Euler* method, are the earliest examples of monitoring schemes that utilizes probe-based INT. Both of the works generate probe paths and direct them with source routing (SR). To generate the probe paths, both leverage Euler's graph theory and Hierholzer's algorithm, which give a theoretical minimum number of nonoverlapping paths to cover every edge (link) in the network. INT-Path capitalizes Euler's following theorems, (a) a connected graph with two odd vertices has an Euler trail starts from one odd vertex and ends at the other one, and (b) a connected graph with $2m$ odd vertices con-

tains at least m distinct trails which, altogether, traverse all links of the graph exactly once [66]. Accordingly, the INT-Path's performance relies on the number of odd degree vertices in the network, limiting the applicability and scalability. For instance, if there are few odd vertices, then the generated paths will be long and potentially unbalanced. Furthermore, it does not address possible cycles in probe paths generated by Hierholzer's algorithm, which decreases the freshness of information. Bhamera et al. [67], INTOpt, employ probe-based INT to monitor Service Function Chains (SFCs). They propose a simulated annealing-based random greedy heuristic to address a range of requirements of different service functions. Marques et al. [68] consider multiple monitoring applications requesting link statistics, similar to INTOpt. They formulate two different optimization formulations, one to cover every link with a minimal number of simple INT paths. The other is to balance the length of the simple paths to avoid saturation due to data growth in probe packets. They prove the optimization problems are NP-Hard and propose two heuristic algorithms correspondingly. However, using simple INT paths to cover every interface drastically increases the number of generated paths. Besides, joint paths increase the overhead on switches, and in the case of high-frequency monitoring applications, the overhead may become significant. NetView [29], on the other hand, utilizes simple paths to cover switches instead of links and collects additional telemetry data to estimate link status. NetView uses only one origin server to generate and collect probes, which introduces a disadvantage. When a probe packet finishes data collection on a distant switch, it needs to be forwarded back to the origin server. Consequently, this introduces additional risks of losing the accumulated data if the packet gets dropped due to link failures or congestions. Furthermore, for these packets, every forward introduces extra delays and degenerates the information. Additionally, NetView allows paths to be discontinuous, which causes switches that are not on the path to parse INT packets and introduce extra delays.

In our work, we utilize NetView's findings that covering all switches is sufficient to summarize link statuses. Accordingly, we propose an optimization model and a graph partitioning-based path generation algorithm to cover all switches assuring minimum overlapping and length deviation. We compare our proposals with the Euler method, which also focuses on path generation with a single monitoring demand and does not

have topology limitations.

CHAPTER 3

A RELIABLE IN-BAND NETWORK TELEMETRY IN PROGRAMMABLE DATA PLANE

The programmable data plane grants immense flexibility in terms of designing new protocols or customizing existing ones. One might argue that too much flexibility might arise many design questions and, as a result, it can over-complicate system architectures. In this chapter, we explain our solutions to two design questions that should be answered while developing an INT-based monitoring framework. These questions shape how we architecture the system, from the controller to the processing pipelines of switches. We start with our controller architecture, then move on to the packet layout and finish with the modifications we introduce to the data plane to recognize customized packet layout and employ reliable INT. Before we discuss our architecture, let us state our assumption in this work that the network is homogeneous, and all switches are P4 programmable. The complications that heterogeneous networks bring are explored in Chapter 8.2.

3.1 The Controller Design

One of the design questions that should be addressed is the probe generator and data collection entity placements. Multiple variations can be considered, such as placing both entities within a single target or separating them. For instance, it is possible to place probe generators behind switches and the data collector in the controller. While this separation offers some benefits, such as reduced controller overhead, it also comes with a significant disadvantage. That is, whenever the topology changes, the controller needs to notify the probe generators. However, since the probe gen-

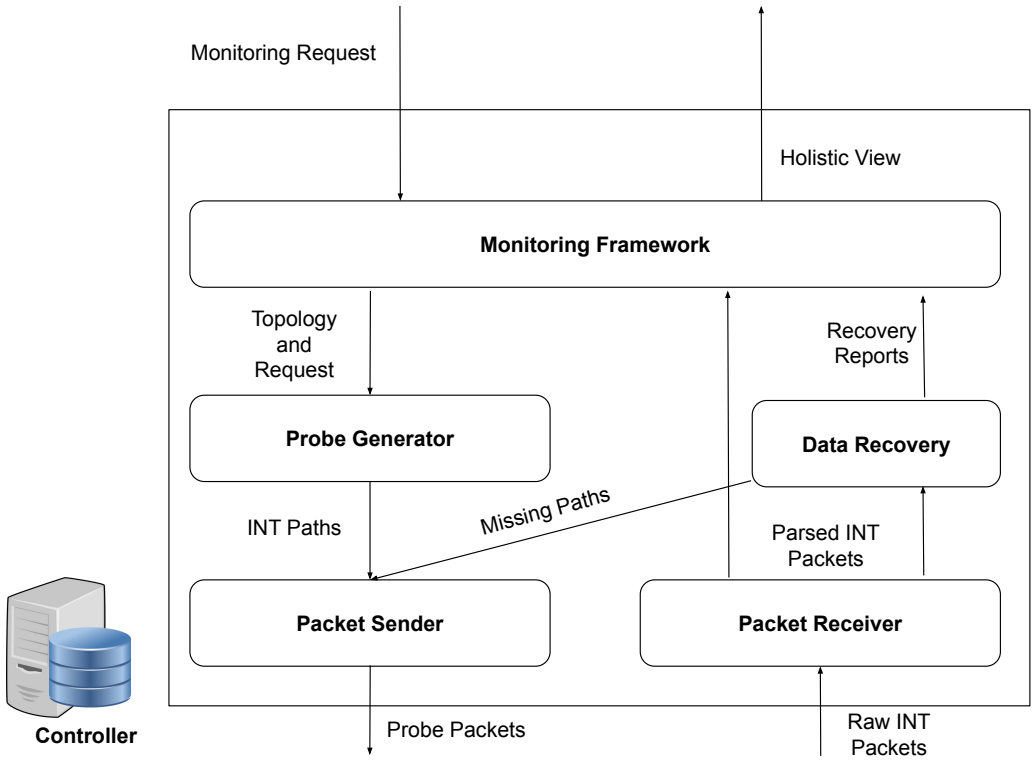


Figure 3.1: The developed controller architecture that is used in our simulations.

erators are behind switches, notification packets are prone to possible congestions. Another option is to place both the probe generator and the data collector in the same device behind a switch [29]. This option indeed ceases the need for notification packets but introduces a new problem. When a probe packet finishes data collection on a distant switch, it needs to be forwarded back to the origin server. Consequently, this introduces additional risks of losing the accumulated data if the packet gets dropped due to link failures or congestions. Furthermore, every forward introduces extra delays and degenerates the information. Addressing these shortcomings, we find it best to place both the probe generator and the data collection entity in the controller. Even though sending out and collecting probe packets may induce high overhead on the controller, this design is flexible as the centralized entity can be distributed.

In Figure 3.1, we provide a rather generic architecture we use in this work. At the highest level, we have the **Monitoring Framework**, which accepts monitoring requests and constructs a holistic view as a result. Monitoring requests can be in different forms, such as extracting rule hits or port packet counters with a specified range

of information unique to each switch. However, the type of requests can easily be extended by leveraging the programmable data plane to customize both INT protocol and statistics collected in switches. Once a request arrives, we parse and forward it to the **Probe Generator** along with the underlying topology in the form of an undirected graph. The probe generator produces INT probe paths that the controller can deploy to collect measurements. One of our main contributions in this work is designing an effective probe generator. We lay three requirements in Chapter 1.1 to realize such design. In Chapter 4, we formalize these requirements and propose the Balanced Simple INT Problem (BSIP). Since networks evolve in time, by the addition of new switches or extraction of links, the controller may need to run probe generators frequently. Deploying optimizers to solve the problem on the controller becomes infeasible as the topology scales due to the problem's search space complexity. Therefore, we propose a heuristic algorithm, Graph Partitioned INT (GPINT), in Chapter 5 to generate balanced INT paths. Additionally, we employ two extra probe generators: SNMP, INT-Path's Euler method [28], and compare them with our proposal GPINT in Chapter 7. Once the selected probe generator returns generated paths, we forward them to the **Packet Sender**. In the Packet Sender, we construct SR headers accordingly and generate INT Probe packets. Once probe packets are generated, we use a process pool to release them to the network and wait for them to complete their paths.

When a path completes its journey, it arrives at the **Packet Receiver**. Here, we parse the raw INT packets and generate readable INT reports. There can be many applications subscribed to these reports apart from the Monitoring Framework. One of such applications that we deploy and propose in this work is the Data Recovery Module. The goal of this module is to achieve seamless report recoveries under harsh conditions that may cause INT reports to be dropped. We explain the data recovery module in detail in Chapter 6. Once all of the paths arrive at the Monitoring Framework, it notifies all its subscribers that the current measurement request has concluded. In this case, the data recovery module is a subscriber. On notification, it provides its findings to the framework, such as congested links or the number of additional paths it needed to deploy.

3.2 Packet Layout

Another design question is the encapsulation of INT headers. The official documentation describes several approaches, including encapsulation over common protocols TCP and UDP. While both of the L4 protocols are perfectly well equipped to carry INT headers, in this work, we prefer UDP encapsulation due to its smaller header size. One of other design question to be addressed is which information to carry in INT headers. Technically, one can employ INT to monitor any sort of information available in the data plane, whether it is device level or user-defined measurements. The only limitation is the MTU since the packets grow at each switch. To overcome this limitation, one can either introduce INT packet fragmentation at the data plane or limit the feature set to fetch accordingly. One thing to note here, shorter INT paths inherently mean larger feature space is available to the monitoring system. There is also another hidden option: instead of limiting the feature set, one can design several INT variations to carry different feature sets. Doing so limits the number of features to be carried in each INT variation but not the feature set's size. To demonstrate its applicability, in this work, we employ two different INT variations for the feature set consisting of rule hit counters and port-based packet counters. A rule hit counter refers to a count of the number of times a certain forwarding rule has been hit. Likewise, a port-based packet counter measures the number of packets received from a given port. In modern networks, these counters' sizes can be enormous depending on the number of switches and servers in the network. Consequently, employing INT packets to fetch these counters in one pass would not be feasible as they might exceed the MTU limits in only one hop. Hence, we realize another design problem, supporting custom measurement ranges. Two different approaches can be taken to support such ranges. The first one is to make the measurement range static and always fetch that much information from switches. The only customization then would be the beginning and end of the static range. The advantage of static ranges is their simplicity and lesser complexity. However, the disadvantage is the uncontrollable bandwidth consumption which might be a limiting factor in challenging network conditions. The other approach is to allow customized ranges whose length can be controlled by the controller. This offers full control over how much information to be fetched from each switch; however, it comes with implementation complexity. In this work, we

prefer the latter approach and offer full control to the controller. We explain how we accomplish custom ranges in Section 3.3. In the remaining section, we describe the packet layout we employed in this work, which is depicted in Figure 3.2.

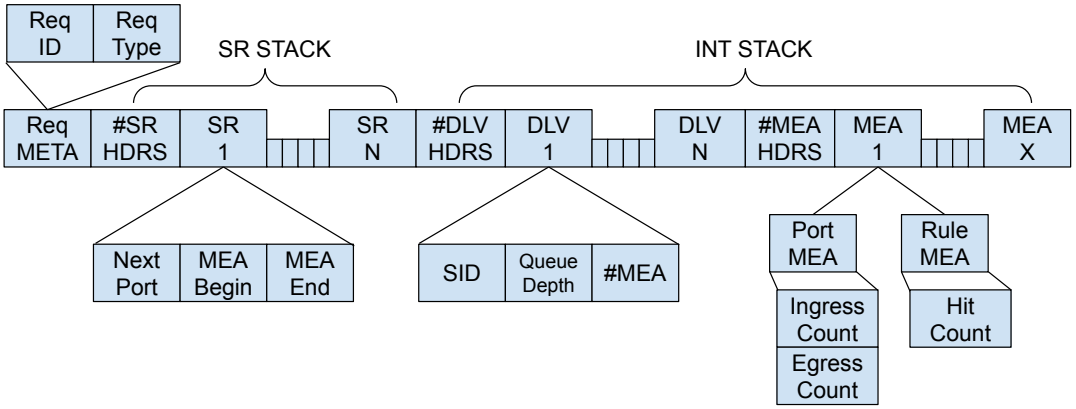


Figure 3.2: The packet layout used in this work.

The header layout consists of three different partitions: **Request Meta**, **SR Stack**, and **INT Stack**. The **Request Meta** contains two fields. With the **Request ID (15 bits)** field, we identify requests at the controller side to track which paths were collected successfully. The **Request Type (1 bit)** field specifies the custom measurement type. If the measurement type is zero, rule measurements will be collected, which counts the number of times a certain forwarding rule's been hit. Likewise, if it is one, we request the ingress and egress packet counters of certain ports to be collected.

The **SR Stack** accommodates the number of the SR headers (16 bits) and the header list itself. Every SR Header contains the Next Port (8 bits) to guide switches to forward INT requests correctly. It also has a measurement (MEA) range, **MEA Begin** and **MEA End** both of which are 16 bits. The custom measurements are stored in registers that can be indexed like normal arrays. With the measurement range, we lead switches to fetch only the portion of the information we require. This separation provides flexibility to the monitoring applications, which may require only some parts of measurements of some switches but different from others. Furthermore, in some network conditions, the monitoring application may want to keep INT packet size as small as possible. By utilizing measurement range, the application can adjust the size of collected information freely.

The last part of the layout is the **INT Stack**. It consists of two sub-stacks, including device-level (DLV) information and the custom measurements (MEA). In the device-level information stack, we have the number of DLV headers (16 bits) and the list of DLV headers. Each header has the switch id (8 bits) field, the queue depth (8 bits) field, and the number of measurements (16 bits) collected from a switch, calculated according to the measurement range specified in the corresponding SR header. Similarly, the custom measurement stack contains the number of measurements (16 bits) and the measurement headers. Each header contains counter values according to the request type specified in the request meta. If the request type is rule extraction, the size of the single header becomes 16 bits. Otherwise, it is 32 bits. Every switch may insert a different number of measurement headers, specified in the DLV header's number of the measurement field. The controller utilizes this information to group measurement headers correctly.

3.3 The Data Plane Design

Employing a new protocol or modifying existing ones requires instructions on how a switch can parse such packets. We capitalize on the flexibility and customization that the P4 language offers and redesign how switches parse INT packets and implement a custom recovery algorithm in ingress and egress pipelines.

3.3.1 Packet Parsing

In Figure 3.3, we present a flow chart diagram of the packet parsing process of switches. When a packet is received, we parse ethernet, IPv4 headers and check the transport layer protocol. If it is not UDP, we accept the packet directly, which means forwarding it to the ingress pipeline. Since we carry INT headers within UDP packets as a payload, they require special treatment on how we parse them. To identify INT-specific packets, we modify the Differentiated Services Field Codepoints (DSCP) field of the IPv4 header. If the DSCP field is something other than 0x17 or 0x19, we accept the UDP packet. When the DSCP field is 0x19, the packet is a type of recovery feedback containing only the Request meta. The meaning of recovery

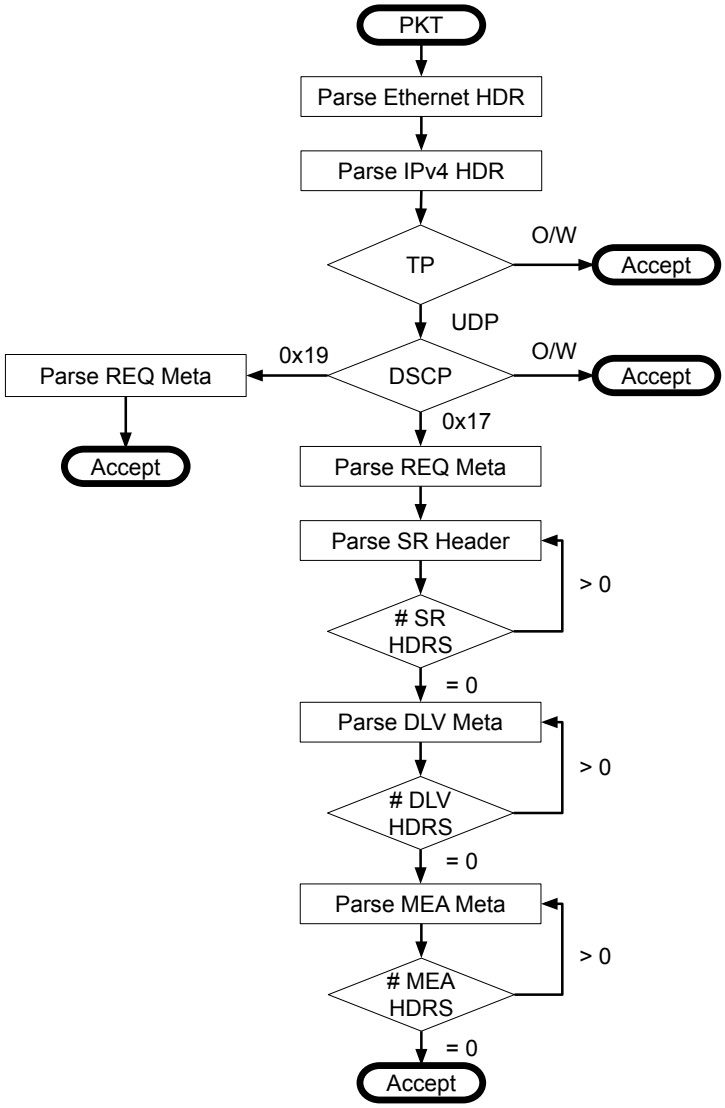


Figure 3.3: The flowchart of packet parsing.

feedback packets will be explained in Chapter 6. The last option is when the DSCP field is 0x17, which means UDP packet carrying measurements. We extract the Request Meta and obtain the number of SR headers in the packet. While the number of SR headers is not zero, we keep extracting SR headers. Then we repeat similar procedures to both DLV and MEA header stacks. One might argue that if the number of measurement headers is immense, it might slow down the packet parsing and, inevitably, the switch. Potential delays of INT packet parsing is yet another design question that we do not address in this work but provide effects of utilizing a different number of measurement fields in our analysis to lay a foundation for further works.

3.3.2 Ingress Pipeline

Once the packet gets parsed, it arrives at the ingress pipeline. If the packet is not an INT-typed packet, we update measurements, such as counters of which port it arrived from and which forwarding rule the packet hit. Otherwise, the packet's type can be either feedback or a probe packet carrying measurements. If it is a feedback packet, we read the packet's request-id and update a register that keeps track of received feedback packets. Afterward, we drop the feedback packet.

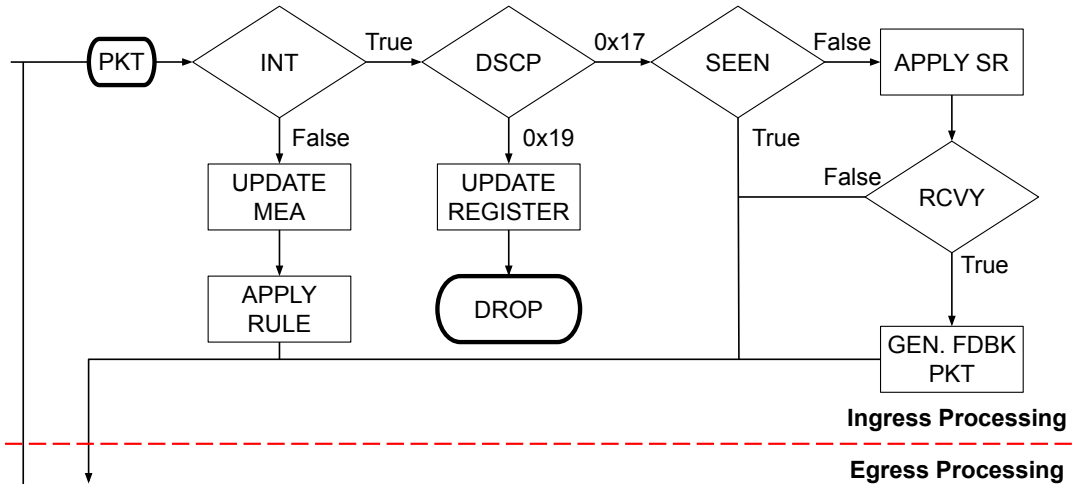


Figure 3.4: The flowchart of the ingress pipeline.

Offering custom measurement ranges requires us to create an artificial loop that is not supported by the P4 language. To accomplish this, we recirculate INT packets between ingress and egress pipelines until we reach the range limit. To reduce unnecessary computations, we identify looping packets and simply forward them to the egress pipeline. If it is the first time we observe an INT probe packet, we read the SR fields and adjust the port number accordingly. Then, we check if the controller has enabled recovery mode or not. If it is not enabled, we forward INT packets to the ingress pipeline. Otherwise, we generate a feedback packet destined to the previous hop and forward both packets to the egress pipeline.

3.3.3 Egress Pipeline

When a packet arrives at the egress pipeline, we perform a similar check where we figure the type of the packet. If the packet belongs to traffic other than INT packets, we forward them to their next destination. Otherwise, we need to check if the packet is cloned from the ingress or not. If it is a clone from the ingress pipeline, it means we need to convert the packet into a feedback type. Since a feedback packet does not require any information on board except the Request Meta, we truncate the packet. To distinguish feedbacks from regular INT packets, we mark these packets, which means changing the IPv4's DSCP field to 0x19.

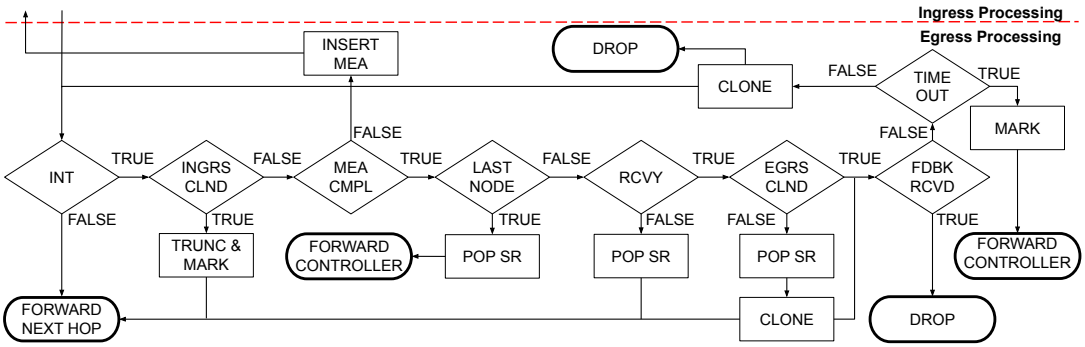


Figure 3.5: The flowchart of the egress pipeline.

When we observe a regular INT packet, we check if the customized range limit has been reached or not. If it is not, we insert a measurement field and recirculate the packet to the ingress pipeline. It is also possible to insert multiple measurement headers in one loop, but we opt against that option to measure if it would create considerable bottlenecks.

Once we complete the measurement insertion loop, we check if it is the last switch on the path. This information is deduced from the number of remaining SR headers. When the SR header count hits one, we understand that the current switch is the last hop in the path. The reason why it is one but not zero is that every SR header also carries custom measurement ranges. Therefore, we pop the last header and forward it to the controller. Otherwise, if the recovery mode is not enabled, we pop the current SR header and forward the packet to its next hop. On the other hand, if the recovery mode is enabled, we need to preserve a copy of the current INT packet. To gener-

ate a copy, we clone the current packet to the egress pipeline. Then, we forward the original packet. When the cloned packet is being processed again, this time, it goes through the egress-cloned check, indicating this is a recovery packet. For these packets, we check if feedback from the next hop has been received or not. If it is received, it means the INT packet passed through this switch successfully arrived at the next one. Hence, we do not need to store a copy of that packet and safely drop it. While we do not receive a feedback packet from the next hop, we preserve the packet for a predetermined time that the controller can adjust. The preservation requires us to clone the packet and drop one of them since we do not want to notify the controller before timeout happens. If we do not receive a feedback packet and timeout occurs, we set the IPv4 DSCP field of the preserved packet to 0x21 to indicate that the feedback packet has not been received within the time limit. Then, we forward the packet to the controller.

In the following chapter, we formulate the requirements and propose the BSIP.

CHAPTER 4

BALANCED SIMPLE INT-PATH PROBLEM

In this chapter, we define the Balanced Simple INT Path generation Problem (BSIP). First, we give the necessary constraints to generate simple paths. Then, we formalize the requirements, defined in Chapter 1.1 and complete the definition of BSIP.

4.1 Path Generation Constraints

We consider the network as a connected undirected graph G consisting of only P4 programmable switches, which we represent with V and e. V represents the node set, and e is a matrix representation of edges, where $e_{vu} = e_{uv} = 1$ if v and u are connected in G. We use paths and INT paths interchangeably, and we always mean INT paths.

Our main objective in this work is to generate simple (i.e., cycle-free) paths such that the requirements we listed in Section 1.1 are met and well-optimized. We discuss the advantages of simple paths over cyclic paths at the end of the section, where we layout objective functions. We assume that the controller decides the number of required paths using a fractional value q (i.e., between zero and one) depending on the system conditions. We use q to calculate $k = |V| \times q$, which gives us the number of paths to be generated. However, instead of applying k strictly, we use it as an upper bound to explore smaller values in constraint (4.1). The reason behind this decision is to minimize the effect of k in the search for an optimum number of paths at the cost of increasing the search space. Consequently, the size of P, the set of paths, is always k, but the optimum number of generated paths can be less than k. Therefore, we name k as the suggested maximum number of generated paths. We use x_p as an

optimization variable that indicates if path p is selected to be used as an INT-path part of the optimal solution.

$$\sum_{p \in P} x_p \leq k. \tag{4.1}$$

To generate simple paths, we have to cover the following restrictions.

1. Every node in a path should have a degree of one or two.
2. Path p should be acyclic and connected. In other words, we should be able to traverse every node in path p exactly once.

The first step is to represent given network topology so that it is possible to keep track of incoming and outgoing edges for every path which will come handy to satisfy both of the restrictions. To do so, we create an optimization variable, vector of adjacency matrices ξ_{pvu}, whose size is $k \times |V| \times |V|$ to keep track of incoming and outgoing edges for every node in every path. With constraint (4.2), we ensure that there can be an edge from node v to node u if and only if both are in the same path p, and they are neighbors in the original topology (i.e., $e_{vu} = e_{uv} = 1$). We use an inequality since there might not be an edge from v to u in path p even though they share an edge in the original graph. Consequently, when ξ_{pvu} is zero, it indicates that there is no edge from v to u in path p.

$$\xi_{pvu} - \delta_{pv}\delta_{pu}e_{vu} \leq 0 \qquad \forall p \in P, \tag{4.2}$$

where δ_{pv} is an optimization parameter indicating whether v is part of p, $\delta_{pv} = 1$. If vertex v is not in path p, then $\delta_{pv} = 0$.

In the first restriction, we want every node $v \in p$ to have a degree of one or two to construct p as a simple path. With our graph representation variable, ξ_{pvu}, we can easily define necessary constraints (4.3) and (4.4) to ensure that every path satisfies the first restriction. If node v has one incoming or outgoing edge, it can be considered as an endpoint. Similarly, if v has both incoming and outgoing edges, it is considered

as an intermediate node in the path:

$$\sum_{\substack{u \in p \\ u \neq v}} \xi_{puv} \leq 1 \qquad \forall v \in p,\ \forall p \in P, \tag{4.3}$$

$$\sum_{\substack{v \in p \\ v \neq u}} \xi_{pvu} \leq 1 \qquad \forall u \in p,\ \forall p \in P. \tag{4.4}$$

The degree constraints themselves are not enough to guarantee that p is a simple and connected path. Therefore, we use the second restriction to complete the connected simple path generation. The first thing we focus on is eliminating subcycles in a path. To do so, we enforce Miller-Tucker-Zemlin's (MTZ) constraint of the traveling salesman problem [30] onto every path. In essence, MTZ enforces ordering among nodes in a path. In the ordering procedure, if there is an edge from node v to u, then MTZ requires the order of u to be higher than v. If they do not share an edge, both counter values would be independent and can get any other value. We use counter variables, β_{pv}, to determine the ordering of nodes in every path. Since we already keep track of incoming and outgoing edges with variable ξ_{pvu}, we can easily introduce MTZ constraint:

$$\beta_{pv} + 1 + |V|(\xi_{pvu} - 1) - \beta_{pu} \leq 0 \qquad \forall v, u \in p. \tag{4.5}$$

Let us briefly show that MTZ indeed eliminates subcycles. Assume that there exists a subcycle with following edges, $v \rightarrow u \rightarrow t \rightarrow v$. Such a cycle would create the ordering of $\beta_{pv} < \beta_{pu} < \beta_{pt} < \beta_{pv}$. However, β_{pv} cannot be both less and greater than β_{pu} at the same time, which contradicts MTZ constraint.

Even though MTZ eliminates subcycles in path p, it does not guarantee that p is connected. In other words, there might be multiple vertex isles. Since we are confident that different vertex isles cannot contain cycles, it is trivial to generate the complementary constraint that enforces p to be connected:

$$l_p - \sum_{v \in p} \sum_{\substack{u \in p \\ v \neq u}} \xi_{pvu} = 1 \qquad \forall p \in P, \tag{4.6}$$

where we introduce strict equality between the number of nodes and the number of edges in p, where l_p refers to the length of path p. If the total number of edges is one less than the l_p, then p has to be connected since it cannot contain cycles. Let us

briefly prove that it is indeed a fact. Assume that p contains two disconnected vertex isles. It means that one of the isles must contain a loop to satisfy the constraint (4.6). However, this is a contradiction since we know that isles are cycle-free due to MTZ constraint.

Every selected simple path p has to respect the link capacities. Accordingly, we define constraint,

$$d_{vu} + \sum_{p \in P} (\xi_{pvu} + \xi_{puv}) x_p l_p h \leq c_{vu} \qquad \forall v, u \in V, \tag{4.7}$$

to ensure that link capacities are not exceeded. Accordingly, we assume that links have an initial load, d_{vu}, and maximum capacity, c_{vu}. Although the representation is directed, (v, u) and (u, v) point to the same edge in the original graph. Hence, to avoid duplicate rules, we consider directed combinations of (v, u) pairs such that (u, v) is not considered again. We use h to represent the fixed INT header size and calculate the overhead that INT headers create over link (v, u).

In the following section, we complete the BSIP definition by introducing our objective functions.

4.2 Objective Function Definitions

Now that we have the necessary constraints to generate simple paths within link capacities, we can address the requirements listed in Chapter 1.1. We translate them as our objective functions and completes the definition of BSIP.

Requirement 1 emphasizes network coverage with an optimum number of INT paths. For the network coverage, every node must belong to exactly one path. However, such constraint directly depends on the topology structure, as, for instance, star-topology does not allow to have completely disjoint paths. Therefore, we first relax that requirement letting shared nodes between non-cyclic paths. Another way to break that dependency would be by using cyclic paths, whose drawbacks are discussed afterward. Accordingly, we enforce that a node $v \in V$ should belong to at least one

simple path $p \in P$ with constraint,

$$-\sum_{p \in P} \delta_{pv} x_p + 1 \leq 0 \qquad \forall v \in V. \tag{4.8}$$

Furthermore, this constraint ensures that at least one path is generated as a result.

There is a tradeoff for the path generation. While several short-length INT paths increase the number of INT packets generated and processed by the controller, fewer long-length paths impose a higher delay to deliver INT packets even though the controller has less processing burden. As an implication of the *Requirement 1*, we define the objective function, BSIP-N,

$$\min \ \chi = \sum_{p \in P} x_p, \tag{4.9}$$

to minimize the number of selected paths to reduce the load on the controller. Even though Objective (4.9) seems to be favoring longer paths, *Requirement 2* focuses on the path lengths to find a balance between long and short paths. In this work, we assume that the switches (nodes) are homogenous. Consequently, the length of paths directly affects the arrival time of INT paths due to the processing, queuing and propagation delays. In other words, longer paths spend more time in the network than shorter paths. During that additional time, the already delivered shorter INT paths wait and age in the controller. When the longest path arrives, the shorter INT paths already refer to an older state of the network, which hinders a consistent network view. On the other hand, if the deviation of INT path length is close, the paths will arrive concurrently at the controller, which minimizes the aging time of smaller paths. Furthermore, we realize that if INT paths are cyclic, then previously added INT headers age unnecessarily as revisited nodes introduce additional delays. Therefore, we prefer simple paths to decrease the aging factor of existing INT headers. We present the objective function, BSIP-D, which minimizes the deviation of INT path lengths:

$$\min \ \Psi = \max\{x_p l_p\}. \tag{4.10}$$

As one can notice, one way to minimize the length deviation is to generate joint paths. However, such an approach would maximize the redundant information and load on the data plane. In other words, whenever paths intersect, they introduce redundant information and enlarge the INT packets. Furthermore, intersection points (i.e., shared

switches) need to forward more INT packets. Consequently, INT overheads on shared switches increase and may become a bottleneck in the network. Therefore, with *Requirement 3*, we emphasize minimizing the redundant information and distributing the load equally to the data plane. To fulfill this requirement, we define the last objective function, BSIP-S,

$$\min \ \Theta = \sum_{p,t \in P} \sum_{v \in V} \delta_{pv} \delta_{tv} x_p x_t, \tag{4.11}$$

which minimizes the number of shared switches.

The optimum solution should address all three objective functions, BSIP-N, BSIP-D, and BSIP-S. Therefore, we define BSIP-A,

$$\min \ \chi + \Psi + \Theta. \tag{4.12}$$

For completeness, we provide the representation of the formulation below, which is simply the combination of what we described.

$$\begin{aligned}
&\min \ \chi \ + \ \Psi \ + \ \Theta && && (4.13)\\
&\text{s.t.} \ \sum_{p \in P} x_p \leq k && , \\
&-\sum_{p \in P} \delta_{pv} x_p + 1 \leq 0 && \forall v \in V, \\
&\xi_{pvu} - \delta_{pv} \delta_{pu} e_{vu} \leq 0 && \forall p \in P, \\
&\sum_{\substack{u \in p \\ u \neq v}} \xi_{puv} \leq 1 && \forall v \in p, \ \ \forall p \in P, \\
&\sum_{\substack{v \in p \\ v \neq u}} \xi_{pvu} \leq 1 && \forall u \in p, \ \ \forall p \in P. \\
&l_p - \sum_{v \in p} \sum_{\substack{u \in p \\ v \neq u}} \xi_{pvu} = 1 && \forall p \in P, \\
&\beta_{pv} + 1 + |V|(\xi_{pvu} - 1) - \beta_{pu} \leq 0 && \forall v, u \in P, \\
&d_{vu} + \sum_{p \in P} (\xi_{pvu} + \xi_{puv}) x_p l_p h \leq c_{vu} && \forall v, u \in V.
\end{aligned}$$

4.3 Search Space Analysis

In the BSIP formulation, there are $k \times (|V|^2 + |V| + 1)$ number of binary and $k \times (|V|+1)$ number of integer variables. Additionally, the formulation includes quadratic constraints so that we can define our graph representation ξ_{pvu}. Therefore, the BSIP is a mixed-integer problem with quadratic constraints. Even though the number of variables does not depend on the number of edges a topology has, increasing the edge count increases the options to be considered by the optimizer. For instance, if there is no edge between u and p in the graph, then there will be a strict constraint of $\xi_{pvu} = 0$ for all paths. In the other case, the constraint becomes $\xi_{pvu} \leq 1$.

As we analyze in Chapter 7.1, the problem becomes infeasible rather quickly, even for small topologies. Consequently, deploying optimization models on the controller is not suitable since the controller will need to run path generators multiple times, most likely with different conditions.

CHAPTER 5

GRAPH PARTITIONED INT

In this chapter, we cover our heuristic path generator, Graph Partitioned INT (GPINT), addressing the BSIP. First, we point out why there is a need for a heuristic approach. Then, in Section 5.1, we provide the necessary information about KernighanLin's graph partitioning algorithm, which we use as a basis for GPINT. Afterward, we explain our extension GPINT in Section 5.2 and conclude this chapter with complexity analysis.

Based on the requirements, BSIP addresses three main objective functions to achieve efficient INT-based monitoring: (4.9) minimize the number of generated INT paths that cover the network to reduce overhead, (4.10) minimize length deviation for concurrent path collection, and (4.11) minimize overlapping simple (i.e., loop-free) paths to minimize redundancy. Suppose we formulated the BSIP in a way that we generate simple paths prior and select the most suitable ones. Note that there is an exponential number of candidate simple paths, which can be selected with respect to different criteria. For example, one implication of the Requirement-2 is to minimize path length deviation. Favoring simple paths with only a fixed-length m reduces the search space and, eventually, the complexity to $O(|V|^m)$, which is less than generating every simple path. However, in such a case, there may not be a group of paths that addresses the Requirement-3, minimizing redundant information via disjoint paths. Consequently, we would need to generate all possible simple paths to satisfy the given three requirements. Please note that generating the number of simple paths depends on both the number of vertices and edges in the topology. In our formulation, BSIP's search space grows in the order of $k \times |V|^2$. The number of edges in the topology directly correlates to the number of constraints we have in our formulation. Hence, even though

an increase in the number of edges does not directly affect the number of variables in the problem, it still contributes to the model's complexity. Since the networks are likely to evolve (addition of new nodes, disconnected nodes), exponentially growing search space and the complexity make deploying optimization models infeasible in real-life scenarios. It leads us to develop a heuristic to solve this multi-objective problem within a reasonable time on the controller. In the following section, we present an overview of Kernighan-Lin's algorithm, which we use as a basis of our proposed approach.

5.1 Overview of Kernighan-Lin's Graph Partitioning Algorithm

The original Kernighan-Lin's graph partitioning algorithm addresses the problem of dividing a given graph that contains $2|V|$ nodes into two parts, each of $|V|$ size, such that the weights of the edges crossing between two parts are minimized. Roughly, the algorithm contains three stages, initial partitioning, exchange, and repetition phases. In the initial partitioning phase, the algorithm starts by generating two parts with a size of $|V|$. Initially generated partitions are not expected to be optimal, so the algorithm proceeds to the exchange phase. In the second phase, the algorithm exchanges nodes between two partitions in a greedy fashion until every node is visited. The greedy part comes from selecting the node pair that reduces the cost the most. In the original algorithm, it is the weight of a crossing edge. Once a pair is selected, the algorithm saves how much gain is achieved with the selected pair. Once it visits every node, the exchange phase terminates, returning the set of changes it made, sorted in a descending order with respect to how much gain each change achieves. In the next phase, repetition, the algorithm calculates the maximum cumulative gain that can be achieved with the result of the exchange phase. If the cumulative gain is positive, the algorithm applies all the changes and deduces that there is still room for improvement, and calls the exchange phase. The algorithm stops when the cumulative gain is negative, which implies the best partition it can generate has been found. Throughout the literature, there have been many extensions on Kernighan-Lin's graph partitioning algorithm, such as $k-$way partitioning [69, 70], multi-level Kernighan-Lin [71]. In this work, we utilize the main idea of the Kernighan-Lin, the three-phase implementation,

and extend it to generate INT paths that satisfy the requirements.

5.2 Graph Partitioned INT

We extend Kernighan-Lin's graph partitioning algorithm to design GPINT within two main principles:

1. We consider simple k-paths at the beginning, which can only grow or shrink from their endpoints, unlike subgraphs that can exchange any nodes between them.
2. We require paths to cover the entire graph and allow them to overlap.

Accordingly, GPINT contains three stages, initial partitioning, exchange, and improvement or repetition.

5.2.1 Initial Partitioning Stage

In stage (i), we perform an initial partitioning algorithm, as described in Algorithm 1. The algorithm accepts the graph, $G(V, E)$ and k, where the V is the set of vertices with additional information of requested information, E is the set of edges that also contains edge capacity and load, and k is the suggested maximum number of paths. We start by sorting the nodes with respect to their degrees (line 1), the number of edges they have. Our intuition behind this operation is that a node is more likely to be an endpoint if it has less degree than others. After the ordering, we start traversing every node on the sorted vertex list. If a node belongs to a path (line 4), we move on to the next node since this phase aims to construct paths that can be used as a foundation in further phases. Otherwise, we create a path with only the current node within (line 6) and set a flag to indicate if the current path will belong to the result or not that we will explain shortly. Now, we are ready to start the growth phase of our current path, which lasts until we can no longer expand our path due to either (i) we cannot find suitable nodes, (ii) reach the maximum length of $\lfloor \frac{|V|}{k} \rfloor + 1$, (iii) hit the MTU or (iv) link limitations (line 8). We can hit the MTU limits before reaching the

maximum length since the monitoring framework may request different information from each node. Similarly, the monitoring framework may adjust link loads or even disable some to avoid using them for INT packets for some reasons. Afterward, we attempt to grow the current path with function *grow_path* (line 9). In this function, we iteratively check neighbors of the path's endpoints to find if there are any unvisited (i.e., does not belong to any path) nodes until either we reach the maximum length, MTU, link limitations, or cannot find such node. We grow paths from both ends equally, which is similar to performing BFS. When this function finally returns, we check the path's length (line 10). In this phase, we try avoiding generating paths with length one. Hence, if the path's length is one, we check the current node's neighbors to see if any of them are endpoints of a suitable path (line 11). Here the suitable path refers to a path that has stopped growing due to length limitations (line 8) rather than MTU or link limitations. If we can find such a neighbor, to eliminate the path with length one, we relax this condition by appending the current node to the found path (line 13). Then, we mark the current path to be invalid and break the loop. Otherwise, if there are no such neighbors, we look for a neighbor with the most unvisited neighbors or a neighbor whose edge with the current node is under its capacity (line 17). If we can find such a neighbor, we grow the current path with that node (line 19). Otherwise, it means we cannot find a neighbor to grow the current path and break the growth loop. Lastly, we check if the current path is a valid one (line 22). If so, we append it to the result. After traversing every node, we return the resulting partitions P_i (line 24).

5.2.2 Exchange Stage

In the exchange stage, (ii), we perform Kernighan-Lin like exploration to target the requirements pointed out in Section 1.1, as described in Algorithm 2. Algorithm 2 accepts $G(V, E)$ and P as an input, where the P is the generated paths from either previous iteration or the initial partitioning phase. We first calculate the local objective value o_l (i.e., objective 4.12), and generate an empty ruleset $\mathcal{L}$ to store the proposed rules (lines 1-2). At each iteration, we generate an empty ruleset $\mathcal{R}$ where we stash possible improvement rules (line 4). For every path, we first check if they are suitable for a growth operation. The suitability of a path means the information carried in a

Algorithm 1 Initial Phase of GPINT

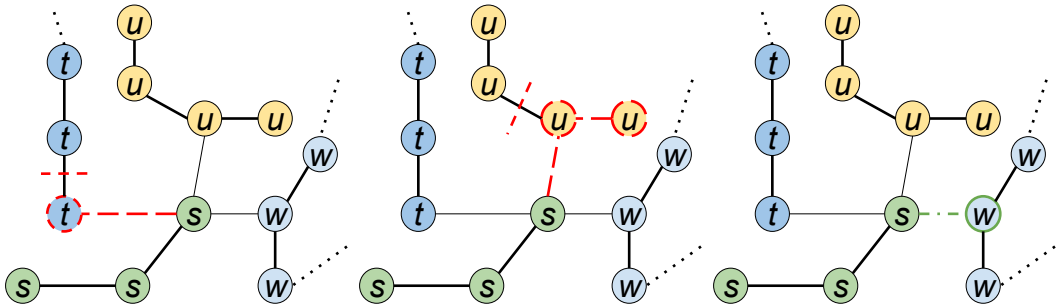
Input: $G(V, E)$, k

Output: P_i

```
1:  V_s ← sort_nodes(V)
2:  P_i ← {}
3:  for each v ∈ V_s do
4:      if belongs_to_path(v) then
5:          continue
6:      p ← create_path(v)
7:      erase_path ← False
8:      while can_grow(p) do
9:          grow_path(p, v, G, k)
10:         if p.len() = 1 then
11:             if can_join_n_path(v, P_i, G) then
12:                 p_n, n_node ← get_n_path(v, P_i, G)
13:                 p_n.grow(n_node, v)
14:                 erase_path ← True
15:                 break
16:             else
17:                 b_node ← select_best_neighbor(v, P_i, G)
18:                 if b_node then
19:                     p.grow(v, b_node)
20:                 else
21:                     break
22:     if not erase_path then
23:         P_i ← P_i ∪ p
24: return P_i
```

path should be less than the MTU supported by the links and link capacity should not be exceeded with new headers (line 6). If a path is suitable, we only consider its endpoints, head, and tail, as potential growth points and check for unvisited endpoints (line 6). If we find one, we have three options to grow the current path, (ii-a) capture an endpoint of a neighboring path, (ii-b) divide a path from an intermediate node, or (ii-c) overlap with a longer neighboring path. In Figure 5.1, we depict an example representation of available exchange options from path *s*' point of view, which we cover shortly. For all the operations, we make sure to remain under MTU limitations

and link capacities.



(a) Option (ii-a) (b) Option (ii-b) (c) Option (ii-c)

Figure 5.1: Illustration of available exchange options from path s' point of view.

In option (ii-a), we check the growth points' neighbors to find an endpoint of a different path for a potential capture operation (line 8). If we can find such a neighbor, we try capturing, which is to remove a node from its original path and append it to the current path (Figure 5.1a). If the original path has only two nodes, we capture both since we do not allow a path to contain a single node. As a result, we return $\mathcal{C}$, containing the changes made, which is a set of rules including how lengths of involving paths and number of overlapping paths have changed as well as the involving nodes themselves. This time, in option (ii-b), we instead consider any neighbors of growth points to capture, which we refer to as the division of a path (line 11). We consider a path to be divisible from node x if (d.1) its length is greater than three or if (d.2) its length is three, at least one of the side nodes (i.e., x's left and right neighbors in the path) has to belong to another path. The reasoning behind (d.1) is to avoid generating paths with length one. Consequently, we also avoid division loops that may arise from those paths and reduce the complexity. In the case of (d.2), if a side node belongs to another path, removing that node from the target path reduces the overlapping paths. As a consequence, we can disband the target path completely, which would decrease the objective value. Now, for the division operation, even if the target path is divisible, we still need to check node x's length of left and right sides as one of them can be one. If so, we check whether it belongs to another path, or we can capture that remaining node as well (Figure 5.1b). If both of the options hold, we generate the rules with the best outcome, which reduces the objective value the most. If we cannot capture the remaining node and it does not belong to another

path, we abort the division operation for the given target path. Otherwise, we capture the remaining node as well. As a result of this operation, we generate the $\mathcal{D}$ ruleset, which contains the changes made with involving nodes.

Lastly, in option (ii-c), we search neighboring paths to find a longer path (line 12). The intuition behind this decision is that we realize if a path overlaps with a smaller one, it only increases the value of objective functions, which is unlikely to be applied. On the other hand, if we overlap with a longer path (Figure 5.1c), there is a chance to lower the objective value. Therefore, we only check for a longer path to reduce unnecessary computations.

After we consider all of the options above, we calculate the objective value changes for every generated ruleset (line 13). Then we save these rules with their achieved gains to $\mathcal{R}$ (line 14). After we cover all of the endpoints, we pick the rule with the most gain that respects link capacities from $\mathcal{R}$ (line 15). We save the selected rule to $\mathcal{L}$ and apply it locally (lines 18-20). In the local application, we update visited nodes with the nodes involved in the selected rule and update local copies of paths accordingly. Local changes allow us to build the decisions incrementally, which is one of the key features of Kernighan-Lin's algorithm's structure. At the end of this stage, we return the ruleset $\mathcal{L}$, which stores the best local improvements.

5.2.3 Repetition Stage

In the last stage, (iii), we perform another essential feature of Kernighan-Lin's algorithm, running stage (ii) iteratively until we can no longer find any local changes that improve objective functions. For every local change we find in stage (ii), we calculate the cumulative gain and find the peak point index, that is, the point where if we include the next rule, the cumulative gain will decrease. If the peak point is negative, we deduce that the algorithm can no longer find an improvement and return the generated result up to that point. On the other hand, in the case of a positive peak point, we apply all local changes up to the peak point on P and continue repeating this stage until we reach the iteration limit or point where we can no longer generate improvements.

Algorithm 2 Exchange Phase of GPINT

Input: P, G

Output: $\mathcal{L}$

```
o ←: calculate_local_objective(P)
L ←: ∅
while (not all endpoints of P is visited) do
    R ←: ∅
    for each p ∈ P do
        if not p.can_grow() then
            continue
        ends ←: unvisited_endpoints(p)
        for v ∈ ends do
            C ←: capturable_endpoints(v, P)
            D ←: dividable_nodes(v, P)
            N ←: longer_neighbors(v, P)
            gains ←: calculate_objective_changes(o_l, N, C, D)
            R ←: R ∪ generate_rules(gains)
    B ←: best_improvement(R, G)
    if B is ∅ then
        break
    L ←: L ∪ B
    o_l ←: update_local_objective(o_l, B)
    update_visited_endpoints(P, B)
return L
```

5.2.4 Complexity Analysis

The complexity analysis of the initial partitioning phase is similar to the BFS. We first sort the nodes with respect to their degrees, which is $O(|V| \log |V|)$. The only difference from BFS is for some nodes, we sort their neighbors based on the number of unvisited nodes they have. In the worst case, if the graph is complete, the sorting introduces $|V| \log |V|$ of additional complexity. When we combine the BFS and the sorting parts, we have $O(|V| \log |V|) + O(|V| + |V|^2 \log |V|)$. In the exchange phase, we iterate k times, and at each iteration, we check k paths and their endpoints. For every endpoint, we check their neighbors, which is $|E|$. The exchange phase's overall runtime complexity then becomes $O(2k^2|E|)$, or $O(k^2|V|)$ in case of a fully

connected graph. We realize that the initial partitioning phase is more expensive than the exchange phase. In return, after the initial partitioning phase, we generate relatively good partitions, which reduces the number of times we run the exchange phase. As we explore in Chapter 7, the GPINT can generate near-optimal results, and it is scalable.

CHAPTER 6

DATA RECOVERY FOR IN-BAND NETWORK TELEMETRY

The process of data collection is vulnerable to network failures, whether it is the traditional pull, or push mode, or INT. Especially in the case of INT, the information of multiple devices is accumulated in a single packet. Consequently, a single INT packet loss can blackout a large portion of the network in terms of available information and can hinder the analysis may be more than other frameworks. However, whether it is INT or not, the underlying framework should provide data recovery mechanisms in place to minimize the information loss. Hence, the monitoring system can detect network failures (device or link failures) as quickly and accurately as possible. To the best of our knowledge, there has been no prior work on data recovery for INT protocol so that the controller can obtain measurement even in harsh conditions. Therefore, in this work, we implement and combine an existing packet loss recovery technique, SQR [33], with INT so that we can prevent accumulated data loss and initiate data recovery for affected INT paths.

In this chapter, we first give the necessary information about how SQR operates. Then, we explain how we integrate it with INT packets and propose our data recovery for INT protocols.

6.1 SQR: Recovery for Commercial Packet Losses

According to Gill et al. [72], the link failures are more common than the device failures in their data centers. The motivation and idea behind SQR are to protect packets against such link failures in the data-plane to prevent data loss. Hence, they mask the failure in the data plane so that the end hosts are unaware of any failures and

transmit packets as if everything is normal. By doing so, end hosts do not retransmit the affected packets, and consequently, both the throughput and latency levels are maintained.

At the implementation level, the programmable switches store every packet and send their copies according to their rule tables. Once a packet arrives at the next hop, it generates a new feedback packet in the reverse direction to acknowledge that it received the packet successfully. If everything goes well, the current switch discards the stored packets destined in that direction. However, suppose the current switch does not receive a feedback report within a limited time window. In that case, the switch realizes that the destination link is broken and continues storing the packet until the maintainer updates the rules. After the second delay, the switch sends the packet through the backup port, set by the controller during the delay.

6.2 Enabling Data Recovery For In-band Network Telemetry

In our integration, we employ the packet storing mechanism of the SQR on only INT packets. We forward INT packets according to their SR headers but store them until the next hop returns a feedback message. If we receive a feedback message within a certain time window, we drop the stored INT packet destined to that hop. Otherwise, we deduce that there might be either a link or switch failure in the data plane and send the stored INT packet to the controller for further analysis. Before forwarding the stored INT packets to the controller, we mark their DSCP field of the IPv4 header to 0x19 so that the controller can distinguish incomplete INT paths from completed ones. Accordingly, the controller can initiate recovery probes for the remaining switches in the failed path.

In Figure 6.1, we depict an example scenario of how the data recovery module behaves and functions. In this example, we divide switches' points of view with dotted lines. If a packet is going through or coming into the switch, it means the switch observes them and is aware of their existence. This scenario starts with the controller releasing a probe packet to the network over switch *s1* and awaits for T_f (feedback time window) seconds for a response. If the recovery module did not receive such a

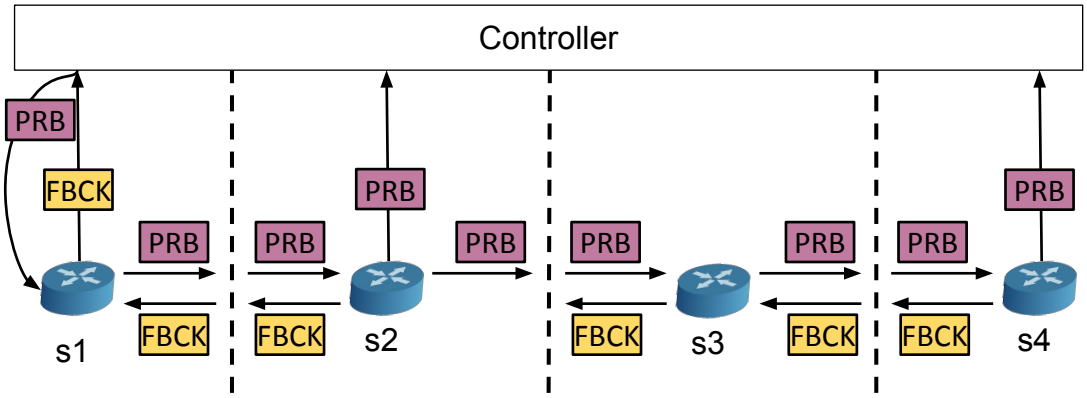


Figure 6.1: An example scenario of how the data recovery module functions.

packet, it would mark the switch and try R_a (number of recovery attempts) times to initiate the INT path from *s1* after T_r (time to initiate recovery) seconds. If there were no responses after R_a times, then the controller would assume *s1* as unresponsive and initiate INT path from *s2*. However, in this case, *s1* generates a feedback packet containing only the Request metadata of the INT probe and sends it to the controller. As the controller and data recovery module receive this feedback packet, the data recovery module deduces that the *s1* is responsive and marks it so. When *s1* finishes inserting requested measurement fields, it forwards a copy of the INT packet to the next hop according to SR headers and awaits T_l (looping time window) seconds for a feedback packet from the next hop. As *s2* receives INT packet, it generates and sends a feedback packet to *s1*. In this figure, we observe that the feedback packet reaches to *s1* within T_l seconds. Hence, it drops the stored INT packet. Then *s2* forwards the INT packet after inserting requested measurements to *s3*. On the other hand, *s2* does not receive a feedback packet in T_l seconds. Therefore, it sets the accumulated INT packet's DSCP field as 0x21 and forwards it to the controller for further analysis. When the controller receives a marked INT packet, the data recovery module understands that a potentially broken path has been detected in the data plane. Once an INT packet arrives at the data recovery, it performs several actions. If the INT packet represents a complete path, which is the best scenario, it marks all of the switches and links in that path as responsive. On the other hand, if the packet is incomplete, it may need to initiate a recovery probe for the incomplete path's remaining switches. For instance, in this example figure, the recovery module knows that up to *s2*, switches are

responsive, but it is *s3* that may not be so. It is still possible that it was the feedback packet that got lost but not the INT packet sent from *s2* to *s3*. Hence, the INT path that was thought to be incomplete at *s2* may still be continuing its journey through the network, collecting measurements, which is the case in this scenario. Since the data recovery module cannot know which is the case at the current moment, it provides a customizable option. It waits up to T_r (time to initiate recovery) seconds before declaring that the path is indeed incomplete and should continue from *s3*. If, for example, we receive the complete path within T_r, the data recovery module deduces that it was a false alarm raised by *s2* and marks the link between *s2* and *s3* as there might be high load on it. However, if that is not the case, after T_r seconds, we still do not know whether *s3* is working properly or not. Hence, to understand if it is working or not, the recovery module starts a path from *s3* and applies the same scenario that it applied to *s1*. Once the monitoring framework receives all probe paths successfully, the data recovery module presents its findings to the controller, including marked links and switches. In this scenario, *s3* is functional and forwards probe packet to *s4*, which successfully finishes telemetry insertion and sends the probe packet to the controller.

We explain the implementation details in Section 6.2.2. In the following section, we discuss how the customizable parameters affect the behavior of the data recovery module.

6.2.1 Parameter Discussion

As we explained the figure, we introduced several parameters. Let us discuss how they can customize and change the recovery module.

T_f indicates time to wait for a feedback packet from the first switch in the INT path. If T_f is too short, the recovery module can mark paths too quickly, and with the correlation of T_r, it may lead to generate too many recovery packets as a result. On the other hand, if it is too long, the recovery module may not deduct fast enough that the path's first switch is unresponsive.

T_r represents the time to wait before attempting recovery of an incomplete path. When the data plane forwards an incomplete path, there is still a possibility that the

supposed to be lost INT packet still in the network and collecting measurements. Hence, if the T_r is short, we may overreact and generate many INT packets before the INT probe can finish its collection. However, if it is too long, we cannot collect measurements quickly enough to ensure seamless recovery.

R_a corresponds to the number of recovery attempts. While many attempts can provide more concrete results whether a switch is unresponsive or not, it also delays the data collection from the rest of the path. On the other hand, fewer attempts can mark switches unresponsive relatively quickly and can collect measurements from the rest of the switches in the path. If, for example, we receive a result of a marked switch from any other path, we can still correct and mark the switch responsive.

T_l corresponds to how long switches are going to store INT packets as they wait for a feedback packet from the next hop. If it is too long, switches can consume too many resources, especially when the measurement frequency is high. However, if it is too short, then switches will react too fast and generate many incomplete paths, increasing the controller's load and the number of false positives.

T_{ro} is the resolution time the module uses as it waits for incoming packets from the packet receiver. We explain the usage of this variable as we cover the implementation details. However, in short, it affects how frequently we check the timeout events that occur due to exceeding either T_f or T_r. The smaller values increase the controller's load as it turns into a busy waiting but achieve faster timeout event detection. If the data recovery module is deployed as a separate entity to the controller, minimizing this value would be better as there would not be any increase in the controller's load.

Correctly determining optimal values for these parameters is potentially another optimization problem or even a challenge for machine learning-based management systems. For instance, the T_f and T_r depend on the link qualities unique from a network to network. Similarly, R_a depends on network conditions. As an example, if the management entity is aware that the network is under heavy load, it would be crucial to deduce whether switches are responsive and minimize false positives. In which case, R_a can be set to a higher value. Lastly, the T_r depends on the performance of the underlying INT probe generator, where the performance is the measure of how fast the controller can collect measurements under various loads. Hence, the pre-

diction of performances of probe generators can fine-tune the recovery module quite significantly.

In addition to the customizable variables, we also support an aggressive mode for the data recovery module. In this mode, whenever the controller detects a failure, it immediately generates a recovery packet without waiting for T_r seconds. Precedentely, this mode increases the number of generated paths and false detections of link congestions.

We leave determining the correct values as future work and use more generic values to display the data recovery module's applicability and effectiveness. However, to show its customizability, we employ a fine-tuned version of the parameters for GPINT and discuss the effect of changes in parameters.

6.2.2 Implementation Details

The behavior described over the Figure 6.1 can be realized with various implementations. In this section, we explain our implementation in detail.

In Algorithm 3, we have a pseudo-code of the data recovery module's main loop. The data recovery module is spawned as a thread by the controller and the controller provides P_r, *pending_feedbacks*, *recovery_queue*, and $\mathcal{A}$. P_r contains customized data of generated paths, which includes the following main attributes, which will be explained as we go through code snippets:

- a unique identifier,
- a timestamp to indicate deadline of initiating recovery,
- the latest index that the module observed,
- generated path metadata which includes specified requests from each switch as well as the generated path,
- a map object to distinguish different recovery packets from each other.

Algorithm 3 Data Recovery Module

Input: P_r, pending_feedbacks, recovery_queue, $\mathcal{A}$

Output: $\mathcal{A}$

feedback_timestamp $\leftarrow$: time.now()
while True **do**
    measurement_packet $\leftarrow$: packet_queue.get(to=T_{ro})
    if timeout then
        if time_elapsed(feedback_timestamp, T_f) **then**
            for each $p\_id \in$ pending_feedbacks **do**
                $p \leftarrow$: P_r[p_id]
                if not p.in_recovery_queue **then**
                    if p.attempts > R_a **then**
                        notify_measurement_framework(p)
                        p.increment_index()
                    p.attempts $\leftarrow$: p.attempts $+1$
                    insert_to(recovery_queue, p)
            feedback_timestamp $\leftarrow$: time.now()
        process_recovery_queue(recovery_queue)
        continue
    else if measurement_packet.type is **end_loop then**
        break
    update_pending_feedbacks(measurement_packet.p_id)
    $p \leftarrow$: P_r[measurement_packet.p_id]
    if measurement_packet.dscp **is** $0\text{x}17$ and **not** p.in_recovery_queue **then**
        insert_to(recovery_queue, p)
    else if measurement_packet.dscp **is** $0\text{x}21$ **then**
        p.forward_index(measurement_packet)
        p.mark_congested_link($\mathcal{A}$, measurement_packet)
        if not p.in_recovery_queue **then**
            insert_to(recovery_queue, p)
        if aggressive_mode is **enabled then**
            initiate_recovery(p, pending_feedbacks)
    else if measurement_packet.dscp **is** $0\text{x}17$ **then**
        p.mark_completed()
    mark_switches_responsive(measurement_packet)
append_unresponsive_switches($\mathcal{A}$)
return $\mathcal{A}$

The data recovery module needs to keep whether feedback packets of released probe packets have reached the controller or not. To do so, we use the *pending_feedbacks*, which is a set containing unique identifiers of paths. Since there is no feedback packet received initially, it contains identifiers of every path. The recovery module also requires a *recovery_queue*, a priority queue to access the earliest deadline in $O(1)$. The deadline value is the summation of the timestamp a path inserted into the recovery_queue and T_r. Both pending_feedbacks and recovery_queue can be initialized within the data recovery module, but doing so may induce additional delays before starting the main loop. Hence, as a design choice, we opt to initialize them before starting measurements and activating the recovery module. Finally, the controller gives an empty set $\mathcal{A}$ as an argument so that the module can save its conclusions, such as congested links or unresponsive switches.

Before we start our infinite loop, we set a feedback timer, saving the current timestamp to a variable (line 1). The feedback timestamp is used to understand whether enough time has passed to examine *pending_feedbacks* or not. We start our infinite loop and wait for the packet receiver to forward measurement packets. The packet receiver uses *packet_queue* as a communication medium (line 2). Waiting for packets to arrive would be a blocking call and hinder the purpose of the recovery module. Hence, we set a timer, T_{ro} (i.e., resolution time), on the queue to break out of the blocking call and perform recovery actions. When we observe a timeout event, it means we have not received any packets within T_{ro} (line 4). In such an event, we first check if we crossed the T_f (line 5). If that is the case, we traverse over pending feedbacks (lines $6-7$) and check whether they are in the recovery queue or not (line 8). A pending feedbacks means that the switch path p released from did not generate a feedback packet, or the generated packet has not arrived at the module yet. A path being in the recovery queue indicates that it is already noted as incomplete, either received no packets from that path or reported as broken by the data plane. Furthermore, there can only be one path object representing p in the recovery queue to prevent generating more recovery packets than necessary. Doing so requires us to guarantee that the path object is always up to date, and we explain how it is achieved at line 24. When p is in the queue, the module is waiting for either path's deadline to arrive to initiate recovery packets or, better, its completion packet. However, when

we initiate recovery packets, we also remove paths from the recovery queue. Hence, to ensure the recovery module functions correctly, we constantly need to keep track of paths, and checking missing feedbacks plays an important role in doing so. We attempt to initiate recovery from the same switch for R_a times, but when we cross R_a, we need to notify the measurement framework and continue with the next switch in path p (lines 9-11). Notifying the framework means that we do not expect a measurement report from the current index (switch) of path p and mark it as potentially unresponsive. We perform notification over a similar logic of how packet receiver passes reports to the recovery module. There is a case that we might be already at the last switch in path p and cannot increment it any further. We opt to mark such paths as completed in the recovery module so that we will not loop forever if the switch is indeed unresponsive. Regardless of the case, we behave as if everything is as expected and proceed on inserting p into the recovery queue (lines 12-13). First, we increment the number of attempts. The *insert_to* function updates the deadline of the current path since it has not been processed for a while. Then, the function inserts it into the recovery queue. After performing the same actions over every path with pending feedbacks, we update the feedback timestamp (line 14). We conclude the timeout event by process the recovery queue (line 15), which will be explained in Algorithm 4. The monitoring framework also has writing access to the packet queue and can notify the recovery module that all INT probes are collected successfully (line 17). In such notification, the recovery module finishes its infinite loop (line 18).

If we receive a packet from the packet receiver within the time limit, we first update the pending feedbacks (line 19), which is removing the path id from the set. Then we fetch the path object p using the unique identifier (line 20). A received packet can have three types. The first type of the packet is when its DSCP field is 0x17, which means feedback packet that a switch generated to notify the previous hop. In this case, the controller is the previous hop of the first switches of paths. If the packet's type is feedback and is not in the recovery queue (line 21), we call the insertion function and insert p to the queue (line 22). We preemptively insert it into the recovery queue to ensure that there is always a path in the queue so that if something undetectable occurs, we can still initiate recoveries. We cover an example scenario in Section 6.2.3.

If the packet's DSCP field is 0x21 (line 23), we understand that the switch in the data

plane did not receive a feedback packet from the next hop in T_l seconds. In this case, we need to forward the path's latest observed index with the number of switches observed in this measurement packet (line 24). In order to perform this action correctly, we need to identify from which index the arrived packet was released previously. One way to identify the starting index of the measurement packet is by encoding it in the request metadata. However, it would occupy additional space in the packet and may require constant modifications of the data plane pipelines on design changes. Therefore, we opt to give unique identifiers to every request without changing the packet layout by modifying the destination MAC address of the released packets. When we initiate a recovery packet, we generate a unique MAC address and map it within the path p with the latest index observed. Hence, we can identify starting index of the measurement packet and increment the latest observed index correctly. When the calculated index is not greater than the latest observed index, which may occur if multiple recovery packets were deployed, this action has no effect. Then, we update $\mathcal{A}$ with the switch pairs that could not communicate and caused the data plane to detect a failure (line 25). If the path p is not in the recovery queue, we update its deadline and insert it into the queue (lines 26-27). The data recovery module also supports an aggressive mod. If enabled, we do not wait T_r seconds and immediately send the recovery packet for p (lines 28-39). In this function, we generate a unique MAC address and match the latest observed index of p with it so that we can forward indices correctly. Then, we save p to pending feedbacks.

Another packet type that carries information is when DSCP is 0x17 (line 30), which means that the received packet is a complete one. In that case, we mark the path p as completed (line 31). Regardless of the packet type, we mark the switches carried in the measurement packet as responsive since they successfully forwarded the probe (line 32). When the infinite loop finally ends with the signal from the monitoring framework, we extend $\mathcal{A}$ with the unresponsive switches detected during the pending feedback check and return $\mathcal{A}$ (lines 33 and 34).

In Algorithm 4, we have the pseudo-code of processing the recovery queue, which contains potentially incomplete paths. We start with saving the current time (line 1) and start the processing loop until there is none left (line 2) or a break condition occurs. The queue is ordered with respect to the earliest deadline, and by peaking,

Algorithm 4 Processing Recovery Queue

Input: recovery_queue, pending_feedbacks

Output: $\emptyset$

```
1:  current_time ← : time.now()
2:  while not recovery_queue.empty() do
3:      p ← : recovery_queue.peak()
4:      if current_time < p.deadline then
5:          break
6:      recovery_queue.pop_front()
7:      p.in_recovery_queue ← : False
8:      if p.completed then
9:          continue
10:     initiate_recovery(p, pending_feedbacks)
```

we obtain path p without modifying the queue (line 3). If we have not reached p's deadline, there is no need to traverse other paths at all, and we can stop the recovery loop (lines 4-5). Otherwise, we can process the path and pop p from the queue (line 6) and mark it as such (line 7). A path might have been collected and completed while it was in the queue. To not generate unnecessary packets, we check if a path is completed or not (line 8) and proceed to the next one in the queue if that is the case (line 9). If it is not completed, we initiate recovery over path p (line 10). In short, the function generates a unique MAC address and matches the latest observed index of p with it, then puts p in pending feedbacks and forwards the path to the packet sender.

6.2.3 Implementation Limitations

There are several limitations that we observe with the current design and implementation of the recovery module.

The first limitation is caused by the T_{ro}, time resolution that we use to fetch packets from the packet receiver of the controller. The smaller values improve the rate at which we can detect events and improve the reaction time of the module. However, they also induce extra load on the controller, which may degrade the performance of other modules if they run at the same hardware. Therefore, it might be better to decouple modules from the controller and offer the monitoring framework as an application

to the controller. While the monitoring framework and the data recovery module can be bundled together, other resource-demanding modules should be separated.

The other limitation can be best explained over the Figure 6.1. Suppose that switch *s2* actually received a feedback packet from the *s3*. At the same time, *s3* processed the probe packet, but due to high congestion levels, it had to drop the packet in the egress queue. In this scenario, *s2* drops the looping probe packet since it received a feedback packet from the *s3* and deduced that the next hop is responsive. However, in reality, the next hop is congested and could not forward the probe to the *s4*. The data recovery module cannot deduce the exact issue in this scenario. In order to resolve this scenario, we ensure that paths are always in the recovery queue unless they are finished so that we can initiate recovery with the delay of T_r. Another approach that could tackle this issue would be assuming the next hop is always unresponsive in the data plane and forward a copy of the probe packet to the controller. However, this approach would increase the communication overhead on the controller immensely and cease the meaning of deploying INT probe paths. On the other hand, we acknowledge that this approach might be useful for mission-critical network infrastructures. Accordingly, INT probe paths can be used as an event triggering mechanism for such systems in which switches send requested information immediately to the controller when they observe INT packets. We note that the data recovery module would work perfectly fine for such deployment.

Another limitation is that we start the recovery initiation for the paths that the module could not collect feedback packets after $T_f + T_r$ seconds at worst. It is possible to initiate recovery as soon as we detect such paths after T_f seconds. However, we would generate many extra probe packets and induce more load on the system.

CHAPTER 7

RESULTS AND DISCUSSION

In this section, we present our numerical results analyzing our optimization model and heuristic in Section 7.1, and verify some of our results with P4 simulations in Section 7.2.

Throughout the evaluation, we compare GPINT with optimization models, and Pan et al.'s Euler heuristic, available in GitHub [73], and we refer to it as *Euler*. Both our optimization models and GPINT accept k, the suggested maximum number of paths, as an input. Therefore, to analyze the effect of different k values, we use GPINT-2, GPINT-3, and GPINT-5, where they refer to $k = \lfloor \frac{2|V|}{10} \rfloor$, $k = \lfloor \frac{3|V|}{10} \rfloor$, and $k = \lfloor \frac{5|V|}{10} \rfloor$ respectively. In other words, we set q to 0.2, 0.3, and 0.5 respectively. The same translation applies to our optimization models as well.

We consider three comparison metrics corresponding to the requirements we have presented in Section 1.1:

The number of paths. As described in *Requirement 1*, the number of paths each solution generates has a direct correlation to the controller load and the path lengths. Therefore, lower values indicate less controller load and longer paths. However, higher values might improve reliability at the cost of more controller load.

The length deviation. In *Requirement 2*, we state that the controller has to wait for measurements from all INT paths to obtain a holistic view. Consequently, high path length deviations cause delays and decrease the freshness of collected information. Therefore, lower values indicate more balanced paths and fresher reports. On the other hand, larger values may indicate misinformation due to the freshness and unintentionally affect reliability.

The average number of shared switches. In *Requirement 3*, we argue that solutions should minimize redundant information and provide minimal INT packet overheads. Minimizing the number of shared switches decreases repetitive information. This measurement also indicates the extensibility of the path generator to carry different sets of information according to the controller's needs. Therefore, lower values mean less redundant information and indicate a larger pool of information that can be carried. Note that cycles within a path are also considered as *shared* since they overlap with themselves and increase the overhead and redundancy.

7.1 Numerical Results

We divide our numerical evaluation into three parts. In the first part, we analyze where GPINT fits compared to the optimization model. In the second part, we challenge GPINT and Euler with larger topologies and compare the quality of INT paths. Even though Euler aims to cover entire interfaces, these comparisons provide great insight into the drawbacks of cyclic paths, such as scalability and freshness of information. In the last part, we analyze the applicability of GPINT and Euler on data center topologies, Fat-Tree, and Leaf-Spine. We obtain numerical results from a server with 64 GB RAM and 10-core Intel Xeon Silver-4114 2.20 GHz.

7.1.1 Optimality Analysis

In this section, we conduct the optimality analysis of GPINT. Since BSIP is a multi-objective optimization problem, we include different BSIP modules where each optimizes one objective. In our analysis, we denote these modules with a single letter as *BSIP-#-k*, where # refers to the minimization mode of:

- *A*, all three objectives,
- *N*, only the number of generated paths,
- *D*, only the deviation of path lengths,
- *S*, only the shared number of switches.

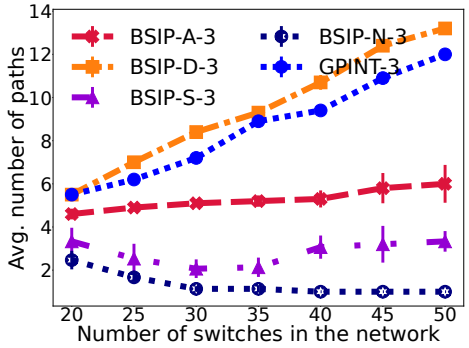
To solve the BSIP problem models, we use the Gurobi Optimizer [74] and wait for models to reach the gap of 0.000% without any time limits. We conduct our optimality analysis with two different graph generators, Erdős-Rényi, which can be considered complete random graphs, and Watts-Strogatz possessing small-world properties. We perform only one set of experiments for complete random graphs, increasing the number of vertices, $|V|$, and fixing the edge attribute. For Watts-Strogatz graphs, we also experiment with increasing $|E|$, the number of edges, with a fixed number of vertices in the topology. To provide better readability, we separate the analysis of GPINT with different k values. Finally, we present results with a 95% confidence interval after 30 repetitions.

7.1.1.1 Complete Random Graphs

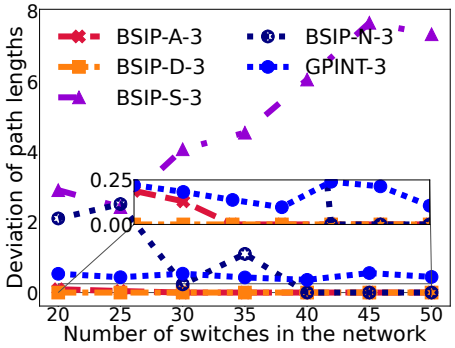
In this section, we compare GPINT with our optimization models on topologies that are randomly generated Erdős-Rényi graphs. In this experiment, we increase $|V|$ from 20 to 50 and fix the average connectivity (or edge probability) property of Erdős-Rényi graphs to 0.15.

In Figure 7.1, we analyze the results for $k = 3$. The first objective we examine is the number of generated paths in Figure 7.1a. The suggested maximum number of paths for $k = 3$ on 50 switches is $\frac{k \times |V|}{10} = 15$. Accordingly, when we only optimize the length deviation, BSIP-D utilizes the maximum number paths followed by GPINT, reducing the number of generated paths by at most three. On the other hand, when we only optimize the number of generated paths, BSIP-N generates the least amount and can find a Hamiltonian path in the graph after 40 switches in the network. BSIP-S, optimizing only the number of shared switches, comes after BSIP-N, generating three to four paths regardless of $|V|$. When we combine all of the objectives, BSIP-A generates five to six paths throughout the experiments. We observe that GPINT does not perform well enough in reducing the number of generated paths.

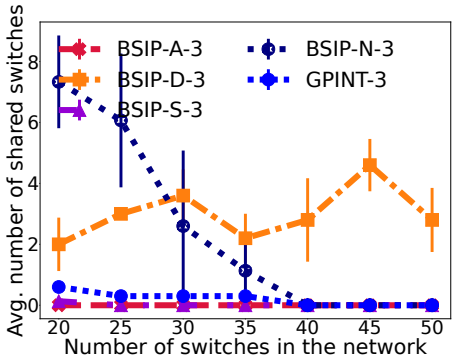
In Figure 7.1b, we depict the standard deviation of path lengths, our second objective. Since BSIP-S does not consider minimizing this objective value, we observe the most deviation in its results. A similar case occurs for BSIP-N as well, which generates non-zero values whenever it cannot find a Hamiltonian path. BSIP-D can find a per-



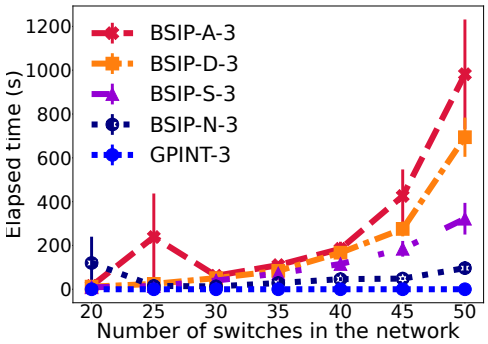
(a) The number of generated paths



(b) The standard deviation of path lengths



(c) The average number of shared switches



(d) Time to generate a solution

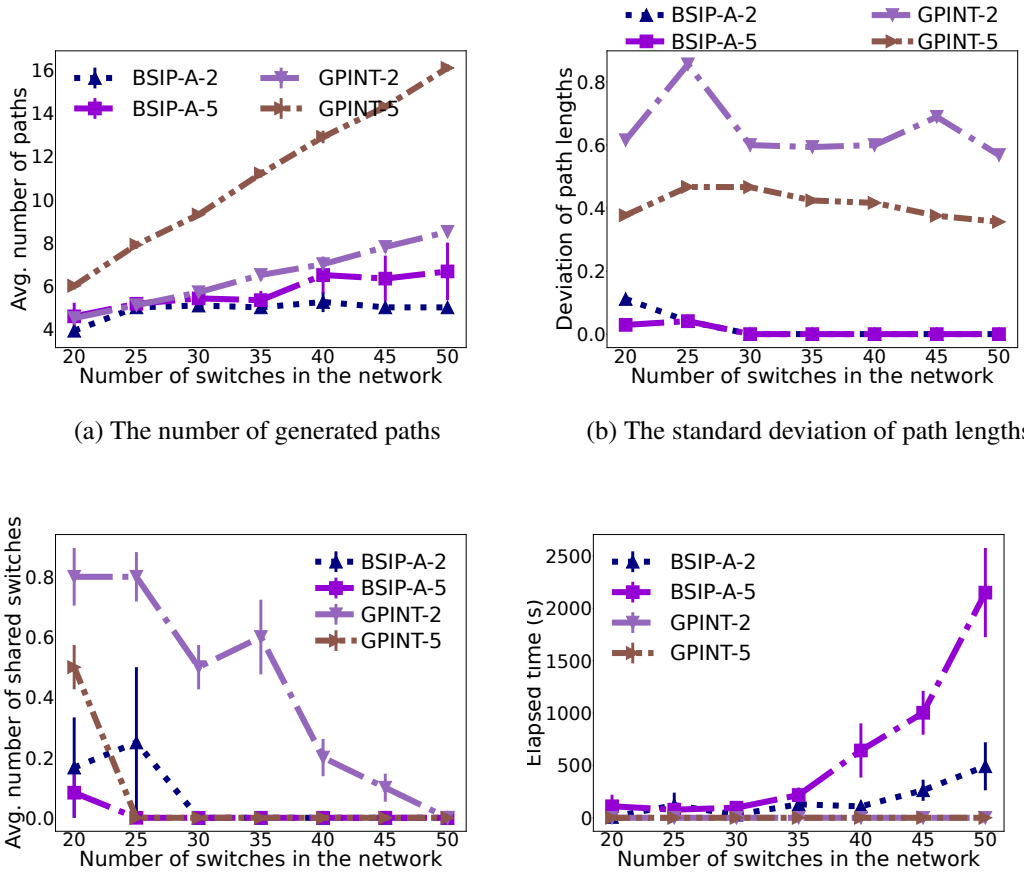
Figure 7.1: Numerical results of experiments where we increase $|V|$ and fix the edge probability to 0.15 on Erdős-Rényi graphs.

fect balance regardless of the topology size. BSIP-A generates slightly off-balanced paths up to 25 switches, but after 30, it can also achieve perfect balance. When we compare GPINT to BSIP-A, we observe that their performances are almost the same up to 25 switches. After 25 switches, the gap fluctuates but remains within the limit of 0.25. Hence, GPINT excels in this objective and delivers almost perfectly balanced paths.

In Figure 7.1c, we examine the results of our last objective, the number of shared switches. Since BSIP-N does not consider minimizing this objective value, it can overuse some of the switches in the topology to achieve better results in its objective, the number of generated paths. Compared to BSIP-N, BSIP-D's paths share relatively few switches. BSIP-A and BSIP-S achieve the best results and generate zero shared

switches. Up to 40 switches, GPINT generates paths that share at the most one switch. After 40, we observe that it can also achieve zero shared switches. Consequently, GPINT performs considerably well in this objective as well.

Lastly, we depict the elapsed times to generate results in Figure 7.1d. Between different objective values, we realize that minimizing the length deviation is the most challenging one for this kind of topologies by examining BSIP-D's results. Combining all three objectives does not decrease the performance too much, but the gap between BSIP-D and BSIP-A enlarges as topology grows. On the other hand, BSIP-N generates solutions the fastest among optimizer models, followed by BSIP-S.



(a) The number of generated paths

(b) The standard deviation of path lengths

(c) The average number of shared switches

(d) Time to generate a solution

Figure 7.2: Numerical results of experiments with different k values where we increase $|V|$ and fix the edge probability to 0.15 on Erdős-Rényi graphs.

In Figure 7.2, we analyze the results of GPINT with different k values, two and five. Similarly, the first objective we examine is the number of generated paths in Figure

7.2a. For the GPINT-5, the suggested maximum number of paths for 50 switches is 25. We observe that it can reduce the number of paths by nine. However, compared to BSIP-A-5, it still generates many paths. For the GPINT-2, this value is 10 for 50 switches and can reduce the number of paths to eight. On the other hand, BSIP-A-2 generates four or five paths regardless of the number of vertices in the topology. From Figure 7.1a and Figure 7.2a, we see that despide different k values, GPINT generates almost twice more number of paths compared to optimal results.

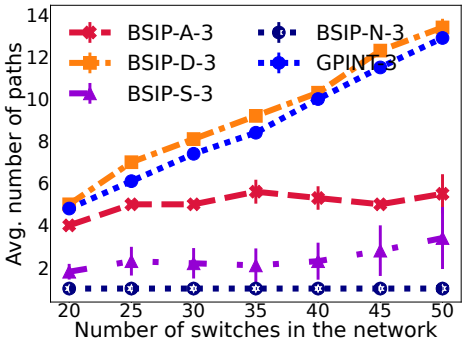
In Figure 7.2b, we analyze the length deviation, our second objective. Even with different k values, we observe that GPINT can generate balanced paths with a deviation of at most 0.9. We examine our last objective, the number of shared switches, in Figure 7.2c. We see that GPINT-2 generates the most joint paths, but the number of shared switches is still less than one and decreases as the number of vertices increase. GPINT-5 comes right after, but after 25 switches, it matches with BSIP-A-5. Similarly, BSIP-A-2 generates zero shared switches after 30 switches. Finally, we analyze elapsed times to generate a solution in Figure 7.2d. If we also consider BSIP-A-3's results in Figure 7.1d, we see that as the k increase, the required time to obtain a solution also increase drastically.

As a result of these experiments, we see that GPINT successfully delivers near-optimal results for each comparison metric except the number of generated paths.

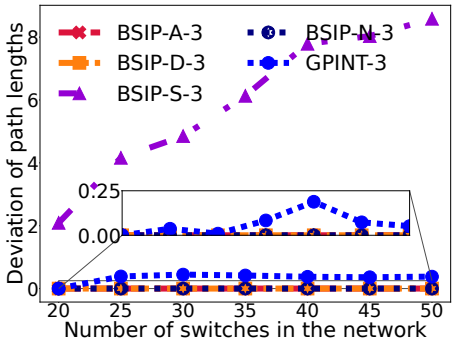
7.1.1.2 Random Graphs with Small-World Properties

In this section, we analyze the optimality of GPINT on topologies generated using the Watts-Strogatz model. In the first set of experiments, we increase $|V|$ from 20 to 50 and fix the number of edges each node has to six.

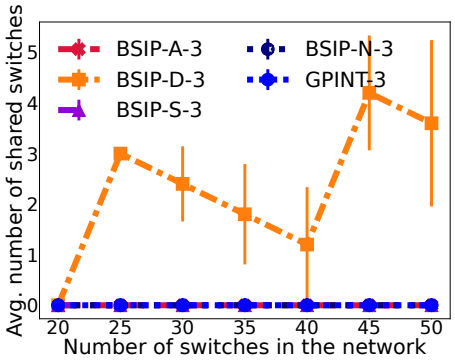
In Figure 7.3, we present the results for $k = 3$. The first objective we analyze is the number of generated paths in Figure 7.3a. When we only optimize the length deviation, the BSIP-D generates the maximum number of paths, followed by our heuristic GPINT, which reduces the generated paths just by two. On the other hand, when we only optimize the number of generated paths, the BSIP-N finds a hamiltonian path in the graph, generating just one path. Even though BSIP-S does not concern the num-



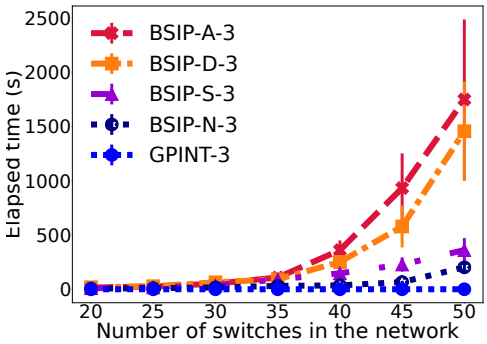
(a) The number of generated paths



(b) The standard deviation of path lengths



(c) The average number of shared switches



(d) Time to generate a solution

Figure 7.3: Numerical results of experiments where we increase $|V|$ and fix number of edges each switch has to six on Watts-Strogatz graphs.

ber of generated paths, it generates relatively few paths, increasing very slightly as the topology grows. When we optimize all three objectives, the BSIP-A generates rather a consistent number of five paths, which translates as a balance between the controller overhead and the time the longest path spends on the network. Compared to the BSIP-A, GPINT generates almost two times more paths which are considerably shorter than BSIP-A's. Hence, under some network conditions, generating smaller paths may be an advantage. However, in terms of optimality, we realize that GPINT performs poorly in reducing the number of generated paths. When we compare path generators' performances on the Watts-Strogatz graph to Erdős-Rényi graphs (Figure 7.1a), we see that both GPINT and BSIP-D generate one more path on Watts-Strogatz graphs. On the other hand, BSIP-A and BSIP-S produce one fewer path than their val-

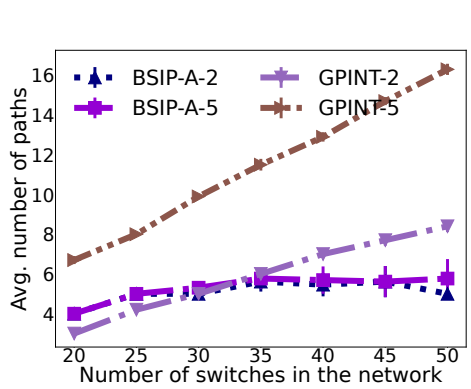
ues on Erdős-Rényi graphs. Similarly, BSIP-N can find a Hamiltonian path starting from 20 switches on Watts-Strogatz graphs, whereas, for Erdős-Rényi, it can only find after 30 nodes in the topology.

We depict our other objective's results, the standard deviation of path lengths, in Figure 7.3b. Since the BSIP-S only concerns the number of shared switches, its length deviation is quite significant. Furthermore, the BSIP-N generates only one path, whose deviation is zero as a result. We observe that GPINT generates almost perfectly balanced paths, only off by at most 0.25. Moreover, GPINT can find a slightly better balance on Watts-Strogatz graphs than Erdős-Rényi (Figure 7.1b). The BSIP-D and BSIP-A generate perfectly balanced path lengths, whose deviation is zero for all experiment values. As a result, we understand that GPINT performs quite well in this objective.

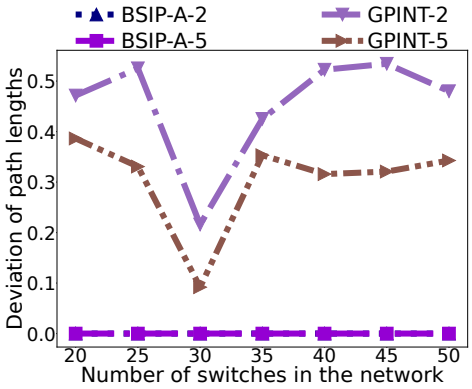
In Figure 7.3c, we present the results of our last objective function, the number of shared switches. The BSIP-D only optimizes the length deviation. Hence, to minimize its objective, it can generate paths that intersect a lot. Even though the number of these intersections seems to be random and dependent on the layout of topology, we see that it is at worst five. Since BSIP-N only generates one path, its results are zero. BSIP-A, BSIP-S, and GPINT also generate zero intersections. Consequently, GPINT excels in its performance in this objective as well.

Lastly, we analyze the time required to generate solutions in Figure 7.3d. By just looking at the BSIP-A and BSIP-D, we can understand that generating balanced paths is a challenging task of the problem as they struggle the most. On the other hand, minimizing the number of paths, hence finding the Hamiltonian path in these topologies, is the least challenging task followed by BSIP-S, generating any number of disjoint paths.

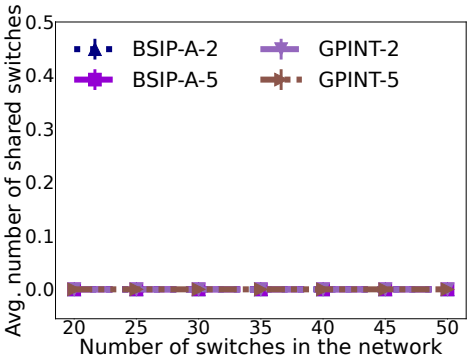
In Figure 7.4, we conduct an analysis of the same setting but with different k values for GPINT and BSIP. We analyze the number of generated paths in Figure 7.4a. The first thing we note here is that up to 30 nodes, GPINT generates fewer paths than BSIP-A-2. Afterward, both BSIP-A-2 and BSIP-A-5 generate the least number of paths, followed by GPINT-2 and GPINT-5. Compared to completely random graphs, we observe that GPINT-2 can generate slightly fewer paths. Furthermore, BSIP-A-5



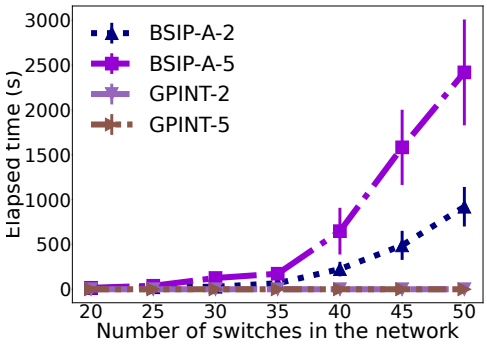
(a) The number of generated paths



(b) The standard deviation of path lengths



(c) The average number of shared switches



(d) Time to generate a solution

Figure 7.4: Numerical results of experiments of different k values where we increase $|V|$ and fix number of edges each switch has to six on Watts-Strogatz graphs.

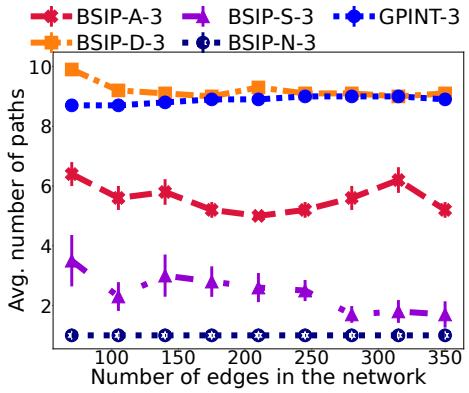
is more stable and can reduce the number of paths to the BSIP-A-2's results.

We have the results for our second objective, the length deviation, in Figure 7.4b. We see that BSIP-A-2 utilizes additional paths that it generated compared to GPINT-2 to minimize the length deviation. Meanwhile, we observe that GPINT-5 finds a slightly better balance than GPINT-2. However, the gap between them seldom exceeds 0.1. One thing to note here, under 30 nodes, BSIP-A performs better on Watts-Strogatz compared to Erdős-Rényi graphs. Furthermore, GPINT also delivers better balance on this graph model even though its performance fluctuates insignificantly.

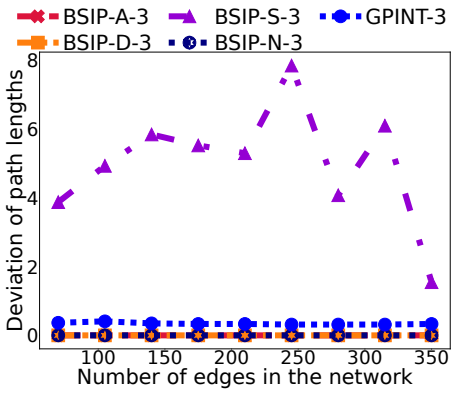
In Figure 7.4c, we have results of our third objective, the number of shared switches. We observe that in Watts-Strogatz graphs, in accordance with Figure 7.3c, both GPINT

and BSIP-A can deliver no joint paths. Whereas in Erdős-Rényi graphs, we observe a limited number of shared switches for BSIP-A-2 and GPINT-2.

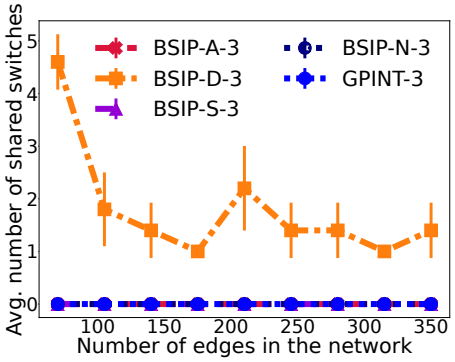
Finally, we analyze the elapsed time to generate a solution in Figure 7.4d. We observe that an increase in k dramatically changes the required time to obtain a solution for BSIP-A. Compared to the Erdős-Rényi graphs (Figure 7.2d), BSIP-A takes up to 500 more seconds to generate a solution as the number of vertices in the topology increase. One partial reason for this behavior is that in Erdős-Rényi graphs, edges are less structured (i.e., some vertices having twice or third times more edges than others), limiting the options of the optimizers automatically. Whereas in Watts-Strogatz graphs, all vertices have the same number of edges. Hence, the optimizers need to explore more options.



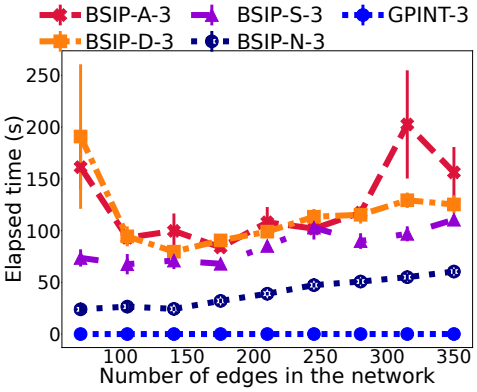
(a) The number of generated paths



(b) The standard deviation of path lengths



(c) The average number of shared switches



(d) Time to generate a solution

Figure 7.5: Numerical results of experiments where we increase $|E|$ and set $|V| = 35$ on Watts-Strogatz graphs.

In the second set of experiments, we increase the number of edges each node has from four to 20, which corresponds to an increase in $|E|$ as we set $|V| = 35$.

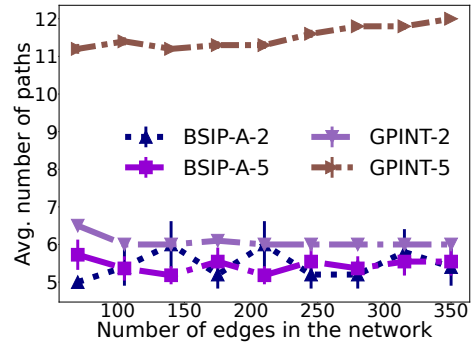
In Figure 7.5, we present the results for $k = 3$. As always, the first objective we analyze is the number of generated paths, depicted in Figure 7.5a. BSIP-A, BSIP-D, and GPINT generate a consistent number of paths throughout the experiment since they do not depend on the number of edges in the topology. Among them, BSIP-D and GPINT generate the most paths, just off by one path from the suggested maximum number of paths. As we observed earlier, GPINT performs poorly in reducing the number of generated paths compared to BSIP-A, and it continues in this experiment as well. BSIP-N finds a Hamiltonian path in the graph. Hence its results are one, followed by BSIP-S, which generates a decreasing number of paths as the connectivity increases.

In Figure 7.5b, we depict standard deviation of path lengths. Since BSIP-S's concern is to minimize the number of shared switches, its results are rather hard to depict as they seem to be random. BSIP-N generates only one path. Therefore, its results are zero. While connectivity does not change the number of generated paths, we see that it helps GPINT to close the 0.25 gap from the optimal value.

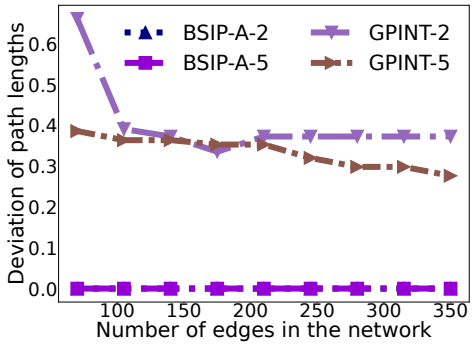
We analyze our last objective, the number of shared switches, in Figure 7.5c. The first thing we observe is that as the connectivity increases, BSIP-D generates more disjoint paths. On the other hand, other optimization models and GPINT generate no joint paths.

Lastly, in Figure 7.5d, we evaluate the time required to generate solutions. We observe that the increasing connectivity does not affect BSIP models significantly compared to the increasing number of vertices in Figure 7.3d. Even though all of the elapsed times are close, it is interesting to see that BSIP-S and BSIP-D almost take the same amount of time. This indicates that as the connectivity increases, it becomes easier to balance the paths.

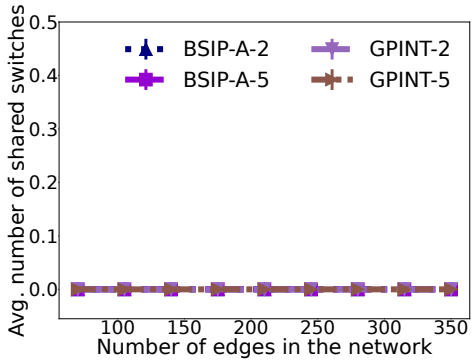
In our final analysis for this section, we compare different GPINT and BSIP-A with different k values as the number of edges in the network increase in Figure 7.6. We analyze the number of generated paths in Figure 7.6a. We observe that BSIP-A-2



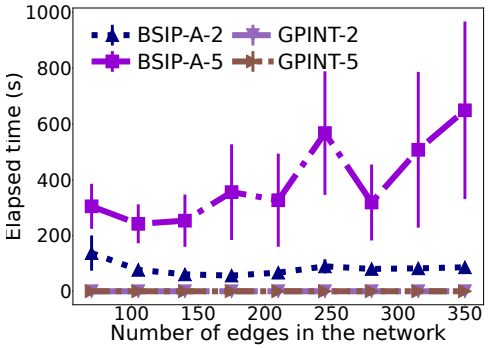
(a) The number of generated paths



(b) The standard deviation of path lengths



(c) The average number of shared switches



(d) Time to generate a solution

Figure 7.6: Numerical results of experiments with different k values where we increase $|E|$ and set $|V| = 35$ on Watts-Strogatz graphs.

and GPINT-2 generate a similar number of paths with a gap of at most one path. However, the gap between BSIP-A-5 and GPINT-5 is vast, where GPINT-5 generates almost twice more paths than BSIP-A-5. Even though the gap is large, we need to highlight that GPINT-5's paths are considerably shorter, which might be a desirable configuration option depending on the network conditions. Furthermore, a similar degree of configuration can also be added to the optimization models.

In Figure 7.6b, we analyze our second objective, the length deviation. Despite the similarity between the generated number of paths for GPINT-2 and BSIP-A-2, we observe that BSIP-A-2 can generate perfectly balanced paths along with BSIP-A-5. For GPINT-2, we notice that it generates a relatively constant level of balance after 100 edges. On the other hand, as the number of edges in the topology increase,

GPINT-5 improves the balance by utilizing the additional paths it utilizes in Figure 7.6a.

In Figure 7.6c, we compare our third objective, the number of shared switches. We observe that even with different k values, GPINT and BSIP-A can generate paths that do not overlap. Hence, they can successfully minimize the redundant information, which corresponds to the maximization of the available information pool.

Finally, we analyze the elapsed time to generate a solution in Figure 7.6d. As we observed in the analysis of $k = 3$ in Figure 7.5d, increasing the number of edges has a minimal effect on BSIP-A. Hence, their generation times are relatively stable and constant except for BSIP-A-5, which we observe some outliers.

Based on these analyses, GPINT can generate balanced and disjoint paths at a near-optimal level for both completely random graphs and Watts-Strogatz graphs. However, it performs poorly in the optimization of the number of generated paths. Hence, the selection of k plays an important role for GPINT to minimize the number of generated paths. We leave finding the optimal k value given the network conditions as future work.

7.1.2 Scalability

In this section, we focus on the scalability aspect of GPINT and compare it with the Euler method [28] and SNMP. The Euler method's goal is to traverse every edge, unlike GPINT's, to traverse every node. Furthermore, the Euler method's performance depends on the number of odd degree vertices. In this work, we do not control such vertices. Hence, our results may differ from their original work. We perform our analysis on two different graph topologies, completely random and Watts-Strogatz, similar to the optimality analysis.

7.1.2.1 Complete Random Graphs

We conduct our analysis on two different topologies for completely random graphs. First, we test the increasing number of vertices on Erdős-Rényi topologies. Then, we

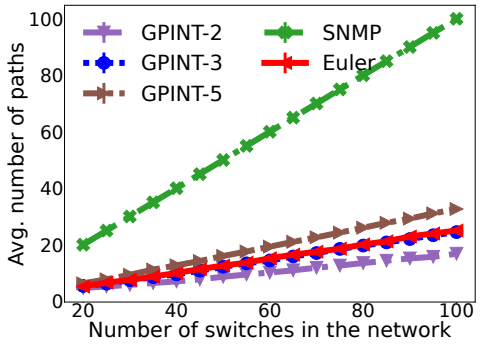
fix the number of vertices in the topology and increase the number of edges. However, we cannot use Erdős-Rényi topologies on the latter since they are not suitable for generating a consistent number of edges in a graph that can be depicted and analyzed. Instead, we use Networkx's [75] dense topology generator, which accepts a number of vertices and edges as input and returns a random graph out of a large graph pool.

In Figure 7.7, we depict results of an increasing number of vertices on Erdős-Rényi graphs with a fixed edge probability of 0.15. We analyze the number of generated paths in Figure 7.7a. The SNMP probes every switch and generates an equal number of paths to $|V|$. We observe that the Euler method generates the same number of paths to GPINT-3. Meanwhile, GPINT-2 and GPINT-5 generate the least and most paths after SNMP, respectively. Note here that even though GPINT performs poorly in reducing the number of generated paths, we see that GPINT-5 can reduce the number of paths by almost half, considering the suggested maximum number of paths being 50 for 100 nodes. Whereas GPINT-2 can only reduce it by five.

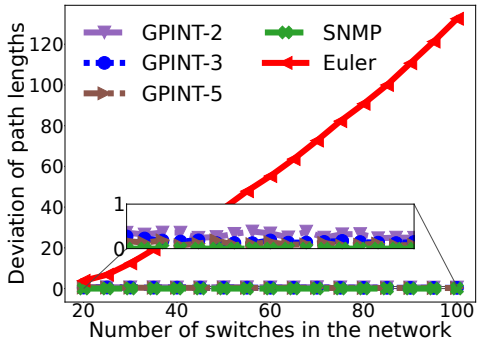
In Figure 7.7b, we analyze our second objective value, the length deviation. Even with an increasing number of vertices in the topology, we see that GPINT can generate almost perfectly balanced paths all the time. Since SNMP's path lengths are always one, its length deviation is automatically zero. In the case of Euler, we see that they fail to balance the path lengths. The high deviation indicates some extraordinarily long and small paths, given that it covers all edges with a considerably small number of paths. Deploying such paths on the network would hinder obtaining a timely holistic view, as we explore in our simulation results.

In Figure 7.7c, we measure the number of shared switches, our third objective. We see that GPINT with different k values can generate disjoint paths with at most one shared switch. Furthermore, GPINT generates no shared switches after 80 vertices. For the Euler method, they need to traverse every node multiple times so that they can cover every edge. However, by doing so, they generate too much redundant information. Hence, the Euler method is not suitable for carrying a large pool of information and is only limited to tracking hop by hop latency.

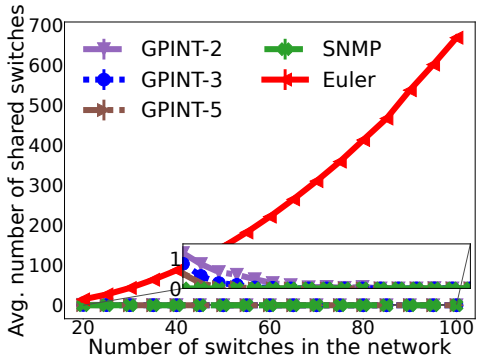
We measure the required time to generate a solution for Erdős-Rényi graphs in Figure 7.7d. We observe that the required time to generate a solution follows an exponential



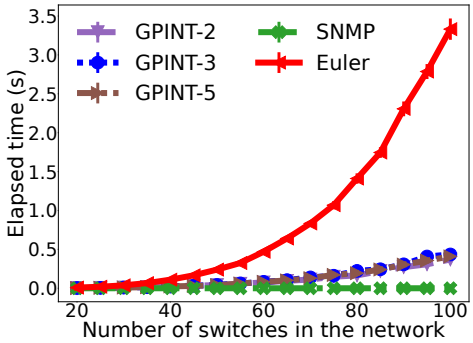
(a) The number of generated paths



(b) The standard deviation of path lengths



(c) The average number of shared switches

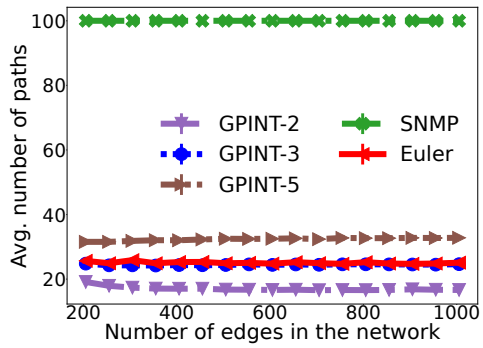


(d) Time to generate a solution

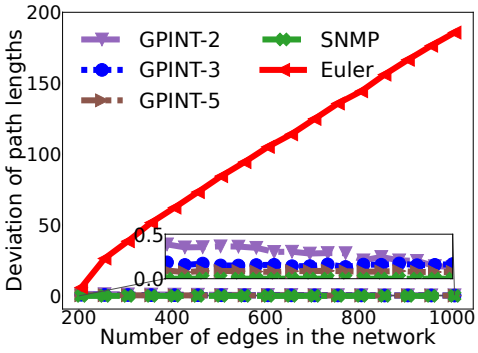
Figure 7.7: Numerical results of experiments where we increase $|V|$ and set edge probability to 0.15 on Erdős-Rényi graphs.

increase for the Euler method. As the number of vertices increase, we see that GPINT also starts to take more time to generate a solution. However, it can generate a solution under 0.5 seconds even at 100 switches.

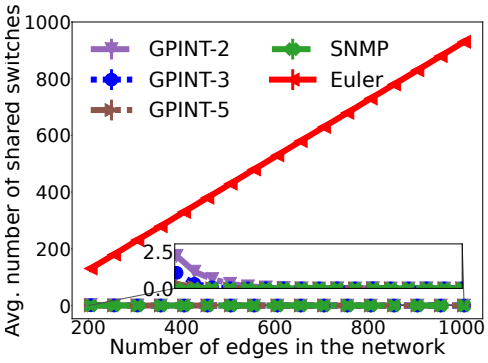
In Figure 7.8, we analyze how the increasing number of edges (connectivity) with a fixed number of 100 vertices affects path generators. We use Networkx's [75] dense random topology generator for this experiment. In Figure 7.8a, we measure the number of generated paths. GPINT does not depend on the number of edges but the number of vertices. Hence, we observe a constant number of paths for GPINT with different k values, except GPINT-2. As the number of edges increases from 200 to 400, we see that GPINT-2 can utilize additional links to decrease the number of paths. Once every node has a certain number of edges, GPINT-2 also generates a constant



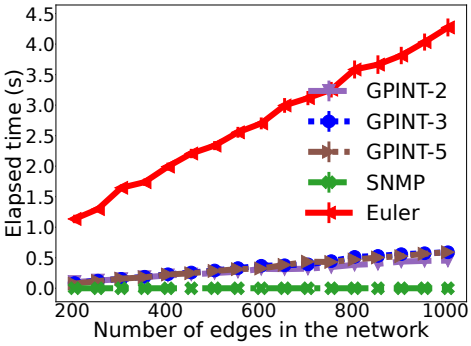
(a) The number of generated paths



(b) The standard deviation of path lengths



(c) The average number of shared switches



(d) Time to generate a solution

Figure 7.8: Numerical results of experiments where we increase $|E|$ and set $|V| = 100$ on random graphs.

number of paths. When we analyze how Euler performs, we see that they also generate a constant number of paths, despite covering every edge. Doing so degrades other objective functions as we examine in Figure 7.8b and Figure 7.8c.

In Figure 7.8b, we measure the length deviation of paths. We observe that GPINT can deliver balanced paths with any k, and as the number of edges increases, GPINT generates slightly more balanced paths. On the other hand, the Euler method fails to generate balanced paths, which is similar to our observations for Figure 7.7b. Additionally, we realize that Euler performs worse as in this experiment compared to an increasing number of vertices. That is because Euler generates a fixed number of paths despite an increasing number of edges. Of course, this could be avoided if the generated paths were divided according to the mean length of the paths. However, in

such a case, there would be a dramatic increase in the number of generated paths.

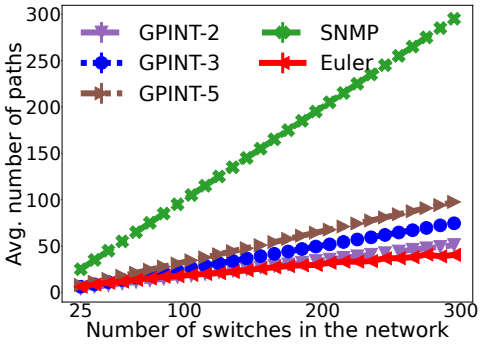
We analyze the number of shared nodes in Figure 7.8c. When there are only 200 edges, both GPINT-3 and GPINT-5 generates paths that share nodes to cover the topology. However, as the number of edges increases in the network, we see that GPINT generates no shared switches for all k values. For the Euler method, however, the number of shared switches increases significantly. Similar to the analysis of Figure 7.7c, they carry too much redundant information, which reduces the applicability of Euler in a real-life scenario.

Finally, we analyze the required time to generate a solution in Figure 7.8d. We observe that increase in the number of edges does not affect GPINT too much, and for different k values, GPINT takes almost the same amount of time. For the Euler method, we observe a constant but steep increase in the time required to generate a solution.

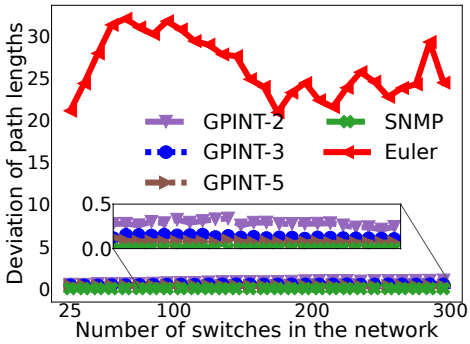
From these two sets of experiments, we see that GPINT is scalable for completely random graphs. Furthermore, GPINT can sustain near-optimal performances in both length deviation and shared number of switches for these graphs. On the other hand, we see that Euler fails to generate balanced paths and extensively overuses switches, which increases the redundant information.

7.1.2.2 Random Graphs with Small-World Properties

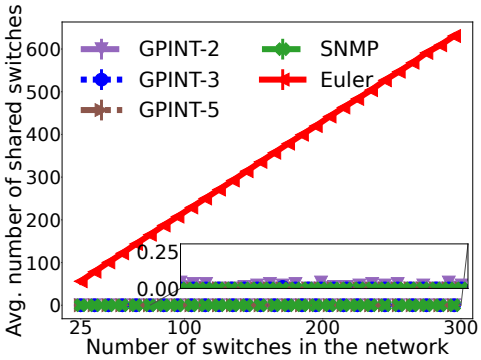
In this section, we conduct scalability analysis and use the Watts-Strogatz model to generate random graphs with small-world properties that are more likely to resemble real-life topologies than complete random graphs. In the first half, we increase the number of switches, $|V|$, in the network from 25 to 300 by five and fix the number of edges between any two switches to six. In the second half, we increment the number of edges (links) between two switches from four to 20 by two as we fix $|V|$ to 100 switches. We present results with a 95% confidence interval after 100 repetitions. Furthermore, please note that the Euler method is sensitive to the number of odd degree switches a topology has. In this work, we do not control or adjust such switches, and hence, our results may differ from Pan et al.'s paper [28].



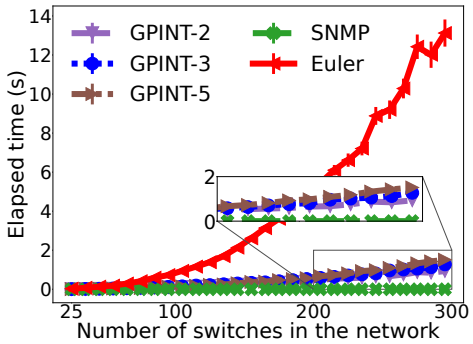
(a) The average number of INT paths



(b) The deviation of INT path lengths



(c) The average number of shared switches



(d) Time to generate a solution

Figure 7.9: Numerical results of experiments where we increase $|V|$ and fix the number of edges each node has to *six*.

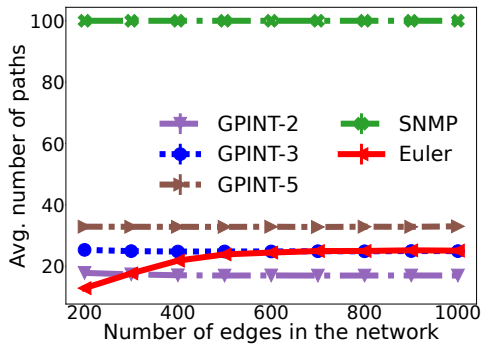
For the first set of experiments, Figure 7.9a depicts the average number of paths each method generates. All methods follow a similar trend where the average increases with the number of switches. We observe that after a 100 number of switches, the gap between GPINT and the Euler slightly opens where the Euler generates the least amount of paths while GPINT-2, GPINT-3 and GPINT-5 following accordingly. Since the SNMP probes every switch, the number of paths equals the number of switches in the network. One key takeaway from this figure is that the Euler manages to cover every edge in the network with considerably few paths. The consequences of this behaviour is evident in both Figure 7.9b and Figure 7.9c.

In Figure 7.9b, we measure the path length deviation, which directly correlates freshness of the information and how fast the controller can collect INT reports. Despite

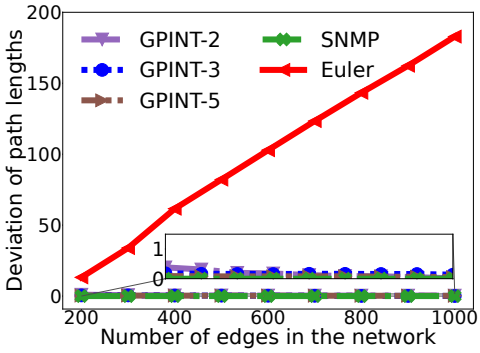
generating a lower number of paths, the Euler fails to generate paths with balanced lengths even though it is better than random graphs (Figure 7.7b). Imbalanced path lengths degrade the information's freshness and can delay the time to collect reports. Such delays can reduce the controller's reaction time in case of failure or misinform the controller to take suboptimal decisions. On the other hand, GPINT-5 finds the best balance after the SNMP and GPINT-3 with GPINT-2 follow right behind, ensuring minimal degradation of the freshness and delay. In Figure 7.9c, we analyze how many times a switch is being traversed by INT paths, which might increase the redundant information and the overhead. Since Euler's goal is to traverse every edge, they cannot avoid traversing a switch multiple times. However, if the Euler were to be employed to fetch other information that can be obtained at one visit, then there would be a vast amount of redundancy that may exhaust available resources depending on the size of the information. On the other hand, we observe that GPINT successfully minimizes the redundant information and matches with SNMP. In other words, GPINT can be employed to carry any information with minimal redundancy.

We evaluate the same metrics for the second set of experiments when the number of edges (links or connectivity) in the network increases as we fix $|V|$ to 100. In Figure 7.10a, we observe that GPINT generates a constant number of paths independent from the number of edges in the network. That is because GPINT's concern is to cover switches rather than the edges. Furthermore, GPINT-2 generates the least number of paths followed by the Euler and GPINT-3, GPINT-5, and the SNMP as the number of links increases. Similar to the analysis of Figure 7.9a, we note that the Euler generates a considerably few paths despite the increase in the number of edges. However, as expected from the previous analysis, this behavior affects other metrics dramatically. In Figure 7.10b, the Euler's unbalanced path generation becomes apparent as there is a linear increase in the standard deviation as the number of links increases. On the other hand, GPINT generates almost perfectly balanced paths. In the number of shared switches, we note a similar pattern in Figure 7.10c. We would like to emphasize that Euler's behavior is because they cover every edge in the network. Therefore, they cannot avoid over-visiting the same switches. Consequently, these results indicate that the Euler approach is limited to fetch only port statistics.

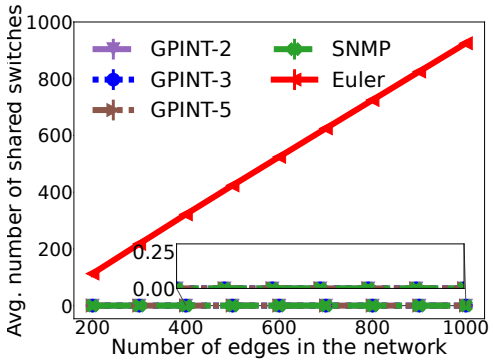
In Figure 7.9d and 7.10d, we examine the required time to generate solutions for



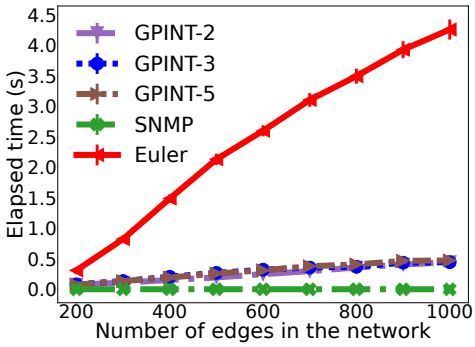
(a) The average number of INT paths



(b) The deviation of INT path lengths



(c) The average number of shared switches



(d) Time to generate a solution

Figure 7.10: Numerical results of experiments where we increase $|E|$ and set $|V| = 100$.

analyzing scalability. This metric is critical because it is likely to frequently run these path generators since topologies evolve in time with the addition of new switches or extraction of some due to failures. Therefore, it is desirable to have a fast path generator.

In Figure 7.9d, we depict the elapsed time to generate a solution for the first set of experiments where we increase the number of switches in the network. We realize that up to 100 switches, all methods can generate solutions within a second. However, after 100, we observe an exponential growth for the Euler method. As the number of switches in the network reaches 300, the Euler method can take 15 seconds to generate a solution. On the other hand, GPINT can generate a solution under two seconds even when there are 300 switches in the topology. Between the k variations,

we note that GPINT-2 is the fastest and is followed by GPINT-3 and GPINT-5 with a small difference. We observe similar results as the number of edges increases in the network, in Figure 7.10d. The main difference is that the Euler method follows a more linear direction compared to Figure 7.9d. However, please note that in this figure, we fixed the number of switches to 100. Consequently, in much larger topologies (with a high number of switches and high connectivity), the Euler may require more time to generate solutions. Whereas GPINT can generate paths under one second even when the topology is considerably connected.

These two scalability experiments show that GPINT can preserve the quality of paths, balanced, and minimized redundancy, with the cost of additional paths compared to the Euler. Even though the Euler method often generates fewer paths, they generate highly unbalanced paths with the reuse of the same switches to cover every edge. There are two implications of this behavior. First, when one of the extremely long paths gets lost due to failures, which is a lot likely than shorter ones, the controller loses too much information to pinpoint the exact issue. Secondly, they carry too much redundant information, making Euler limited to fetch only port information. For the GPINT, the paths are balanced, and when a packet loss occurs, the controller can react faster and more accurately due to a small portion of information loss. Furthermore, GPINT minimizes redundancy and can be extended to carry any information. Between different k values, we note that GPINT-2 performs slightly better than others due to the fewer paths. However, according to the network statistics, such as link congestions, it might be more desirable to generate more and smaller paths to improve reliability. In such cases, GPINT-5 can be preferred. Therefore, it depends on the network conditions to select the k value, but, in any case, GPINT can deliver balanced paths with minimal redundancy.

7.1.3 Data center Experiments

In this section, we compare GPINT with the Euler method on data center topologies, FatTree, and LeafSpine. For topology generation, we use codes available in Pan et al.'s GitHub repository [73]. We only provide results of GPINT-3 in Table 7.1 since it fills the gap between GPINT-2 and GPINT-5. Note that the topology generation

is deterministic for data center networks, and thus experiments are conducted in the same topologies.

Table 7.1: The results of data center experiments.

		Number of Paths		**Length Deviation**		**Shared Switches**	
	$\|V\|$	Euler	GPINT-3	Euler	GPINT-3	Euler	GPINT-3
FatTree	25	**5**	**5**	24.00	**0.00**	50	**0**
	30	**1**	6	–	**0.00**	67	**0**
	35	**7**	**7**	39.19	**0.00**	98	**0**
	40	**1**	8	–	**0.00**	121	**0**
	45	**9**	**9**	56.57	**0.00**	162	**0**
	50	**1**	10	–	**0.00**	191	**0**
LeafSpine	27	9	**7**	45.25	**0.99**	144	**2**
	30	**1**	8	–	0.93	171	**3**
	33	11	**8**	63.25	**0.87**	220	**3**
	36	**1**	9	–	0.94	253	**3**
	39	13	**10**	83.14	**0.98**	312	**3**
	42	**1**	11	–	0.94	351	**4**

When we compare the number of generated paths, we observe that Euler generates either one or an equal number of paths to GPINT for FatTree. On the other hand, for LeafSpine, we see that GPINT provides slightly fewer paths whenever Euler does not generate one path. Covering every edge with only one path indicates that it is exceptionally long. Deploying such a path to the data plane would delay data collection, limit the carriable information, and increase vulnerability to packet losses. As a result, the monitoring system might perform less efficiently or may not perform at all due to packet losses. Whenever Euler does not generate one path, it produces unbalanced paths with high length deviation in both of the topologies, which hinders concurrent data collection. When we examine the number of shared switches, we observe that Euler overuses switches to cover every link. However, this maximizes the redundancy and consequently minimizes the available set of information that can be

carried. On the other hand, GPINT produces balanced paths with minimized redundancy and length deviation. Interestingly, GPINT's results of the length deviation and shared number of switches on the FatTree topology is zero for all experiment values. That is because the FatTree topologies generated by Pan et al.'s source code consist of pods of five switches, and GPINT-3 can capitalize on this topology attribute as follows. In the initial partition phase, it starts forming paths from the lowest degree nodes, which are the leaves. Then, we grow paths until we reach the length limitation, which is three in this case. This creates isolated nodes in the pods, one of them is a leaf node, and the other is the root of the pod. When we arrive at these isolated nodes, we explore their neighbors to seek unvisited nodes. However, what we realize is that their neighbors are endpoints, and they halted due to the length limitation. Since we relax this limitation for these cases, we can cover these isolated paths in the initial partition phase and generate perfectly balanced paths with no shared switches. Even though we observe some shared switches on LeafSpine topologies, it is because these topologies generated by Euler's source code are fully connected bipartite graphs that contain $\frac{|V|}{3}$ spines and $\frac{2|V|}{3}$ leaves. That is, covering such topology with no shared switches would require $\frac{|V|}{2} - 1$ spines and $\frac{|V|}{2} + 1$ leaves. These results show that GPINT can be utilized on data center networks and can generate balanced paths with minimal redundant information, making GPINT adjustable to carry any information and a perfectly suitable tool for an efficient monitoring system.

7.2 Simulation Results

In this section, we present our simulation results in which we bundled our system design, proposed path generator heuristic, and data recovery for INT-based measurements. As a basis for the simulation environment, we use ETH Zurich's P4 repository [76], as it introduces various improvements for the original P4 simulation environment. As underlying hardware, we have AMD Ryzen 5 3600 running at 4.00GHz and 32GB RAM.

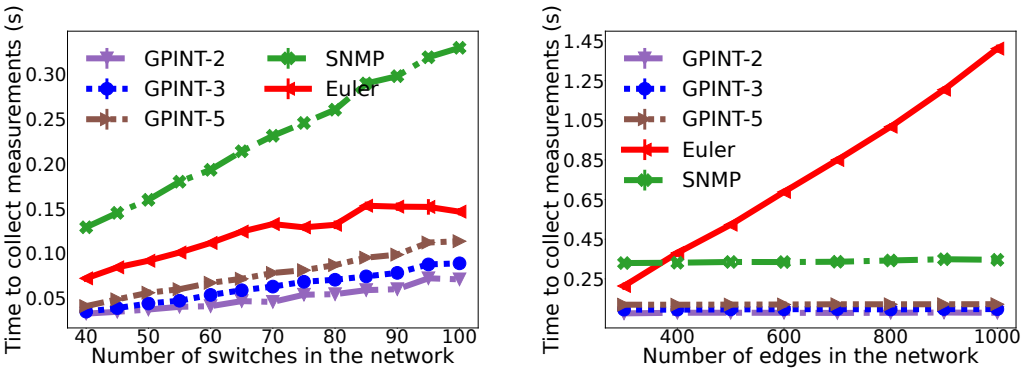
7.2.1 Simulation Setup and Methodology

Throughout the simulations, we use Watts-Strogatz random graphs to show the effects of our claims made in the previous section. In the measurement procedure, we first generate random topology and assign hosts to several switches. All switches and links we deploy in the simulation share the same characteristics. The switches have a queue size of 1000 packets to simulate higher congestion levels within the hardware limitations. The links have unlimited bandwidth, and unless otherwise stated, the links do not have any propagation delays and packet losses.

After we deploy switches and establish links, we start the controller, whose first job is to generate forwarding rules for background traffic based on the shortest distance between hosts. At the same time, we start host processes and assign them roles such as traffic-source and traffic-sink. Once the controller generates forwarding rules, we signal the traffic-source nodes to generate and send UDP packets, which carry 100bytes of payload, to traffic-sinks. The source nodes select sinks randomly for each UDP packet to affect a larger area in the network. Each source node sends packets at the rate of $\frac{\#\text{pps}}{\#\text{source nodes}}$. Meanwhile, the controller starts its measurement loop, in which the controller first generates INT paths running the path generator we provide at the start. Afterward, the controller produces probe packets which include the Request Meta and SR headers necessary to guide packets through the network. Then, the controller uses a pool of 12 processes to release probe packets to the network and measures the elapsed time to collect them. If the INT recovery mode is enabled, it also keeps counting the additionally generated INT packets to achieve seamless recovery. Once every probe packet reaches the controller, it saves the measurements and goes through the measurement loop for 10 times on each topology so that we can provide consistent reports. We refer to these measurement loops as attempts to obtain a holistic view of the network. We repeat these procedures 20 times on randomly generated topologies with different attributes, such as the number of vertices or edges. We provide results with a 95% confidence interval for every measurement metric. Throughout the simulations, we only request rule-hit counters for only one rule unless otherwise is stated.

7.2.2 Without Background Traffic

In this subsection, we analyze the performance of path generators when there is no background traffic. Our performance metric is how fast the controller can collect the measurements after releasing the probe paths.



(a) As $|V|$ increases.

(b) As $|E|$ increases, $|V| = 100$.

Figure 7.11: Elapsed times to gather INT reports with no background traffic.

In Figure 7.11a, we depict the elapsed time to collect measurements as we increase the number of vertices in the network from 20 to 100 while fixing the number of edges each vertex has to six. The first thing we notice is that the SNMP lacks behind due to scalability issues, given that the controller can only send 12 packets concurrently. Despite covering all of the edges in the network, the controller can collect measurements under 0.15 seconds with the Euler method. GPINT, on the other hand, can deliver measurements under 0.10 seconds with every variation. Even though GPINT-2 generates slightly less balanced paths (Figure 7.9b), it achieves the best performance due to fewer generated paths (Figure 7.9a). GPINT-3 finds the right balance between the number of generated paths and the length deviation of them. Despite generating the highest number of paths other than SNMP, the GPINT-5 follows right behind as it finds compensation by providing the best balance among path generators.

Figure 7.11b shows the elapsed time to collect measurements as the number of edges in the topology increases with a fixed 100 number of switches in the network. We notice that the Euler method follows a linear increase and can deliver reports under one second up to 700 edges in the topology. However, once we cross the 700 edges,

Euler exceeds one second and can take 1.45 seconds when there are 1000 edges in the topology. Since the Euler method does not generate balanced paths (Figure 7.10b), shorter paths arrive earlier but wait for the longest paths. As a consequence, the controller delays delivering the holistic view. For the other methods, their performances do not depend on the number of edges a topology has but rather the switches. Hence, we observe constant delivery times, which are the same as the results of Figure 7.11a on 100 switches.

These results depict the baseline performance that each probe generator can provide. The SNMP falls short due to scalability issues as it probes every switch. Even though Euler generates few paths, the generated unbalanced paths hinder obtaining a complete and fresh view which is evident when the number of edges in the network is high. On the contrary, the GPINT delivers reports the fastest as it compensates for the number of generated paths with the balance of path lengths. In the following section, we introduce background traffic and depict how the path generators perform under additional queuing delays, which may even cause packet drops.

7.2.3 With Background Traffic

The solutions that generate INT probe packets to collect measurements have to work under a traffic load that can challenge the processing unit of switches. In such conditions, probe packets will get affected by queuing delay in addition to the processing delay. Even further, they might get dropped under high load if the queue capacity gets exceeded, assuming uniform priority to not disturb commercial traffic under high telemetry frequencies. The required load to exceed queue capacity changes from one simulation hardware to another. However, once such congestion levels are reached, the observed results should be similar. In this subsection, we provide our observation of path generators under such conditions.

In Figure 7.12, we depict the percentage of background traffic loss under different loads. Based on the losses, we categorize 10000pps, 12000pps, and 14000pps as low, medium, and high loads, respectively. Please note that these values might vary from setup to setup. In our setup, we reach 100% CPU utilization on every core when we send a total of 14000 packets per second from traffic sources. For 12000pps, we

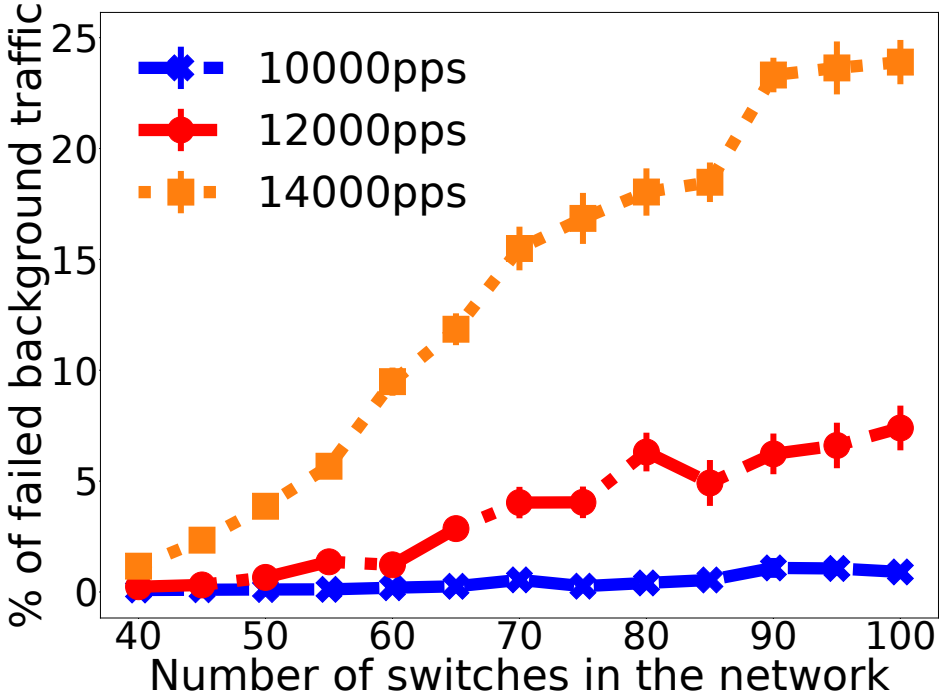
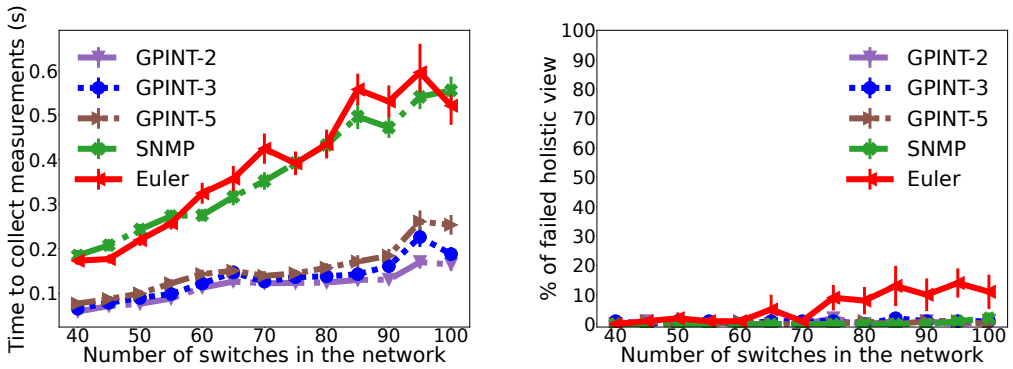


Figure 7.12: The percentage of background traffic's packet losses as $|V|$ increases.

observe $90 - 98\%$ utilization whereas for 10000pps, it is $85 - 92\%$.

In Figure 7.13a, we depict the measurement collection times of path generators under low load (10000pps). The additional queuing delays effects SNMP the least, displaying only the scalability issues. Even though GPINT is affected by queuing delays, due to balanced path generation and low overhead, the effects are limited to only 0.1 seconds. In the case of the Euler, however, we observe a substantial increase in collection times. Since it generates a couple of considerably longer paths, the queuing delays accumulate and can hamper obtaining holistic view up to 0.5 seconds.

To display the effect of packet losses observed in Figure 7.12, we measure how much of the measurement session we failed to conclude in Figure 7.13b. Please note that even one probe packet loss hinders obtaining the holistic view, and we examine how many of the probe packets got lost and prevented obtaining holistic view in Section 7.2.4. For instance, one percent of failed measurement sessions would indicate experiencing some sort of failure in probe packets in one measurement session out of

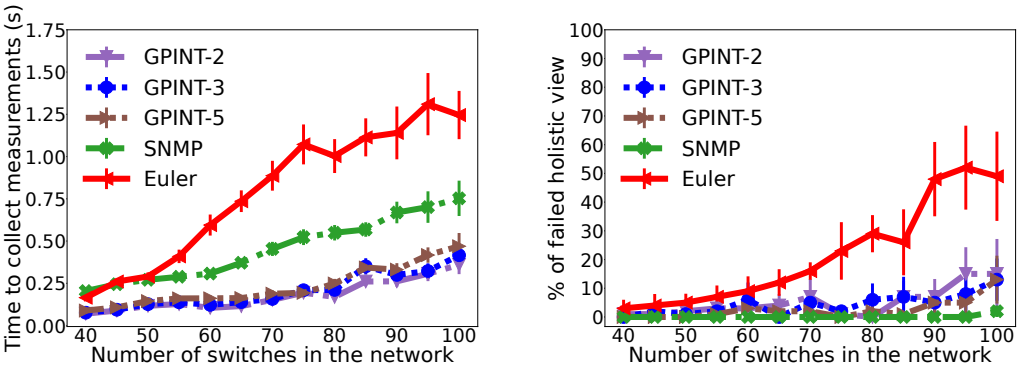


(a) Elapsed time

(b) The ratio of failed holistic view requests

Figure 7.13: Obtained results without recovery mode on low load as $|V|$ increases.

100. The experienced failures only occur due to congestions on switches since there are no losses on the links. Under these conditions, we observe that GPINT is subject to similar conditions of background traffic. That is, $1-2\%$ in background traffic reflects on the measurement losses of GPINT. Whereas for the Euler, we realize the controller fails to construct a holistic view around 10% of the measurement sessions. Even though 10% is surprising to observe, we expect some measurement losses as it generates some longer paths which stay in the network longer and, consequently, subject to packet losses more often than other methods.



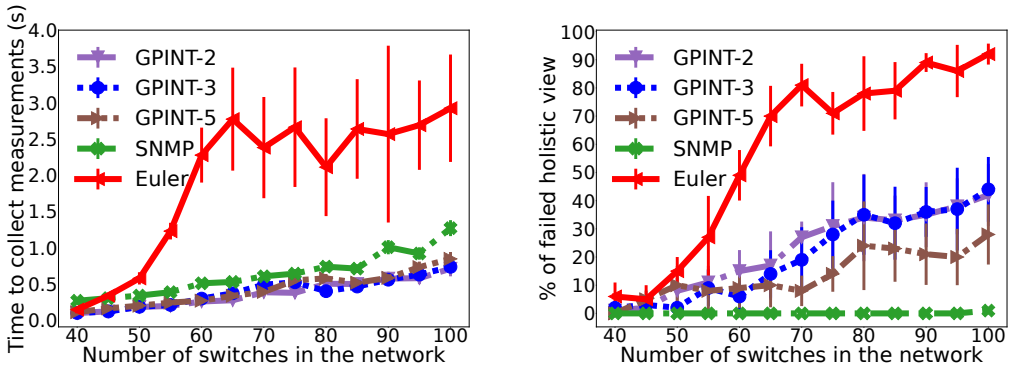
(a) Elapsed time

(b) The ratio of failed holistic view requests

Figure 7.14: Obtained results without recovery mode on medium load as $|V|$ increases.

When we increase the load level to medium (i.e., 12000pps), we observe an unignorable decrease in the Euler method's performance in Figure 7.14a. That is, between $40 - 70$ switches, there is a steep incline in the collection times. Even though the increase gets stabilized after 70 switches, it can still take up to 1.5 seconds to deliver measurements to the controller. On the other hand, both SNMP and GPINT variants deliver reports almost at the same rate with a $0.1 - 0.2$ seconds difference compared to low load measurements.

In Figure 7.14b, we analyze how INT measurement sessions to construct holistic view gets effected by the observed $6 - 7\%$ background traffic losses in Figure 7.12. Even though there is only a 5% difference between packet loss ratio of low and medium loads, we realize around 30% more measurement losses for the Euler and $10-15\%$ for GPINT variations compared to low load levels, Figure 7.13a. Precedentely, the SNMP is the one that is affected the least. These observations indicate that even though INT path generators can achieve fast measurement collections under ideal conditions, they are subject to packet failures and affected deeply under harsher conditions.



(a) Elapsed time

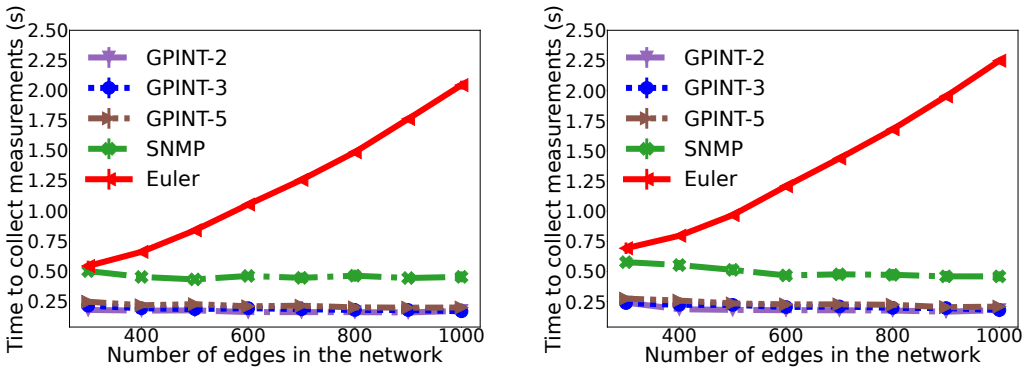
(b) The ratio of failed holistic view requests

Figure 7.15: Obtained results without recovery mode on high load as $|V|$ increases.

It is crucial for monitoring frameworks to collect measurements under heavy loads so that the applications can pinpoint issues and react if there are any. Hence, we challenge path generators with heavy traffic load (i.e., 14000pps) and compare how they perform. Under such conditions, we observe background traffic losses up to 25% in Figure 7.12. Similar to previous analyses, in Figure 7.15a, we measure the required

time to collect measurements. We observe a $0.4 - 0.5$ seconds delay in the delivery times of SNMP and GPINT variations compared to when there is no background traffic, which is expected given the experienced delays in previous analyses. One thing to note here is that as the load in the network increases, the gap between different GPINT variations closes, which is due to better-balanced paths and lesser overhead on switches. Whereas for the Euler, we realize exponential growth in delays up to 60 switches. Even though the collection times seem to stabilize after 60 switches, we observe many fluctuations. The reason for these fluctuations can be seen in Figure 7.15b. After 60% of report losses, the Euler cannot deliver consistent reports, which causes the fluctuations. Under the peak traffic loss (i.e., 25%), the controller fails to construct the holistic view for 90% the sessions for the Euler. When we analyze report losses for GPINT, all variations seem to experience a similar amount of report losses, which is under 50%. However, at 100 switches, we observe report losses for GPINT-5 is around 30%, for GPINT-3 and GPINT-2 45%. As the k increases, the paths get smaller and spend a shorter time in the network. Similarly, longer paths stay longer and have more chance to experience a failure. Accordingly, the GPINT-2 generates the longest paths, followed by GPINT-3 and GPINT-2. However, interestingly, both GPINT-2 and GPINT-3 experience a similar ratio of sessions in which the controller failed to construct a holistic view. It indicates that there might be a threshold in path lengths after which the failure occurrence probability remains stable and increases after another threshold, considering the Euler as well. In the next section, we explore the ratio of the number of failed paths over the number of generated paths in-depth so that we can understand the cause of failures better.

Lastly, it is also important to measure path generators' performances as the connectivity ($|E|$) increases in the network. For this experiment, we fix the number of vertices in the topology ($|V|$) to 100. As the connectivity increases, the hosts get closer. Hence, to engage all of the switches in the network, we deploy 100 switches where 20 of them are traffic-sources, and 80 are traffic-sinks. Due to hardware limitations, we can only increase the background traffic up to 2500pps, where we observe only $0.1 - 0.2\%$ of packet loss which does not affect measurement reports. Therefore, in Figure 7.16, we only display the elapsed times to collect measurements under 2000 and 2500pps background traffic. In Figure 7.16a, we have 2000pps background traf-



(a) When background traffic is 2000pps

(b) When background traffic is 2500pps

Figure 7.16: Time to collect measurements as $|E|$ increases while $|V| = 100$ with different traffic settings.

fic. Compared to Figure 7.11b, we observe a 0.7 seconds increase in the data collection time of Euler at 1000 edges. When we slightly increase the traffic to 2500pps, the gap becomes 0.9 seconds. Under these conditions, one might argue that Euler achieves great performance by traversing all of the edges in the network under 2.5 seconds. However, in reality, Euler can only deliver results after 2.25 seconds of delay even under almost no load. Consequently, the controller will always have a 2.25 seconds delay before obtaining the network's first holistic view, even it deploys paths in high frequency. On the other hand, the GPINT variations do not depend on $|E|$ and can provide results under 0.30 seconds. Furthermore, compared to Figure 7.11b, there is at most 0.15 seconds of delay, which is similar for SNMP as well.

These experiments show us path generators' performances, which are measured by report delivery times. As the load on the network increases, the importance of generating paths with balanced and low overhead becomes evident. For the experiments where we increase $|V|$, the Euler method delivers reports under 1 second when there is no or low load on the network. As we introduce medium load, the Euler fails to generate paths that can be collected under one second. On high load, its reports become unstable but take at least two seconds to collect them. On the other hand, SNMP can collect measurements under 1.2 seconds even under heavy load despite it being unscalable. The GPINT can deliver reports in under one second, even when there is

a heavy load in the network. When we increase the $|E|$ as we fix $|V|$ to 100, Euler delivers measurements in 2.25 seconds under considerably low load. This shows that generating long and unbalanced paths to cover the network causes undeniable delays. On the other hand, GPINT does not depend on $|E|$ as its delivery time is constant and under 0.3 seconds.

Apart from measuring different path generators' performances, we also measure the failed measurement sessions to obtain a holistic view. The experienced failures occur due to congested switches. In our analysis, we observe a correlation between background traffic loss and report losses. For GPINT, we experience $10 - 15\%$ more failures during measurement sessions than background traffic failures. The failures do not exceed 50%, and the controller can still deliver somehow consistent reports. On the other hand, for the Euler method, failures are enormous and prevent the controller from obtaining a holistic view under harsh conditions. Accordingly, the controller struggles to deliver consistent reports and experiences fluctuations in report delivery times. Even though the SNMP probes every switch, we still observe a few report losses, $1 - 2\%$ under heavy load.

These results show that there is a need for a data recovery mechanism to be deployed for frameworks that utilize INT probe paths, regardless of the probe generator. Hence, in the next section, we provide the results with our recovery mechanism.

7.2.4 Enabling INT Recovery Module

In this section, we enable recovery mode for every path generator and measure their performances and the ratio of additionally generated paths. We calculate the ratio as follows $\frac{100\times|P_R|}{|P|}$, where P_R is the set of additionally generated recovery paths and P is the set of initially generated paths by the generators. As for recovery mode parameters, the switches wait for feedback packets for $T_l = 0.1$ seconds. The controller waits for $T_f = 1.5$ seconds for initial feedback packets from switches and attempts to recover them ten times, i.e., $R_a = 10$, before declaring the switch to be unresponsive. Furthermore, to initiate recovery paths in case broken paths reach the controller, we wait for $T_r = 1.5$ seconds as the assumed to be lost packet may still be traversing the network. We set the T_{ro}, resolution time, to 0.5 seconds. Please

note that these recovery parameters can be fine-tuned for each probe generator using the performance results obtained in the previous section as a baseline. Furthermore, throughout the measurements, we observe no unresponsive switches as we successfully recover measurements.

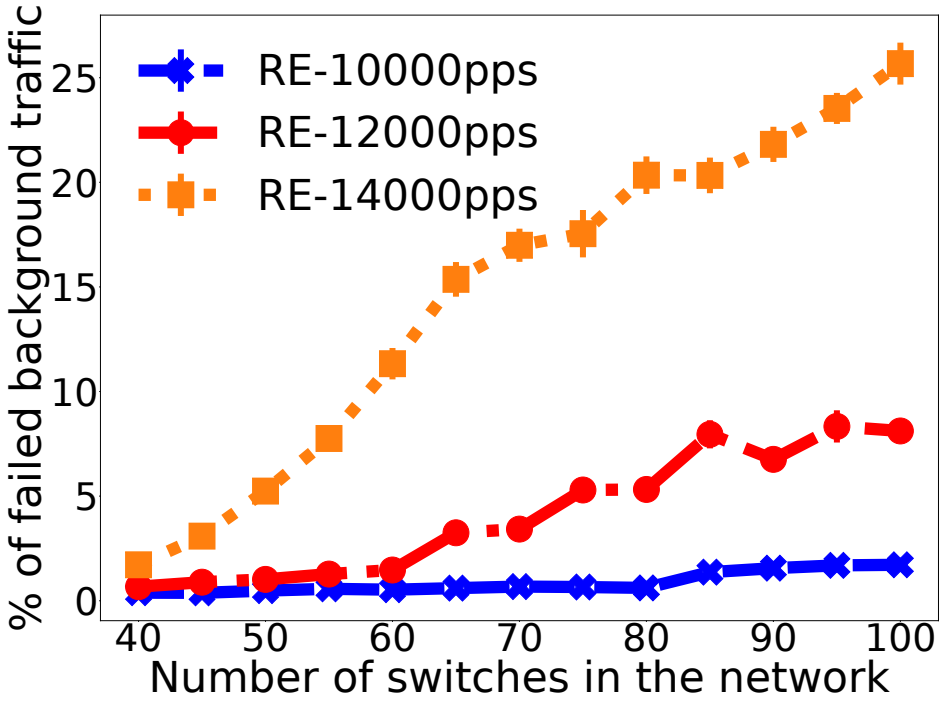
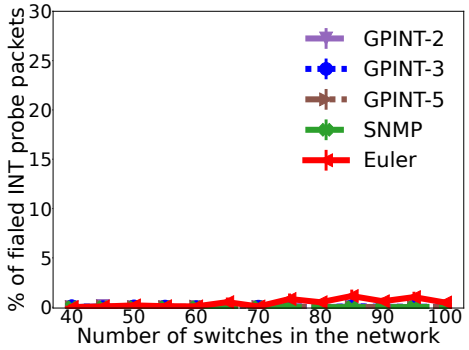


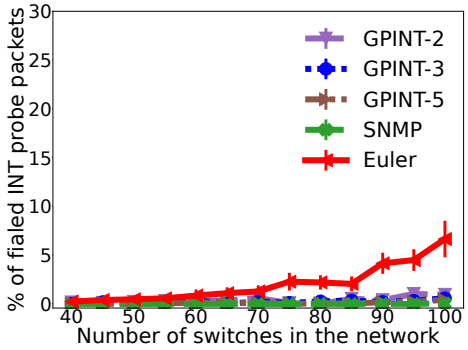
Figure 7.17: The percentage of background traffic's packet losses as $|V|$ increases.

Before we examine our results, we need to analyze whether enabling the data recovery module increases the load on the network or not. In Figure 7.17, we depict the background traffic loss after enabling the data recovery module. We observe that there is a $1-2\%$ increase in the packet losses compared to when we disable the module in Figure 7.12. We examine how this small increase affects measurement collection throughout this section.

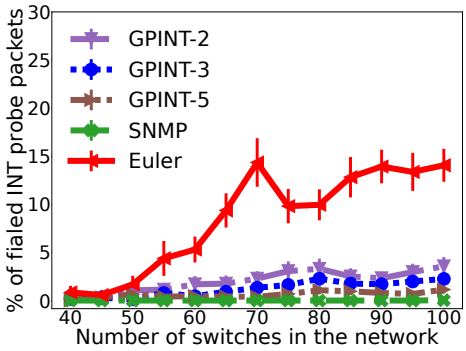
Additionally, in order to understand the metric of the ratio of recovery paths to the number of generated paths better, we need to analyze the number of probe packets that failed when we disable the data recovery module in Figure 7.18. Please note that this measurement depicts the ratio of uncollected paths and different from the percentage of failed INT measurement sessions to gather a holistic view. Observing



(a) Background traffic of 10000pps



(b) Background traffic of 12000pps

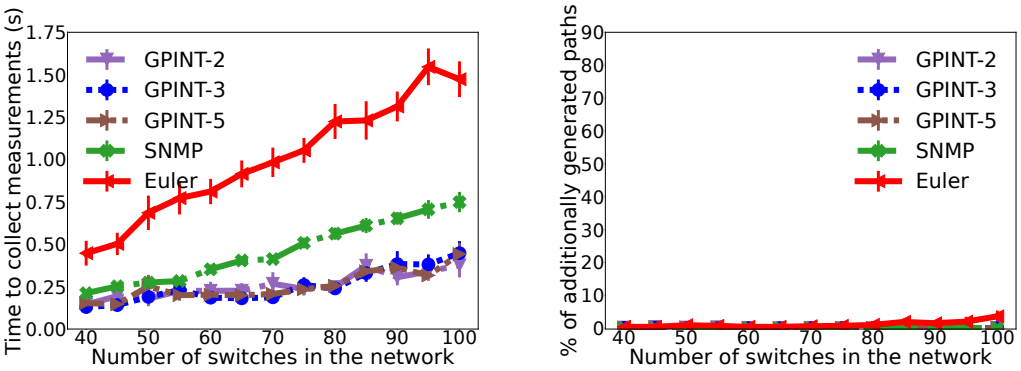


(c) Background traffic of 14000pps

Figure 7.18: The ratio of failed INT probe paths to the number of generated paths on all of the traffic models.

one INT probe packet failure is sufficient to declare that the measurement session is unsuccessful as we failed to obtain a holistic view. However, to conduct a meaningful analysis of how many INT packets the recovery module generates, we also need to depict the percentage of failed or lost probe packets as well. We refer to this figure throughout this section as we discuss how the recovery module performs.

In Figure 7.19a, we measure elapsed times to collect INT reports with recovery mode enabled where the network experiences low load (i.e., 10000pps). We observe an extra delay of at most 0.15 seconds for GPINT variations, and SNMP compared to the recovery mode is disabled 7.13a. For these generators, the ratio of additionally generated paths shown in Figure 7.19b also matches with the probe packet losses depicted



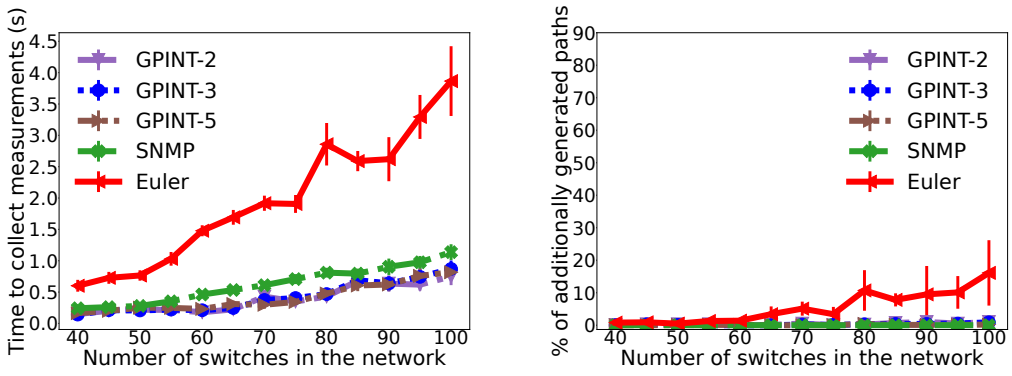
(a) Elapsed time

(b) The ratio of recovery paths

Figure 7.19: Obtained results when recovery mode enabled on low load as $|V|$ increases.

in Figure 7.18a. These observations suggest that the recovery mode's overhead is limited, and its parameters for GPINT variations and SNMP are suitable under low loads. Furthermore, since report losses seldom occur for these generators, time to collect measurements amortizes to $0.3 - 0.4$ seconds for GPINT and 0.7 seconds for SNMP. On the other hand, despite Euler delivering reports under 0.6 seconds in Figure 7.13a, we observe 1.2 seconds of delays in report delivery. Moreover, the ratio of additionally generated recovery paths is about $5 - 6\%$, which is greater than the observed probe packet losses depicted in Figure 7.18a. These may have three indications: (i) the recovery parameters are too aggressive for Euler, (ii) the induced overhead by the recovery mode causes additional packet losses hence degrading the Euler's performance which deploys long paths, or (iii) some of the recovery packets also get lost, and we introduce additional packets to recover them. Even though aggressive parameters (i) would justify the 5% ratio of additional paths, there would not be a 1.2 seconds delay for measurement collection. The reason is aggressive parameters lead to the deployment of new paths, which are shorter than their original forms. The newly deployed paths would traverse less in the network and arrive quicker. Hence, by employing more aggressive parameters, i.e., reducing recovery initiation from 1.5 seconds to a lower value, we may recover some of the performance losses of Euler. However, we would need to generate more paths in such settings, which can exceed 5% easily. To analyze the second possibility (ii), we need to depict the recovery

mode's overhead on the background traffic in Figure 7.17. We observe that the recovery mode causes a $1-2\%$ of traffic loss increase, which occurs as the switches store a copy of packets until the feedback packet arrives from the next hop, eventually consuming more resources. Previous analyses showed that there could be 10% failures in obtaining a holistic view caused by 2% probe packet losses even on $1-2\%$ traffic loss for Euler. Furthermore, the results indicated that even small changes in traffic losses affected Euler's report losses abruptly. Consequently, it is not surprising to observe a $3-4\%$ difference between probe packet losses and the ratio of additionally generated recovery parts as there would not be much of a difference in report losses if traffic losses would have been similar. In the last possibility, (iii), we argue that the recovery packets may also get lost during the procedure. However, this would cause more delays than what we observe as there would be a delay of additional 1.5 seconds to initiate a recovery packet for them. Even though this possibility does not play a significant role for low load traffic settings, we will examine its effect for higher loads in the following analyses.



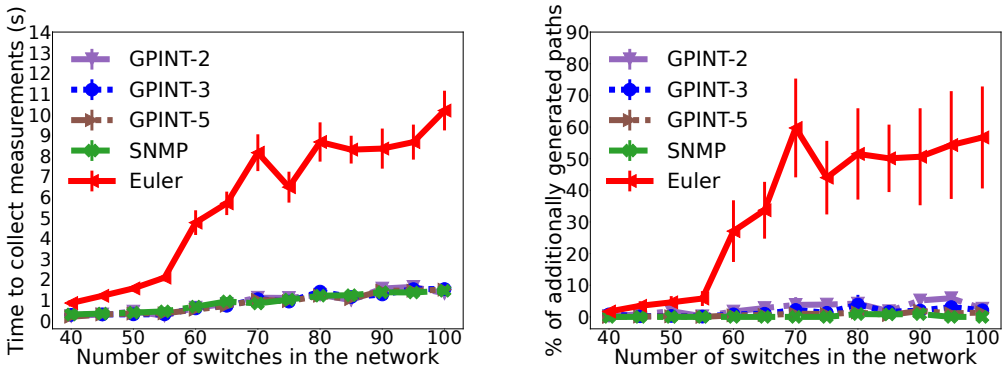
(a) Elapsed time.

(b) The ratio of recovery paths.

Figure 7.20: Obtained results when recovery mode enabled on medium load as $|V|$ increases.

In Figure 7.20a, we analyze path generators' performances under medium load with recovery mode enabled. Similar to the low load case, we observe that GPINT variations and SNMP can deliver reports in under one second. One thing we note here, enabling the recovery mode introduces at most an additional delay of 0.5 seconds for these generators. When we compare the failed INT probe packets (Figure 7.18b) and

the ratio of additionally generated recovery paths for these generators in Figure 7.20b, we see that they are almost the same with at most 1% of difference between. Hence, this indicates recovery parameters are still suitable, but there is room for fine-tuning given the 0.5 seconds of induced delay. When we look at Euler's performance, we observe almost three seconds of extra delays. These delays mainly occur due to recovery packets failing at least twice, given that we initiate recovery packets after 1.5 seconds. Additionally, in Figure 7.18b, we observe almost 7% of probe packet losses for Euler. When we examine the ratio of additionally generated recovery paths in Figure 7.20b, we observe around 20% extra packets, which supports that failure occurs for recovery packets at least twice. It might be possible to reduce the report collection time for Euler if we lower the T_r from 1.5 seconds to one second, for instance. However, the additionally generated paths would increase considerably.



(a) Elapsed time

(b) The ratio of recovery paths

Figure 7.21: Obtained results when recovery mode enabled on high load as $|V|$ increases.

Lastly, we employ recovery mode to where it is the most needed, in high load circumstances. For the first time after we enabled the recovery module, we observe GPINT variations to perform worse by 0.1 seconds than SNMP in Figure 7.21a. Furthermore, they require at least 1.5 seconds to deliver measurements when there are 100 switches in the network. That is, enabling recovery mode requires one second of delay to complete failed reports for these generators. However, one might argue that the recovery module starts sending packets after 1.5 seconds, so the delays should be at least 1.5 seconds. This argument is correct, but the reason why we observe one second delay

is that probe packets seldom fail in the network as depicted in Figure 7.18c especially for GPINT. Additionally, in Figure 7.21b, we see that the recovery module generates almost the same number of paths to that of failed INT probe packets, indicating that a recovery packet does not or rarely fail. Furthermore, the way we implement the recovery module is that we initially assume all paths are failed and ready to be recovered. Hence, the timer, T_r, starts when we activate the recovery module, not from when the failure is detected. For the consecutive packet losses, the timer starts from either when we receive a feedback packet from the path's first switch or when we detect a failure in the data plane. Consequently, the impact of the recovery module is minimal, that we only observe one second delay in the long run. Given that the recovery module generates a minimal number of additional probe packets, it is possible to fine-tune them even further by customizing the parameters to achieve faster delivery at the cost of increased probe packets. We explore fine-tuning option in Section 7.2.4.1 for GPINT. There is also another observation that we need to point out about SNMP and its performance. Even though we introduce almost no additional paths for SNMP, it still takes around the same time with the GPINT to collect measurements. Consequently, this points out the fact that there is a lack of resources for switches. Therefore, even if we make recovery mode more aggressive for GPINT, we may not observe the desired level of effects, and it will be limited. The lack of resources hits the Euler method the hardest, given its longer paths. We observe that it can take 10 seconds to deliver reports. Compared to Euler's performance when recovery mode is disabled (Figure 7.15a), there is a seven seconds difference. We explain the reason for such high delay as follows. The Euler generates long and unbalanced paths, which spend more time in the network and are subject to higher failure chances. Furthermore, the deployed recovery packets can also get lost for these longer paths if the loss is detected early in the path. Hence, experiencing multiple losses on the same path increases the collection time. Let us examine if this is indeed the case. In Figure 7.18c, we see that 15% of the generated probe paths by the Euler experience failure in the network and do not arrive at the controller. On the other hand, the recovery module needs to generate around 60% additional paths after 70 switches. Observing almost four times more generated paths than what is lost indicates two things. The used parameters for the data recovery module are aggressive for Euler, given that it takes around three seconds to deliver reports when the module is disabled (Figure

7.15a). This causes the recovery module to declare paths failed more quickly and leads to deploying more paths. However, this would have caused at most two or three times more additional packets than the lost probe packets and would not explain seven seconds of delay. Accordingly, the reason why we observe such an increase is the combination of the aggressive parameters and the fact that we observe recovery packet losses. It is possible to change parameters for Euler to generate fewer paths. However, in such a configuration, the experienced delay would be a lot higher than seven seconds. Consequently, for path generators such as Euler that generate long and unbalanced paths, it might be better to deploy a different strategy as a recovery option. For instance, when a failure is detected, we might be able to deploy two paths, one from where it is detected to be broken and one from the end of the path going through the reversed direction. In this way, we might be able to deliver reports faster when such path generators are used under harsh conditions. Another option would be to shorten the Euler's longer paths at the cost of additional paths to obtain better reliability.

These results show that the recovery mode works as intended with negligible low overhead. Furthermore, we observe the effect of generating balanced paths once more as they can almost seamlessly be recovered. Additionally, balanced path generation with shorter paths provides the best reliability as we observe the least packet failures for GPINT-5 in Figure 7.18c. With the introduction of recovery mode, we observe that the gap between different variations of GPINT closed in terms of report collection times. In some cases, we need to introduce more recovery paths than intended (e.g., the ratio for GPINT-2 slightly peaking between $90 - 95$ switches in Figure 7.21b). However, the recovery mode contains multiple parameters that can be fine-tuned for any given path generator and their performances without recovery mode. In the following section, we explore the tuning recovery module for GPINT-3. We also observe that the recovery module works as intended, even for probe generators that produce looping and unbalanced paths. However, the module's contributions are limited, especially if generated paths are incredibly long, which experience multiple failures. For such path generators, a different kind of recovery option can be applied, such as deploying two different recovery modules when T_r timeout occurs, one from where a path is left off and the other from the end of the path in a reversed direction.

7.2.4.1 An Example Tuning for GPINT

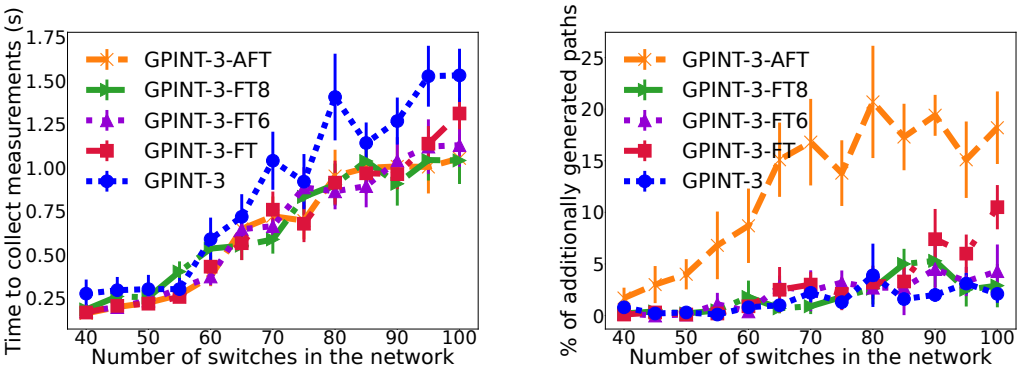
By changing the data recovery module's parameters, it is possible to configure the module's reaction time. For instance, one can assume the probe packet is lost once a switch generates and sends a recovery packet to the controller, without giving a benefit of the doubt that it may still be traversing the network. Such an approach would significantly increase the number of additionally generated paths but might be considered in some cases. In this section, we explore fine-tuning of GPINT-3 on high traffic load and point out the data recovery module's limitations. The packet failures experienced in this section only happen due to congestions occur on the switches.

Table 7.2: Different Configuration Settings of Data Recovery Module for GPINT-3

	T_f	T_r	T_l	T_{ro}	A
GPINT-3-FT8	0.8	0.8	0.1	0.2	✗
GPINT-3-FT6	0.6	0.6	0.1	0.2	✗
GPINT-3-AFT	0.4	0.4	0.1	0.2	✓
GPINT-3-FT	0.4	0.4	0.1	0.2	✗
GPINT-3	1.5	1.5	0.1	0.5	✗

In Table 7.2, we provide different settings of the data recovery module's parameters to improve GPINT-3's performance. In the table, **A** stands for whether the aggressive mode is enabled or not, and in this set of parameters, we activate the aggressive mode for only one set of configurations. As a naming convention, we use FT as an abbreviation for fine-tuned, and the number after FT indicates timeout variables T_r and T_f since we set both equal in this analysis. Additionally, we set R_a for all these configurations to 10, which is not shown in the table.

In order to fine-tune the data recovery module, we need to analyze the probe generator's performance under a given network environment. In Figure 7.15, we have the results of GPINT-3 with 14000pps background traffic and no recovery module enabled. Accordingly, we can expect GPINT-3 to deliver results under 0.6 seconds. With the data recovery module's default parameters, we see that GPINT can take approximately 1.5 seconds in Figure 7.21. Since the percentage of lost probe packets



(a) Elapsed time

(b) The ratio of recovery paths

Figure 7.22: Comperision results of fine-tuned recovery module for GPINT-3 to default parameters on high load as $|V|$ increases.

(Figure 7.18c) and additionally generated paths (Figure 7.21b) match for this configuration, we can actually configure module's parameters to make it more aggresive or proactive. In Figure 7.22, we explore this option and provide results of different sets of configurations shown in Table 7.2. We notice that up to 60 switches, there is not much of a difference in the report delivery times despite different sets of configurations as depicted in Figure 7.22a. That is because a probe packet failure rarely occurs. Hence their impact is negligible. However, in Figure 7.22b, we observe a significant difference between GPINT-3-AFT and the rest up to 60 switches even though failures rarely occur. This indicates that we deduce a path is broken too quickly with the aggressive mode. Since the report delivery times are similar for all the configurations, these deployed recovery packets have no impact, meaning that these are mainly false alarms. After 60 switches, all of the configuration parameters display a similar performance in both the elapsed times and the percentage of generated paths. However, there is one exception that occurs at 100 switches. GPINT-3-FT requires 1.25 seconds to collect measurements, which is about 0.20-0.25 seconds more than the rest of the fine-tuned pack. When we examine the reason, we observe that GPINT-3-FT deploys about twice more paths than the rest except GPINT-3-AFT. This observation points to the limitation of controller architecture design. We deploy both the data recovery module and the monitoring framework on the same hardware. Hence, when we set more aggressive, busy-wait-like parameters for the recovery module, it affects

the performance when there is a lack of resources. The GPINT-3-AFT covers this performance degradation by deploying recovery paths more aggressively and at the cost of deploying more than necessary.

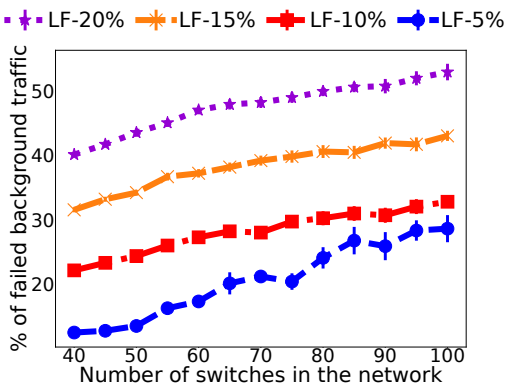
Overall, we can improve the report collection times of GPINT-3 under harsh conditions by 0.5 seconds. Consequently, this shows the data recovery module's limitations for switch-level congestions and indicates developing a smarter approach. For instance, we could be initiating recovery from a reversed direction rather than resuming where paths left off, especially when it is at the beginning. Another approach could be designing INT paths with resilience in mind. Even with aggressive fine-tuned parameters, we generate up to 30% more paths. Hence, if we designed GPINT to improve resilience with a similar number of paths, we would collect results even faster. Accordingly, even though the data recovery module can achieve recovery for any probe generator in a considerable short amount of time with correct parameters, it has limitations and would work best with path generators that consider resilience.

In the following section, we introduce link failures to the network, which decreases the load on the switches but introduces a new challenge for the recovery module.

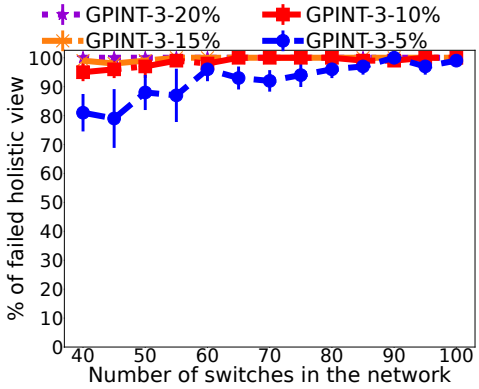
7.2.4.2 Data Recovery Module on a Network with Link Failures

In this section, we introduce link failures to our simulation environment and analyze how this affects measurement deliveries. We have four different link failure (LF) models, 5%, 10%, 15%, and 20%. We use GPINT-3 as a traffic generator and fine-tuned (FT) parameters in the data recovery module for this test. Furthermore, there is 14000pps background traffic in the network for all failure models. Consequently, two types of failures can be observed in this analysis. First, packets and reports may fail due to resource limitations in switches, as observed in previous analyses. Alternatively, they can get lost due to packet drops in the links. We gradually increase the link failure chance, which, in return, decreases the resource usage in the switches due to processing fewer packets. Therefore, on higher failure chances, packets mainly drop due to link failures rather than resource limitations.

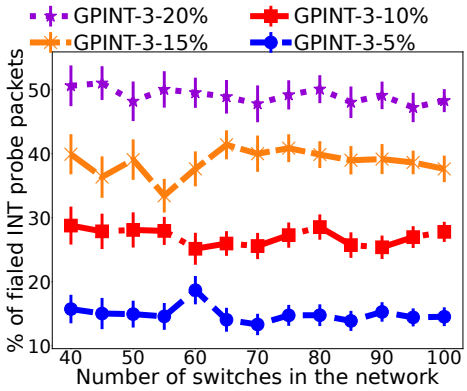
In Figure 7.23, we measure how link failures effect both traffic and report packets



(a) Background traffic loss



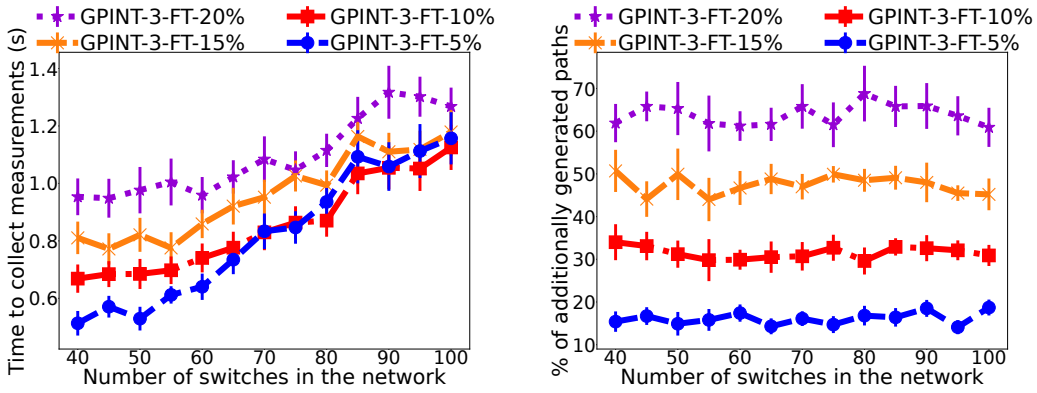
(b) The ratio of failed holistic view requests



(c) The ratio of failed probe packets

Figure 7.23: Background traffic and measurement report losses on high traffic load with link failures.

losses. When there are no link failures in the network, we observe at most 25% of background traffic loss (Figure 7.12). The congestion related failures cause 45% of measurement session to collect holistic view report losses for GPINT-3 (Figure 7.15b). Furthermore, when the controller failed to obtain a holistic view, there are around 2.5% probe packet losses observed in Figure 7.18c. When we introduce link failures, we do not observe much of a traffic loss increase on 5% and 10% failure chance. However, after 10%, we see that the background traffic losses can reach up to 40% and 55% for 15% and 20% failure chances. On the other hand, we realize that even 5% failure chance is enough to hinder consistent measurement deliveries as the failures in obtaining a holistic view increase from 45% to $80 - 100\%$ (Figure 7.23b). Furthermore, the probe packet losses take a great leap from 2.5% to 15% as depicted



(a) Elapsed time

(b) The ratio of recovery paths

Figure 7.24: Comperision results of fine-tuned recovery module for GPINT-3 to default parameters on high load as $|V|$ increases.

in Figure 7.23c. Hence, even subtle instabilities in the network can cause significant telemetry losses and degrade the controller's ability to reach accurate decisions.

In Figure 7.24, we depict the results of the data recovery module, fine-tuned for GPINT-3, on a network with enabled link failures. The parameter configurations can be found in Table 7.2. We analyze the time required to deliver results in various link failure conditions in Figure 7.24a. We observe that there is a clear difference in the report delivery times between different failure conditions up to 60 switches. Afterward, we realize that GPINT-3-FT's performance is similar under both 5% and 10% failure chances. After 80 switches, the difference between 15% and the lower values also ceases. This behavior can be explained as follows. As the failure chances increase, the switches process fewer packets. Consequently, the load on the switches and switch-related failures decrease considerably. Due to the lessening burden on the switches, INT packets spend less time on the queue and being forwarded faster. Since switch-related failures decrease, when switches detect failures, it is more likely due to feedback packet losses. Hence, switches update the controller more frequently on INT packets' whereabouts so that the data recovery module can release additional paths close to endpoints rather than starting from the beginning. Accordingly, INT packet failures due to switch congestions and link failures balance themselves out for low and high chances of link failures, resulting in deliveries at approximately the

same time for all failure conditions. However, in ideal conditions where resource limitations solely occur, the expected behavior under harsh conditions should be similar to what we observe up to 70 switches.

In Figure 7.24b, we depict the ratio of additionally generated packets to actual number so that the module can cover failures. The first thing we observe is that ratios for all of the conditions are rather stable. Even though the ratio is stable, the number of generated paths increases with the increasing number of vertices. Hence, the module generates more and more paths as the topology grows under the same harsh conditions. The reason why we observe a constant ratio is the following. The GPINT can generate almost perfectly balanced paths. With the increasing number of paths, the generated paths' length remains relatively stable due to them being balanced. Therefore, we observe a similar degree of failures and the ratio of additionally generated paths to cover them. In order to analyze the ratios themselves, let us go back to examine the ratio of failed probe packet under these conditions in Figure 7.23c. Compared to this figure, we realize that the recovery module generates almost the same percentage of additional paths to what is lost for 5% failure chance. We observe around 5% more paths introduced by the module for 10% failure chance. For 15% and 20%, the module generates 10-20% more paths than the lost probe packets. It means that after 10% failure chance on the links, the deployed recovery packets also suffer from losses, and the module generates more packets to complete the session. Regardless of these extra recovery packets, the total number of generated paths to obtain a holistic view is considerably smaller than probing every switch individually. For instance, under the 20% failure chance, the module generates around 70% packets. Given that the GPINT-3 generates 24-25 paths to cover whole network for 100 switches (Figure 7.9a), the total number of generated paths becomes $42 - 43$.

This experiment shows that the data recovery module can perform considerably well on networks that may experience unstable links, such as wireless networks. By fine-tuning the configurable parameters that the module offers, one can adapt it to any condition. However, it still has its limitations that we discussed in previous analyses and works best with path generators that can generate balanced paths.

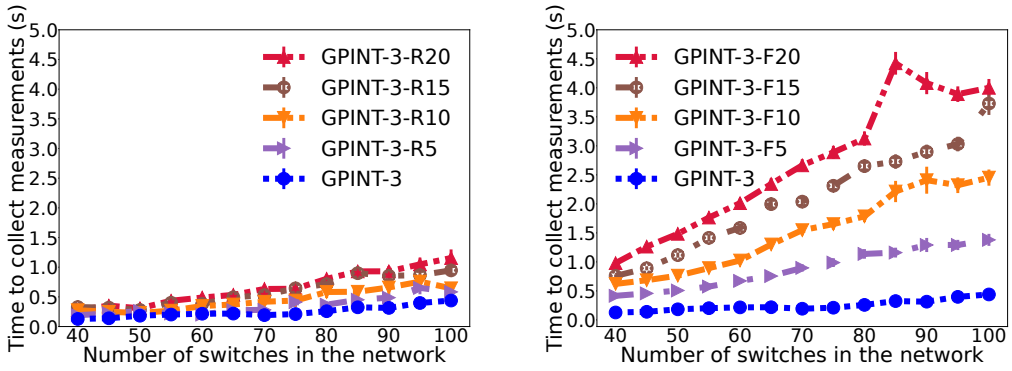
In the following section, we analyze how the customizability of requesting a various

number of measurements affects performance.

7.2.5 The Effect of Request Ranges

During previous analyses, we only requested one measurement from switches. However, since we provide custom ranges in our customized INT design, it is important to depict how different ranges affect performance. The custom ranges can be thought of as the number of loops packets travel before being forwarded. During the measurement loops, there can be many header insertions. In this analysis, we insert only one measurement header in each loop, and switches experience low traffic load (10000pps). To avoid any measurement losses and fluctuations in results, we enable the recovery mode with the default parameters as well. As an underlying INT probe generator, we use GPINT-3.

In Figure 7.25a, we show the affect of different request ranges. We observe that wide request ranges decrease the performance of the probe generator. Up to 15 loops, the controller can collect requests in under one second. When we increase loop count to 20, we see that elapsed time is around 1.2 seconds. While we can collect more measurements with customized ranges, it is clear that there is a price to pay in terms of collection time.



(a) Report collection with different request ranges (b) Report collection with different frequencies

Figure 7.25: The effects of deploying different request ranges and frequencies on collection time with recovery mode enabled on low load.

Another approach to obtain the same amount of information at the same time is to increase the frequency of the measurement, which is to release multiple INT probes with different request headers. Before we start the analysis, there are few things to note. To obtain these results, we use the same controller architecture, which releases probe packets of a measurement request concurrently using a processing pool. For multiple requests, however, the controller processes them sequentially. For example, if the frequency is two, we generate two different requests within the same measurement loop. While we release the first request's probe packets concurrently, the second one waits. Hence, there is a delay before releasing multiple requests as each of them waits for the previous requests' release. Even though this can be mitigated by employing concurrency models for efficiency, the same problem would again appear for higher frequencies since the controller's resources are limited. Therefore, we decided to omit the implementation of a concurrency model for this analysis but be aware that with a distributed or controller with unlimited resources, the frequency approach would perform better. In Figure 7.25b, we depict our results using a controller with limited resources. If we go back and consider how SNMP, which is clearly bottlenecked by the available resources, performed under similar conditions (Figure 7.19a), we see that for 100 switches, the controller takes about 0.70 seconds to release 100 probes and collect them. Given that the GPINT-3 generates approximately 25 paths for 100 switches (Figure 7.9a), by setting frequency to five, the controller needs to release 125 probe packets in total. With the help of SNMP's results, we estimate $0.17 - 0.18$ seconds of additional delay to release 25 packets. Then, the controller takes about 0.88 seconds only to release all of the requests. The calculation of elapsed time to collect measurements depends on the latest arrival, which takes about 0.35 seconds for GPINT-3 (Figure 7.19a). Consequently, the controller collects results only after 1.2 seconds of delay when the frequency is set to five. Clearly, we observe a scalability problem given the limited resources our controller has. For higher frequency levels, this problem is more evident. For instance, the controller takes four seconds to collect all measurement requests when we set the frequency to 20.

In the retrospect of this analysis, we realize the customizable measurement ranges are a valuable addition for controllers with limited resources even though they might

delay telemetry collection by a small amount. On the other hand, if the controller had unlimited resources, increasing the frequency levels would deliver reports a lot faster. However, even for these controllers, customizable ranges can still be helpful since higher frequencies might affect the commercial traffic.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this chapter, we conclude the thesis and provide future directions.

8.1 Conclusion

In this work, we point out the scalability issues and design tradeoffs for the in-band network telemetry-based monitoring systems. We define several requirements to address these tradeoffs to design an efficient monitoring system that can ensure reliable communication. We formulate these requirements and define the Balanced Simple INT-Path problem. Furthermore, to understand the relationship between the tradeoffs, we also include models individually targeting each requirement. However, finding optimal solutions satisfying the given requirements is too complex, making them infeasible to deploy in a real-world scenario. Hence, we propose a graph partitioning-based heuristic, GPINT, to target all three requirements. GPINT is an extension to Kernighan-Lin's graph partitioning algorithm, fine-tuned for generating simple paths that satisfy defined requirements. In our analysis, we measure how well GPINT performs using the BSIP as a reference point. Based on the numerical optimality analysis, we observe that GPINT excels in two objectives out of three. Those are the standard length deviation and the number of shared switches. However, when it comes to the number of generated paths, we realize that GPINT can only reduce the suggested maximum number of paths by one. Whereas in the case of BSIP, it can halve the number of generated paths. Even though the number of generated paths is the only downside of GPINT, by providing different k, the ratio of the suggested maximum number of paths, one can adapt GPINT to situations where few or many

paths are desired. Further in our analysis, we measure the scalability of GPINT and compare it to Pan et al.'s Euler method (proposed in recent work) and SNMP like polling-based method. The numerical results indicate that GPINT is more scalable than Euler's method and can be utilized to carry any information due to its minimal overhead. This flexibility can be a critical factor in designing an efficient and scalable monitoring system.

To validate our findings in numerical analysis, we test path generators in a simulation environment. The Euler performs considerably well despite the poor results obtained in the numerical analysis if there is no background traffic. However, as we introduce background traffic and more challenging environments for switches, the envisioned performance difference becomes evident. On the other hand, our proposal GPINT can quickly deliver results, even under harsh conditions, which shows that our requirements are crucial in achieving effective monitoring.

Under harsh conditions, we realize that INT is vulnerable to packet losses, probably more than any other framework as a single INT packet carries accumulated information. Therefore, we design a data recovery module as an auxiliary application to the INT framework by P4's flexibility about introducing custom protocols. The data recovery module requires switches to send feedback packets as INT probes travel through the network. Using these feedback packets, the data recovery module helps us detect broken paths and introduce new probe packets to recover them from where they left off. The module has several customizable parameters that can change how it behaves when a broken path is observed. For instance, it can release probe packets immediately or can wait for some time to see whether it was a false alarm or not. In our analysis, we discuss the effects of these parameters in detail. Furthermore, we introduce link failures in our simulation tests and highlight limitations of the data recovery module.

Lastly, we extend the INT protocol to support dynamic measurement types and ranges. For example, the controller can request a specific measurement type that may differ for each request, or it can specify a different range of measurements from each switch. We measure the effect of custom measurement ranges by comparing them to deploying INT in higher frequencies to achieve similar gathering. The results show dynamic

measurement ranges can ease the load on the controller with minimal delay. However, if the controller has infinite or abundant resources, deploying INT probes in high frequencies would be quicker.

8.2 Future Work

As future work, we will improve GPINT so that it can reduce the number of generated paths while keeping other objectives at optimal levels. Furthermore, in this work, we only considered a single monitoring application while, in reality, it is safe to assume there are many, each with different measurement requirements. Hence, we plan to extend GPINT so that we can supply information to each monitoring application by leveraging our extended INT protocol.

In our numerical analysis, we observe that k plays a significant role for GPINT in the number of generated paths. Since GPINT can excel in the other two objectives, the length deviation and the number of shared nodes, their performances in simulation results are similar as more paths indicate smaller lengths and hence, less travel time in the network. However, these subtle differences might be important for some network conditions, and better control over the k value may be required. Therefore, we aim to find optimal k value for any given network state and requirements as future work.

Even though we have a data recovery module in place for INT, we realize that failures are still a challenge that needs to be addressed for fast report delivery. Hence, we aim to improve the speed of recovery and also the resilience of the GPINT so that even some paths fail, there will always backup ones traversing the network. Furthermore, we aim to develop a monitoring application on top of the data recovery module to pinpoint broken links or switches accurately.

In this work, one of our assumptions is that we operate on a homogeneous network that all switches are P4 programmable. However, in reality, this assumption can be considered as too strict as it is possible to have heterogeneous networks making the transition from OpenFlow switches to P4 ones, for instance. The heterogeneous networks offer various challenges on how INT can be deployed and utilized for such networks. One of the challenges is how to integrate INT with switches that cannot

recognize the protocol. One idea could be forwarding them according to rules entered by the controller so that the probe packet can still flow through the network and collect measurements from ones that support INT. Additionally, it might be possible to program traditional switches to send measurements directly to the controller when they observe INT if we can couple INT protocol with an event triggering mechanism for those switches. Another challenge is how to design INT probe paths for these networks. It is possible to have disconnected P4 switches, meaning that there is no direct link between two P4 switches, but they connect over OpenFlow switches. In such topology, we need to generate INT paths by considering the intermediate switches between two P4 switches as they induce additional delays. One idea could be introducing weights on the links of the graph that path generators operate on. The weights can be the number of intermediate switches that do not support INT protocol. Consequently, probe generators might be adjusted to balance the weight of paths rather than their lengths to achieve concurrent delivery. We leave the execution of deploying INT on heterogeneous networks and applying mentioned ideas as future work.

In this study, we do not consider propagation delays between two switches. Despite that, we observe considerable delays due to queuing, and processing delays occur at the switches, especially when the generated paths are long. In reality, propagation delays also play an important role in report delivery. Accordingly, they may present a challenge to be addressed in order to deploy the INT framework on networks that span a country, for instance. We leave exploring challenges that these topology structures may offer as future work.

Lastly, we realize that INT can be used as a debugging tool, especially for code coverage or tracing the data plane logic. For instance, when a switch observes an INT packet, it inserts the operations it performs on the packet as if it belonged to a commercial traffic packet. When the packet arrives at the controller, it will obtain every operation performed on the packet through its journey and can compare if there are accurate and expected. As a consequence, the maintainers can pinpoint any logical errors in the data plane.

REFERENCES

[1] Ola Salman and Imad Elhajj and Ali Chehab and Ayman Kayssi, "IoT survey: An SDN and fog computing perspective," *Computer Networks*, vol. 143, pp. 221–246, 2018.

[2] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[3] T. Benson, A. Akella, and D. Maltz, "Unraveling the Complexity of Network Management," in *Proc. of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, p. 335–348, 2009.

[4] J. Pan, S. Paul, and R. Jain, "A survey of the research on future internet architectures," *IEEE Communications Magazine*, vol. 49, no. 7, pp. 26–36, 2011.

[5] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker, "Software-Defined Internet Architecture: Decoupling Architecture from Infrastructure," in *Proc. of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pp. 43–48, Association for Computing Machinery, 2012.

[6] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox, "Intelligent Design Enables Architectural Evolution," in *Proc. of the 10th ACM Workshop on Hot Topics in Networks*, pp. 1–6, Association for Computing Machinery, 2011.

[7] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.

[8] P. Tsai, C. Tsai, C. Hsu, and C. Yang, "Network Monitoring in Software-Defined Networking: A Review," *IEEE Systems Journal*, vol. 12, no. 4, pp. 3958–3969, 2018.

[9] A. Pras, J. Schonwalder, M. Burgess, O. Festor, G. M. Perez, R. Stadler, and B. Stiller, "Key research challenges in network management," *IEEE Communications Magazine*, vol. 45, no. 10, pp. 104–110, 2007.

[10] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple network management protocol," tech. rep., STD 15, RFC 1157, SNMP Research, Performance Systems International, MIT, 1990.

[11] B. Claise, "Cisco Systems NetFlow Services Export Version 9," RFC 3954, 2004.

[12] P. Phaal, S. Panchen, N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks," RFC 3176, 2001.

[13] M. Yang, Y. Li, D. Jin, L. Zeng, X. Wu, and A. V. Vasilakos, "Software-Defined and Virtualized Future Mobile and Wireless Networks: A Survey," *Mobile Networks and Applications*, vol. 20, no. 1, pp. 4–18, 2014.

[14] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software defined networking: State of the art and research challenges," *Computer Networks*, vol. 72, pp. 74–98, 2014.

[15] F. Bannour, S. Souihi, and A. Mellouk, "Distributed SDN Control: Survey, Taxonomy, and Challenges," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.

[16] H. I. Kobo, A. M. Abu-Mahfouz, and G. P. Hancke, "A Survey on Software-Defined Wireless Sensor Networks: Challenges and Design Requirements," *IEEE Access*, vol. 5, pp. 1872–1899, 2017.

[17] C. N. Tadros, M. R. M. Rizk, and B. M. Mokhtar, "Software Defined Network-Based Management for Enhanced 5G Network Services," *IEEE Access*, vol. 8, pp. 53997–54008, 2020.

[18] M. Hicham, N. Abghour, and M. Ouzzif, "5G mobile networks based on SDN concepts," *Int. J. Eng. Technol.*, vol. 7, no. 4, pp. 2231–2235, 2018.

[19] A. Yassine, H. Rahimi, and S. Shirmohammadi, "Software defined network traffic measurement: Current trends and challenges," *IEEE Instrumentation & Measurement Magazine*, vol. 18, no. 2, pp. 42–50, 2015.

[20] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, 2013.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, 2008.

[22] S. Kaur, K. Kumar, and N. Aggarwal, "A review on P4-Programmable data planes: Architecture, research efforts, and future directions," *Comput. Commun.*, vol. 170, pp. 109–129, 2021.

[23] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–7, 2018.

[24] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends," 2021.

[25] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[26] E. Kaljic and A. Maric and P. Njemcevic and M. Hadzialic, "A Survey on Data Plane Flexibility and Programmability in Software-Defined Networking," *IEEE Access*, vol. 7, pp. 47804–47840, 2019.

[27] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, vol. 15, 2015.

[28] T. Pan, E. Song, Z. Bian, X. Lin, X. Peng, J. Zhang, T. Huang, B. Liu, and Y. Liu, "INT-Path: Towards Optimal Path Planning for In-band Network-Wide Telemetry," in *Proc. of the IEEE INFOCOM - IEEE Conference on Comput. Commun.*, pp. 487–495, April 2019.

[29] Y. Lin, Y. Zhou, Z. Liu, K. Liu, Y. Wang, M. Xu, J. Bi, Y. Liu, and J. Wu, "NetView: Towards on-demand network-wide telemetry in the data center," *Computer Networks*, vol. 180, pp. 1–6, 2020.

[30] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer Programming Formulation of Traveling Salesman Problems," *J. ACM*, vol. 7, no. 4, p. 326–329, 1960.

[31] G. Simsek, Ergenç, Doğanalp, and E. Onur, "Efficient Network Monitoring via In-band Telemetry," in *Proc. of the 17th International Conference on the Design of Reliable Communication Networks DRCN 2021*, IEEE, 2021.

[32] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[33] T. Qu, R. Joshi, M. C. Chan, B. Leong, D. Guo, and Z. Liu, "SQR: In-network Packet Loss Recovery from Link Failures for Highly Reliable Datacenter Networks," in *Proc. of the IEEE 27th International Conference on Network Protocols*, pp. 1–12, 2019.

[34] Haoyu Song and Fengwei Qin and Pedro Martinez-Julia and Laurent Ciavaglia and Aijun Wang, "Network Telemetry Framework," Internet-Draft draft-ietf-opsawg-ntf-05, Internet Engineering Task Force, Feb. 2021. Accessed: March 2, 2021.

[35] Cisco, "Model Driven Telemetry." `https://www.cisco.com/c/en/us/solutions/service-provider/cloud-scale-networking-solutions/model-driven-telemetry.html`, 2019. Accessed:March 2, 2021.

[36] Arista, "Telemetry and Analytics." `https://www.arista.com/en/solutions/telemetry-analytics`, 2017. Accessed:March 2, 2021.

[37] Juniper, "Overview of the Junos Telemetry Interface." `https://www.juniper.net/documentation/us/en/software/`

junos/interfaces-telemetry/topics/concept/junos-telemetry-interface-oveview.html, 2019. Accessed:March 2, 2021.

[38] Huawei, "Overview of Telemetry, Network Management and Monitoring." https://support.huawei.com/enterprise/en/doc/EDOC1100004357/81fea299/overview-of-telemetry, 2019. Accessed:March 2, 2021.

[39] The P4.org Applications Working Group, "In-band Network Telemetry (INT) Dataplane Specification." https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf, 2020. Accessed: March 2, 2021.

[40] C. A. Sunshine, "Source Routing in Computer Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 7, pp. 29–33, 1977.

[41] "sFlow." https://sflow.org/, 2018.

[42] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks," in *IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–8, 2014.

[43] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN," in *Proc. of the IEEE 34th Int. Conf. on Distributed Computing Systems*, pp. 228–237, 2014.

[44] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A low cost network monitoring framework for Software Defined Networks," in *Proc. of the IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–9, 2014.

[45] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek, and G. Antichi, "Enabling Event-Triggered Data Plane Monitoring," in *Proc. of the Symposium on SDN Research*, SOSR '20, pp. 14–26, 2020.

[46] M. Charikar, K. Chen, and M. Farach-Colton, "Finding Frequent Items in Data Streams," vol. 2380, pp. 693–703, 2002.

[47] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and its Applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[48] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *Proc. of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 29–42, 2013.

[49] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *Proc. of the ACM SIGCOMM Conference*, pp. 101–114, 2016.

[50] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-Wide Measurements," in *Proc. of the ACM SIGCOMM*, SIGCOMM '18, pp. 561–575, 2018.

[51] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "NitroSketch: Robust and General Sketch-Based Monitoring in Software Switches," in *Proc. of the ACM SIGCOMM*, pp. 334–350, 2019.

[52] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-Directed Hardware Design for Network Performance Monitoring," in *Proc. of the ACM SIGCOMM*, pp. 85–98, 2017.

[53] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-Driven Streaming Network Telemetry," in *Proc. of the ACM SIGCOMM*, pp. 357–371, 2018.

[54] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow," in *Proc. of the 2018 USENIX Annual Technical Conference*, pp. 823–835, 2018.

[55] Y. Zhou, D. Zhang, K. Gao, C. Sun, J. Cao, Y. Wang, M. Xu, and J. Wu, *Newton: Intent-Driven Network Traffic Monitoring*, pp. 295–308. ACM, 2020.

[56] Y. Zhou, J. Bi, T. Yang, K. Gao, J. Cao, D. Zhang, Y. Wang, and C. Zhang, "HyperSight: Towards Scalable, High-Coverage, and Dynamic Network Mon-

itoring Queries," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1147–1160, 2020.

[57] N. Van Tu, J. Hyun, and J. W. Hong, "Towards ONOS-based SDN monitoring using in-band network telemetry," in *Proc. of the 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pp. 76–81, 2017.

[58] Serkantul, "Prometheus INT exporter." `https://github.com/serkantul/prometheus_int_exporter`. Accessed:March 2, 2021.

[59] N. V. Tu, J. Hyun, G. Y. Kim, J. Yoo, and J. W. Hong, "INTCollector: A High-performance Collector for In-band Network Telemetry," in *Proc. of the 14th International Conference on Network and Service Management (CNSM)*, pp. 10–18, 2018.

[60] Y. Kim, D. Suh, and S. Pack, "Selective In-band Network Telemetry for Overhead Reduction," in *Proc. of the IEEE 7th International Conference on Cloud Networking (CloudNet)*, pp. 1–3, 2018.

[61] D. Suh, S. Jang, S. Han, S. Pack, and X. Wang, "Flexible sampling-based in-band network telemetry in programmable data plane," *ICT Express*, vol. 6, no. 1, pp. 62–65, 2020.

[62] T. Pan, E. Song, C. Jia, W. Cao, T. Huang, and B. Liu, "Lightweight Network-Wide Telemetry Without Explicitly Using Probe Packets," in *Proc. of the IEEE INFOCOM - IEEE Conference on Computer Communications Workshops*, pp. 1354–1355, 2020.

[63] Ben Basat, Ran and Ramanathan, Sivaramakrishnan and Li, Yuliang and Antichi, Gianni and Yu, Minian and Mitzenmacher, Michael, "PINT: Probabilistic In-Band Network Telemetry," in *Proc. of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, p. 662–680, 2020.

[64] J. Hyun, N. Van Tu, J.-H. Yoo, and J. W.-K. Hong, "Real-time and fine-grained network monitoring using in-band network telemetry," *International Journal of Network Management*, vol. 29, p. e2080, 2019.

[65] Z. Liu, J. Bi, Y. Zhou, Y. Wang, and Y. Lin, "NetVision: Towards Network Telemetry as a Service," in *Proc. of the IEEE 26th International Conference on Network Protocols*, pp. 247–248, 2018.

[66] J. A. Bondy, *Graph Theory With Applications*. GBR: Elsevier Science Ltd., 1976.

[67] D. Bhamare, A. Kassler, J. Vestin, M. A. Khoshkholghi, and J. Taheri, "IntOpt: In-Band Network Telemetry Optimization for NFV Service Chain Monitoring," in *Proc. of the IEEE International Conference on Communications*, pp. 1–7, 2019.

[68] Jonatas Adilson Marques and Marcelo Caggiani Luizelli and Roberto Irajá Tavares da Costa Filho and Luciano Paschoal Gaspary, "An optimization-based approach for efficient network monitoring using in-band network telemetry," *Journal of Internet Services and Applications*, vol. 10, no. 1, pp. 1–20, 2019.

[69] S. Koranne, "A distributed algorithm for k-way graph partitioning," in *Proc. of the 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*, vol. 2, pp. 446–448, 1999.

[70] L. A. Sanchis, "Multiple-way network partitioning," *IEEE Transactions on Computers*, vol. 38, pp. 62–81, 1989.

[71] Karypis, George and Kumar, Vipin, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[72] P. Gill, N. Jain, and N. Nagappan, "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," in *Proc. of the ACM SIGCOMM*, pp. 350–361, 2011.

[73] INT-Path, "INT-Path repository." `https://github.com/graytower/INT_PATH`. Accessed: May 13, 2020.

[74] L. Gurobi Optimization, "Gurobi Optimizer Reference Manual." `http://www.gurobi.com`, 2021. Accessed:March 2, 2021.

[75] Aric A. Hagberg and Daniel A. Schult and Pieter J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in *Proc. of the 7th Python in Science Conference*, pp. 11 – 15, 2008.

[76] ETH-Zurich, "ETH Zurich P4 Learning Repository." `https://github.com/nsg-ethz/p4-learning`, 2020. Accessed: March 2, 2021.