UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ERNESTO VAZ DE OLIVEIRA

Automation of Traffic Generation and Testing of Programmable Networks using P4 and P4Docker

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Lisandro Zambenedetti Granville
Coadvisor: Dr. Muriel Figueredo Franco

Porto Alegre
Janeiro 2025

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Marcia Barbosa
Vice-Reitor: Prof. Pedro Costa
Pró-Reitora de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do Curso de Ciência de Computação: Prof. Marcelo Walter
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

“As if that blind rage had washed me clean, rid me of hope; for the first time, in that night alive with signs and stars, I opened myself to the gentle indifference of the world. Finding it so much like myself—so like a brother, really—I felt that I had been happy and that I was happy again.”

— Albert Camus

ACKNOWLEDGEMENTS (AGRADECIMENTOS)

Aos meus pais, Iara e Valdeci, cujo trabalho e dedicação me garantiram acesso ao conhecimento e cujo incentivo ao estudo me levou a valorizá-lo desde o princípio.

À faculdade, cuja estrutura me formou como indivíduo e à qual serei para sempre grato pelas experiências adquiridas.

Ao Professor Lisandro Zambenedetti Granville, pela valiosa oportunidade de realizar este trabalho e pelo apoio na definição de seu direcionamento.

Ao Professor Muriel Figueredo Franco, por me acompanhar de perto durante toda essa jornada e cuja dedicação me motivou do começo ao fim.

Agradeço a toda a rede de suporte que adquiri nos últimos anos: à minha namorada, Pietra, que me estendeu todo seu carinho, e aos meus amigos, que sempre estiveram comigo. Por fim, agradeço aos bits e bytes do universo a que me dedico a estudar.

ABSTRACT

Programmable data planes, enabled by languages such as P4 (Programming Protocol-independent Packet Processors), have changed the way network functionality is developed and deployed; however, the lack of straightforward methods for testing and validating P4-based applications poses significant challenges for network developers. This work introduces *P4thTest*, a modular framework for testing automation that provides its own traffic generation and telemetry collection, while also leveraging P4Docker to deploy to a virtualized testing environment using Docker containers. By incorporating In-band Network Telemetry (INT), our solution provides real-time analysis of network behaviors directly from the data plane. We demonstrate our approach in both single and multi-switch topologies, illustrating how INT headers can be seamlessly embedded and removed to preserve original payloads. In addition, *P4thTest* provides an exporter so that measured data can be sent to a time series database and data platform (InfluxDB) demonstrating the potential for real-time visualization and monitoring of the test environment.

Keywords: Programmable Networks. P4. Switch. Computer Networks. In-Band Network Telemetry. Virtual Testbeds. P4Docker.

Automação de Geração de Tráfego e Testes de Redes Programáveis usando P4 e P4Docker

RESUMO

Planos de dados programáveis, cuja evolução foi impulsionada por linguagens de programação como P4 (Programming Protocol-Independent Packet Processors), mudaram a forma com a qual funcionalidades de rede são desenvolvidas e implantadas. Contudo, a falta de métodos simples para testar e validar aplicações baseadas em P4 gera desafios significativos para desenvolvedores de redes. O presente trabalho introduz *P4thTest*, um framework modular para automação de testes que fornece seu próprio gerador de tráfego e coletor de telemetria, além de tirar proveito de P4Docker para implantar um ambiente de testes virtualizado em containers Docker. Ao incorporar In-band Network Telemetry (INT), nossa solução possibilita a análise em tempo real de comportamentos de rede diretamente no plano de dados. Demonstramos nossa abordagem em topologias com um ou vários switches, ilustrando como cabeçalhos INT podem ser facilmente inseridos e removidos para preservar a carga útil dos pacotes. Além disso, *P4thTest* disponibiliza um módulo exportador para que os dados medidos possam ser enviados a um banco de dados de séries temporais e plataforma de dados (InfluxDB), evidenciando o potencial para monitoramento e visualização em tempo real do ambiente de testes

Palavras-chave: Redes Programáveis, P4, Switch, Redes de Computadores, In-Band Network Telemetry, Testbeds Virtuais, P4Docker.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

OS Operating System

DB Database

ONOS Open Network Operating System

TCP Transmission Control Protocol

UDP User Datagram Protocol

JSON JavaScript Object Notation

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

BMv2 behavioral Model version 2

INT In-band Network Telemetry

P4 Programming Protocol-independent Packet Processors

SDN Software-Defined Networking

QoS Quality of Service

VNF Virtual Network Functions

IoT Internet of Things

VM Virtual Machine

GUI Guided User Interface

UI User Interface

CLI Command Line Interface

DoS Denial of Service

PPS Packets per Second

SSH Secure Shell

PID Process Identifier

CPU Central Processing Unit

RAM Random Access Memory

IO Input/Output

KPI Key Performance Indicator

FPGA Field-Programmable Gate Array

CONTENTS

1 INTRODUCTION

In the field of programmable networks (BADOTRA; PANDA, 2020), Programming Protocol-independent Packet Processors (P4) (BOSSHART et al., 2014) is the most common domain-specific language to allow the definition of data plane algorithms that can be run on switches and controlled externally via the control plane. With P4, software developers can fully design and customize the operation of any number of switches they choose to compose a programmable network that is dynamic and fine-tuned to each specific environment.

Since packet processors perform critical network operations, there is a great need to properly test the behavior of this type of software. However, P4 developers usually struggle in various ways when validating and testing their applications. This difficulty of setting up tests and the cost of testing hardware constitute a barrier to obtaining a comprehensible analysis of the behavior of programmable P4 switches.

The closest reproduction of real-life scenarios is achieved through real hardware. However, since we need to set up physical network connections for the topology, this solution demands a lot of knowledge and time to achieve network tests since each node needs to be configured and connected physically. This approach also requires access to costly hardware, a common problem in P4 programming. Devices such as Intel's Tofino (INTEL, 2021) have improved the accessibility of P4 programmable switches but are still costly and thus difficult to obtain for testing. There are also testbeds that can support tests using real hardware, such as the FABRIC (BALDIN et al., 2019) testbed, which is a research infrastructure that enables cutting-edge and exploratory research at-scale for networking.

Additionally, there are approaches for emulation. In this direction, several tools and testbeds have emerged using Mininet (LANTZ; HELLER; MCKEOWN, 2010), a commonly used platform for emulation network applications, to test the behavior of P4 switches in a close enough environment. Although lacking fidelity, these types of approaches are a simpler way to reproduce the tested software.

A novel approach for emulation of programmable networks is the P4Docker tool (SILVA et al., 2024), which has been proposed as a Docker-based tool that provides even greater fidelity than those brought by Mininet and a similar usability since it is capable of simulating the behavior of entire operating systems running inside Docker containers and virtual Ethernet devices. However, developers still need to manually write their own

tests, writing code to be run on each host to assert the communication between them.

For all of these scenarios, developers spend time manually configuring their tools, topologies, and visualization setups, which differ a lot from solution to solution, where assertions of software functionality, like smoke tests, may still require to be automated from scratch. Despite existing network emulators like Mininet for code debugging and tools such as *p4pktgen* (NöTZLI et al., 2018) for generating packets for P4 code, there is a lack of frameworks that seamlessly integrate real-time telemetry with container-based emulation in a user-friendly test environment.

Therefore, in this bachelor thesis, we introduce *P4thTest*, a tool to enable network operators to test and analyze the behavior of P4 programs using simulated realistic scenarios built on top of virtualized infrastructures. Using our tool, developers can test their network programs in general and specific scenarios before deploying them to validate results and guarantee that specific operations can occur consistently. For that, a tool for defining, monitoring, and executing customized tests in programmable networks is provided. The tool uses the P4Docker platform as a testbed to simulate network topologies using Docker containers, thus allowing the creation of customized tests and traffic in a dynamic and controlled way. Users can easily define a testing environment and see it simulated in real-time, along with displaying various performance metrics, including integration with the In-Band Network Telemetry (INT) framework (KIM et al., 2015) and resource usage monitors.

This bachelor thesis provides the following contributions: *(i)* a novel architecture, named *P4thTest*, for automated traffic generation and metric collection; *(ii)* an INT integration module that simplifies collecting real-time telemetry on containerized topologies; *(iii)* a proof-of-concept tool to export and aggregate metrics in InfluxDB, simplifying real-time debugging at scale.

The rest of this document is organized as follows. Section 2 provides academic background on the areas of this research. Section 3 discusses related work on testbeds, simulation and emulation in networking, along with reviewing the state-of the-art in automated testing in P4 programming, and also providing a rundown of network measurement approaches. Next, Section 4 introduces *P4thTest* and provides implementation details along with a workflow description. Section 5 presents an evaluation of the approach presented and its features. Finally, Section 6 concludes the work, discussing its contributions and providing areas of improvement for future work.

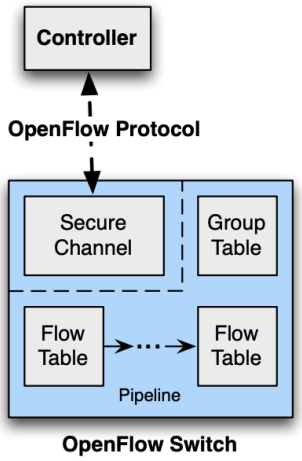
2 BACKGROUND

In this section, we focus on describing concepts and technologies that are essential for fully understanding this work. These include concepts related to Software Defined Networks, the P4 Programming Language, and In-band Network Telemetry.

2.1 Software Defined Networking and OpenFlow

Software-Defined Networking is a paradigm that separates the control plane of the network routers from the data plane by introducing a level of programmability into these routers, usually through the means of an API definition that abstracts communication between the routers and a centralized control plane (KREUTZ et al., 2015). One of the most common APIs for this interaction is OpenFlow (MCKEOWN et al., 2008). It defines behavior using flow-tables, where each programmable node contains a table with entries specifying possible actions. Ingress packets are matched against these entries in order to trigger hardware functions for the switch, such as forwarding or dropping packets.

Figure 2.1: Main components of an OpenFlow switch



Source: (Open Networking Foundation, 2012)

For every packet received in an OpenFlow switch, a packet lookup is performed on the flow table, which contains header values to match against, to define which actions to take. Figure 2.1 showcases a simplified diagram of the OpenFlow architecture, where the controller specifies switch behavior, such as its Flow Tables.The OpenFlow specification defines the header formats and actions that a switch must support in order to be compliant with an OpenFlow version, while also allowing functionality for optional switch features

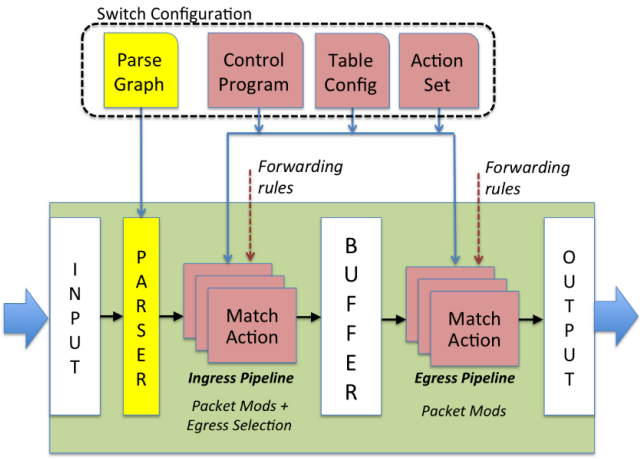
in its API. Only some basic functionality is stated as required, like actions to *drop* and *forward* packets, but among the optional actions are *modify-field* to change switch counters and *enqueue* to manually enqueue the packet in a desired queue.

The separation of functionality between the control and data planes allows that, more easily than conventional networks, SDN implementations can introduce novel features and technologies in them, requiring only the software developmental effort to do so. SDN and OpenFlow have grown in the industry for the benefits of configurability and flexibility that they bring, to the current state that most vendors of commercial switches now include support of the OpenFlow API in their equipment (KREUTZ et al., 2015)

2.2 Programming Protocol-independent Packet Processors

Among existing languages for defining data plane behavior, the P4 language stands out as being able to enable a protocol-independent approach for programming network packet processor switches (BOSSHART et al., 2014) without coupling the switch itself with underlying SDN protocol and its versions, being it OpenFlow or any other. To do this, the language supports a programmable parser, so that headers don't need to be specified by the protocol, while still abstracting hardware actions that are defined by the switch specification, such as the forwarding process. Figure 2.2 showcases P4's forwarding model, where the parser and forwarding rules are defined in the programming language.

Figure 2.2: P4 forwarding model



Source: (BOSSHART et al., 2014)

Using this level of programmability, the packet processors configured to execute P4 code can then be extended to act such as a firewall, inhibiting TCP connections from

specified sources, a load balancer, by changing routes of packets in order to not overflow a host, or even add special headers to packets such as in the In-Band Network Telemetry (INT) specification. Examples such as these can be found at the P4 Tutorials repository (P4 Language organization, 2016b), which programmers frequently use to obtain example topologies and code for their tests.

The reference compiler for P4, called P4C, already supports multiple compilation targets, such as switch software implementations and eBPF compilable C code which can be loaded in the Linux kernel.

2.3 BMV2

A typical tool used to test P4 code is the P4 Behavioral Model (BMV2, for the second version), which is a reference P4 switch created to be a tool for developing, testing, and debugging P4 software (P4 Language organization, 2016a), acting as a switch implementation in C++17 that is able to run P4 compiled code. This switch implementation is not intended for production usage, but is frequently used in order to debug code in varied environments, be it real hardware or not.

In order to facilitate testing, a BMV2 runtime can be executed on top of a Linux environment, such as Debian, to use a general purpose computer in the form of a network switch. This same behavior can be leverage to use BMV2 as the runtime for simulated or emulated P4 switches, and is the common model to do so when using tools such as Mininet.

2.4 Mininet

Mininet (LANTZ; HELLER; MCKEOWN, 2010) ia a tool that enables different types of hosts nodes to be run as computer processes, whose connections are setup by the Mininet controller to simulate network topologies with a variety of switch and host types. The BMV2 runtime can be leveraged in order to simulate a network switch that is able to run real P4 code in this environment.

Since the Mininet runtime is built in Python, the tool leverages its customization possibilities to provide an easy to use API to extend its functionality with code. Network topologies can then be described programmatically using Python, or through a JSON file

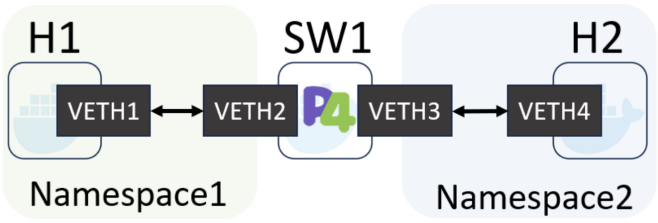
that defines nodes and edges in the topology.

Mininet is able to run real operation system kernels, even general purpose kernels such as Linux, without virtualization. Although this feature brings simplicity, this configuration leads different hosts and switches to share the same resources from the underlying OS, which can invalidate conclusions made on resource analysis and possible switch behavior according to resource usage, such as in stress testing.

2.5 P4Docker

P4Docker (SILVA et al., 2024) is a novel approach for connecting virtualized P4 hosts in Docker containers to a programmable topology. With P4Docker, a more ideal hardware simulation can be achieved through proper isolation of the environment between hosts and switches, allowing developers to use the built-in Docker functions to read the resource metrics for each of the network nodes tested. To do this, P4Docker uses virtual ethernet connections, which are native to the Linux Kernel, to simulate edges in a network environment. With this, each emulated network port can connect to, and only has knowledge of, ports that are directly connected to it in the simulated environment. Figure 2.3 illustrates this configuration in an example with two hosts and a switch.

Figure 2.3: P4 Docker example diagram containing two hosts and a switch

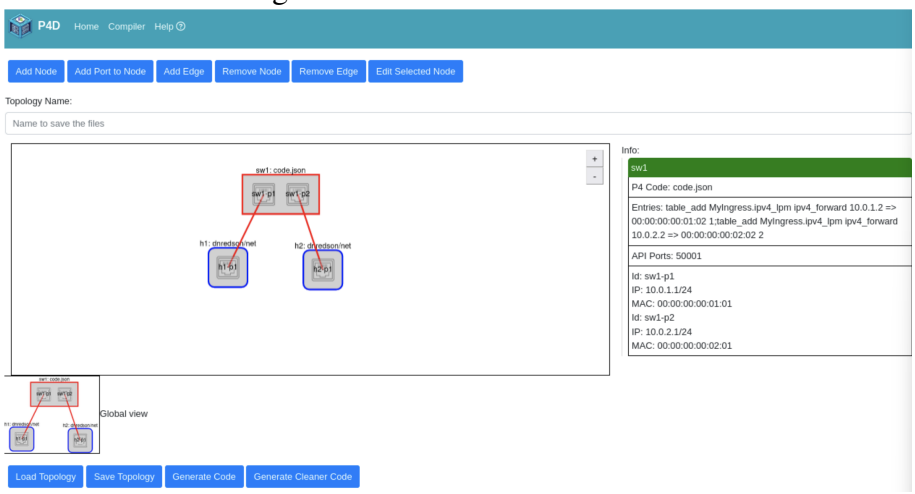


Source: (SILVA et al., 2024)

Network topologies can be defined in a JSON file, just as in Mininet. However, P4Docker also improves the state-of-the-art of development testing of P4 code by providing a dashboard to visually configure complex topologies. A usage example for the P4Docker dashboard is provided in Figure 2.4.

Since the solution uses Docker containers, less configuration is needed in order to simulate tests against P4 code, and since Docker is a common compilation target for code, the same container can be used for running tests. P4Docker also allows for easier debugging of the BMV2 switch, by running it directly in a shell, allowing developers to consult logs in real time while the code is executed.

Figure 2.4: P4 Docker dashboard



Source: (SILVA et al., 2024)

2.6 In-band Network Telemetry

In-band Network Telemetry (INT) is a novel approach for network monitoring brought into spotlight by advancements in SDN concepts and the consolidation of the P4 language (KIM et al., 2015). INT embeds network telemetry data directly into the headers of packets traversing the data plane. With this approach, monitoring information can be stacked on the application packets themselves, without requiring specialized packets. This telemetry data can then be extracted and analyzed at the destination for a complete summary of network conditions between hops. The original INT implementation contained only the switch ID, used to track the path of the packet (through the order in which the headers have been pushed), as well as the time spent in the switch queue to monitor possible switch overflow (KIM et al., 2015).

The P4 Organization provides a written specification for In-band Network Telemetry implementations (The P4.org Applications Working Group, 2020). From this specification, the Table **??** was constructed with the recommendations of metrics to be implemented in INTv2.

Table 2.1: Metrics suggested in INTv2

Metric	**Description**
Node ID	Unique ID for a node. A packet's path can discerned by what the order of the IDs is in a header.
Ingress Interface ID	Identifier for the interface where the packet was received.
Egress Interface ID	Identifier for the interface where the packet was delivered.
Ingress Timestamp	Device local time when the packet was received on the ingress port.
Egress Timestamp	Device local time when the packet was delivered from the egress port.
Hop Latency	Time taken for the packet to be switched within the device.
Ingress Link Utilization	Rate of utilization of the ingress interface the packet was received from.
Egress Link Utilization	Rate of utilization of the egress interface the packet was delivered with.
Queue Occupancy	Build-up of traffic in the queue for the device (in bytes, cells, or packets).
Buffer Occupancy	Build-up of traffic in the egress buffer for the device (in bytes, cells, or packets).

Source: INTv2.1 specification (The P4.org Applications Working Group, 2020)

3 RELATED WORK

In this section, we analyze the existing work around testing frameworks for the P4 language and SDN programmability, along with reviewing the state of the art for network testbeds. In addition, other solutions are presented for approaches for network measurements.

3.1 Testbeds, simulation and emulation in networking

In the field of computer networks, multiple domains require systematically testing network environments, be it for academic research or software and hardware development. Hardware network testbeds provide the most realism in experimentation, but are, however, costly to set up and scale at large. Since these barriers can slow down or even impede computer network development efforts, several solutions have emerged to simulate and emulate computer networks in testing environments.

3.1.1 Network Simulators

Network simulators are platforms focused on providing the most realistic results compared to real hardware, however, to accomplish that given simulation constraints, they tend to focus on specialized aspects, such as network algorithms, protocols, or hardware (BAKNI; CARDINALE; MORENO, 2019). Other uses of network simulators can be found in the educational area, where simulators are used to deliver training and as learning materials in academic environments.

Two commonly used educational tools in this area are Cisco Packet Tracer and Graphical Network Simulator 3 (GNS3). Cisco Packet Tracer (CISCO, 2022) is a discrete-event simulator used in networking courses. It features a simulation environment with a user-friendly graphical interface that facilitates the creation of complex network topologies. Packet Tracer allows for simulation of Cisco devices with a high degree of accuracy and includes other features, such as the ability to simulate IoT concepts. GNS3 (Galaxy Technologies, 2022) is a network simulator that simulates physical devices and can emulate different hardware-specific network operating systems, also providing an interface to operate them.

Network simulation is also typically used for protocol evaluation. These simulators facilitate the testing of network protocols and the observation of their behaviors in a realistic scenario. Network Simulator 3 (ns-3) (RILEY; HENDERSON, 2010) is an open-source tool that provides highly realistic simulations of wired and wireless networks, although most commonly used academically for wireless simulations. Ns-3 is provided as a library that can be used in C++ or Python to define the network topology and the experiments that can be run on it. Objective Modular Network Testbed in C++ (OMNeT++) (OpenSim, 2001) is an open source environment that provides modular tools and visualization components to simplify the testing of network algorithms and protocols at an abstract scale.

3.1.2 Network Emulators

Network emulators use approaches such as virtualization to mimic behaviors and allow real traffic to flow to throw the simulations. This type of simulation allows quick experimentation and prototyping of network solutions. Network emulators may use virtualization-based emulation, virtualizing entire hosts and switches using VMs, or container-based emulation, simplifying the simulation but allowing nodes to share same resources (GOMEZ et al., 2023).

Mininet (LANTZ; HELLER; MCKEOWN, 2010), a commonly used tool for rapid prototyping of networks in software, containerizes its nodes using process-level virtualization, which allows for more scalability than VM-based solutions, while still providing a way to test production-level code (GOMEZ et al., 2023). Mininet provides an interface in Python to describe topologies and define experiments. The solution's process virtualization simplification leads to the emulated environments sharing file system, page tables, and general OS resources with the host system, which may incur in a less realistic simulation. To improve on this, P4Docker (SILVA et al., 2024) was proposed as a way to simulate network topologies, this time focusing on P4 interactions, using Docker containers, bringing a more concise separation of environments while still maintaining resource efficiency (SILVA et al., 2024). P4Docker also provides a easy to use dashboard to edit topologies in a UI.

3.1.3 Network Testbeds

Network testbeds are large-scale testing environments that provide evaluations of network topologies and metrics. These types of tool may or may not use a type of virtualization to achieve results, and the ones that use real hardware may be more costly but provide more realistic results. Emulab (WHITE et al., 2002) is a general purpose testbed that provides a way to use real hardware, virtual machines, and clusters to perform a wide range of experiments. P4Campus (KIM et al., 2021) is a P4-oriented solution which uses programmable switches using real captured campus traffic to test network topologies in a more production-oriented scenario.

Some solutions may use a mix of real hardware and virtualized environments as a way to simplify provisioning resources at large scale, allowing real network distributed tests. FABRIC (Adaptive Programmable Research Infrastructure for Computer Science and Science Applications) (BALDIN et al., 2019) is a novel approach that provides infrastructure distributed across laboratories, campuses, and commercial spaces, providing specialized testbeds for P4 and OpenFlow, wireless networks, and IoT. Using FABRIC, researchers can reserve the usage of resources, which are adaptively provisioned for the tests. Tests are written with collaborative Jupyter Notebooks and a novel UI has been designed to facilitate topologies to be defined.

3.2 Automated testing in P4 programming

Several works have been published in the field of automated testing in P4, which can be separated into solutions using symbolic execution or fuzzing. Symbolic execution is the approach of checking which properties of a program hold for each individual scenario by abstractly representing input values as symbols to test execution paths using a simplified interpreter for the language (BALDONI et al., 2018). Fuzzy testing, on the other hand, randomly tests the outputs of a program by mutating user-provided input in a stochastic manner, which itself may have its flaws in terms of how deep this technique can reach different execution branches, but can also be used in conjunction with symbolic execution to provide better results (BALDONI et al., 2018).

Although several works have used symbolic execution to automate testing for control plane code, the first publication to achieve this for the P4 language is p4pktgen (NöTZLI et al., 2018), a tool that generates packets that conform to the code used as input to

validate each condition exposed in the code. P4pktgen focuses on the packet processing pipeline by consuming the JSON compiled P4 code and producing test cases in the form of packets and table configurations, which are expected to result in a successful execution by the P4 runtime. With the test cases defined by p4pktgen, bugs in the compiler, runtime, or switch implementation can be found when unexpected errors occur while executing the inputs. In the original publication, the authors of p4pktgen used the solution to find four bugs in p4c, the reference compiler for P4 16 (P4 Language Organization, 2021).

Other works have expanded the same concept afterwards. BB-Gen (RODRIGUEZ et al., 2018) uses a similar methodology to create packets that conform to P4 code tables, but also bringing a real-time visualization component to the tool. P4testgen (RUFFY et al., 2023) also improves on the concept by creating control plane entries for the generated code, while also being more versatile in covering a greater variety of compilation targets by being an extension of the P4C compiler. SwitchV (ALBAB et al., 2022) combines both fuzzing and symbolic testing to generate sequences of valid and invalid P4 programs and table entries to find bugs in the entire network stack surrounding the P4 implementation, including the P4 code itself. The SwitchV authors found 154 bugs across different switch layers. Finally, (SAUERESSIG et al., 2024) defined an approach for the collection of performance metrics for behavior fingerprinting of P4 switches.

3.3 Network measurement and telemetry-based approaches

The field of network measurement can be divided into traditional measurement approaches and solutions based on network telemetry (TAN et al., 2021). The scenario of traditional network measurement contains well known and used tools such as IPerf (DUGAN et al.,) and Traceroute (JACOBSON,) which similarly proactively send payload packets from an "observer" perspective to obtain an analysis of network condition and metrics from an specific point of the topology. Other traditional measurement approaches may achieve similar results from a passive standpoint, tools like the Netflow (CISCO,) protocol allow routers that implement it to routinely send results of recent traffic (separated into concisely defined "flows") to collectors that can analyze the data sent.

Although these tools are heavily used for their directness and simplicity of implementation, these tools stand short, however, in terms of using the environment they are monitoring and thus modifying it, which brings a lack of precision in such a field that is

heavily bounded by resource usage in measurements (ROUGHAN, 2005).

Newer developments in the field of network measurement defined the term network telemetry to conceptualize approaches that automate the collection and processing of network information in a systematic way that minimizes the limitations of traditional measurements and brings more scalability and visibility (TAN et al., 2021). Active network telemetry encompasses solutions that actively construct telemetric paths for automated measurement packets to fill in visibility areas of the network, such as Everflow (ZHU et al., 2015). In-Band Network Telemetry (INT) (KIM et al., 2015) is a novel approach that uses business packets to embed telemetric information into the system, thus less heavily affecting the measured system. Since its proposal, INT has rapidly gained traction in network measurement research and has been incorporated into different solutions to leverage its potential.

INTCollector (TU et al., 2018) has been proposed as a method to export telemetric data from INT switches to an external collector that processes data in a more efficient way and allows storage to be leveraged using InfluxDB (DIX, 2013). Prometheus INT exporter (PROMETHEUS. . . ,) is another solution in this regard, allowing it to leverage a Prometheus (SOUNDCLOUD, 2012) instance to send telemetric data and process information. IntMon (TU; HYUN; HONG, 2017) is part of an ONOS-based (BERDE et al., 2014) solution that provides a controller that configures INT switches via their flow tables and allows collected metrics to be processed at an ONOS core. All of these solutions bring different approaches to collecting and processing INT telemetry at scale, but do not consider the testing phases of network management and SDN development.

4 APPROACH

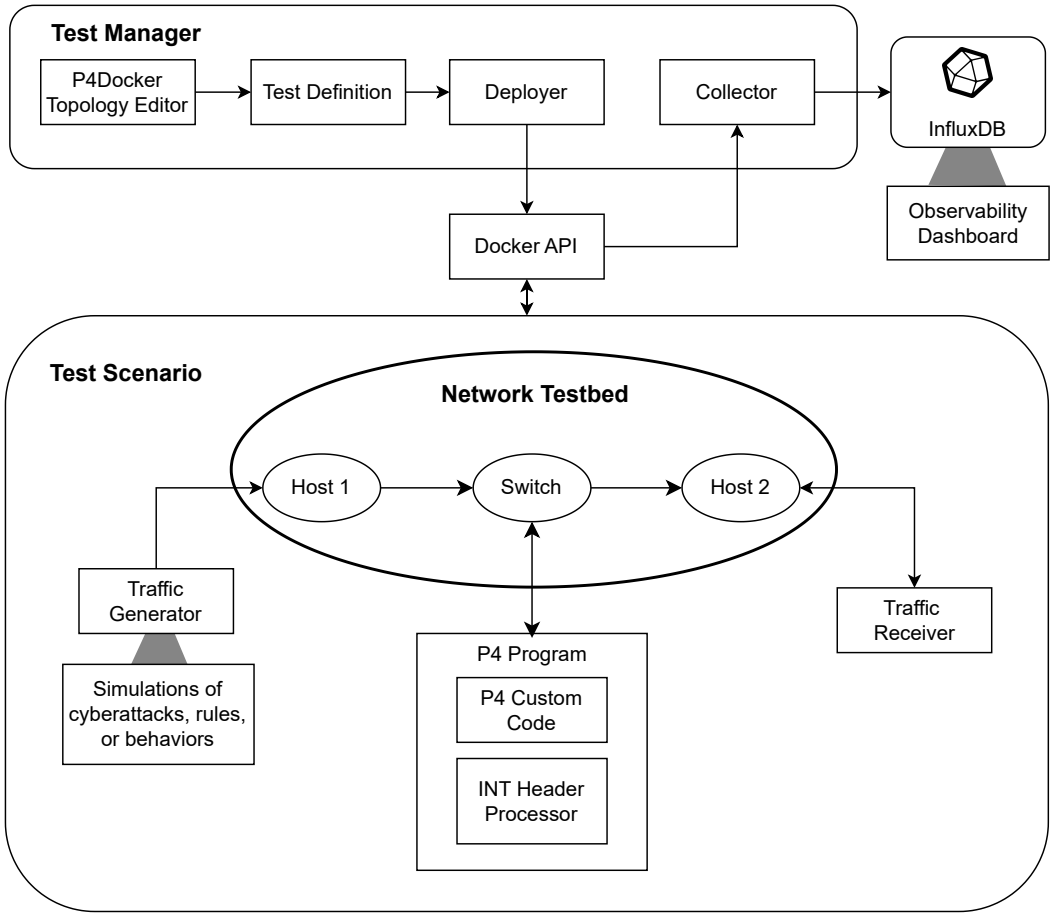
In this section, we detail the methodology and architecture of our solution, *P4thTest*. The proposed framework utilizes INT and the programmable foundation of P4 Docker to provide a modular approach for testing the functionality of P4 switches and code in a virtualized setting. Using our approach, developers can use one of the template topologies or customize one using the P4 Docker tool to then test their applications in a virtualized setting and get metrics and traces of the results.

4.1 Architecture

The architecture for *P4thTest* is presented in Figure 4.1. We define a modular approach by implementing standalone features that together provide an automated testing approach for P4 applications. This includes a Traffic Generator for generating INT test packets and fragmenting payloads, containerized Monitor Switches containing INT Header Processors for embedding INT data and forwarding packets, Traffic Receivers for extracting and removing INT metrics from payloads, and a Collector module to aggregate data, also acting as an exporter for InfluxDB.

For storing the collector results over time and displaying them, the implemented traffic collectors can output their results to an InfluxDB instance that can be run locally. This has been used to test the effectiveness of the approach in integrating in a commonly used time series database.

Figure 4.1: *P4thTest* Architecture



4.2 Usage Workflow

This section contains the recommended usage workflow for applying the proposed functions of *P4thTest*. First, the developer prepares their P4 Code for testing. If the desired outcome is to embed INT information on the same router as the tested code, the developer needs to then add the template insertions found in the solution repository. Afterwards, the user compiles the code for the BMv2 target, a common compilation target in P4 compilers such as p4c (P4 Language Organization, 2021).

Next, the developer defines the topology to be tested by opening the P4Docker GUI and either creating a topology from scratch or modifying one of the provided example templates. It is also possible to edit the topology file manually for added control. After adding hosts switches and links, this will create a topology file that can be used with *P4thTest*, and scripts to deploy or remove the environment. This functionality is explained in detail in Section 4.3

After deploying the environment on their machine, the user defines test parameters and arguments for using the provided CLI. Section 4.4 explains CLI usage and parameters. An example of testing arguments is defined in Listing 4.1 using a topology file named *topology.json*, hosts *host1* and *host2*, a 1MB payload and a speed of 100pps.

Listing 4.1: Test definition example using the CLI

```
$ ./test.py                          \
    --file topology.json             \
    --source host1 --target host2    \
    --payload_size 1 --unit MB --speed 100
```

Then, the user is ready to run the testing script and obtain the collected data. By default, this outputs JSON-structured telemetry data that allows for further analysis, as described in Section 4.7. If the end goal is to run assertions on single test results, the workflow ends here.

Users who wish to store their measurements in a time series database can rely on the InfluxDB exporter by running the separate *test-influx.py* script. Section 4.8 explains how the exporter functions. When this exporter is enabled, developers are able to set up observability dashboards in InfluxDB using the gathered data, as demonstrated in Section 5.4 of Chapter 5

In summary, the workflow begins with the preparation of a P4 program, followed by the definition of a containerized topology with P4Docker. Subsequently, the user employs *P4thTest* to generate traffic, collect INT metrics, and optionally visualize performance in real time via InfluxDB. This modular arrangement simplifies replication or expansion of the experiments for diverse scenarios, thereby streamlining iterative development for P4-based applications.

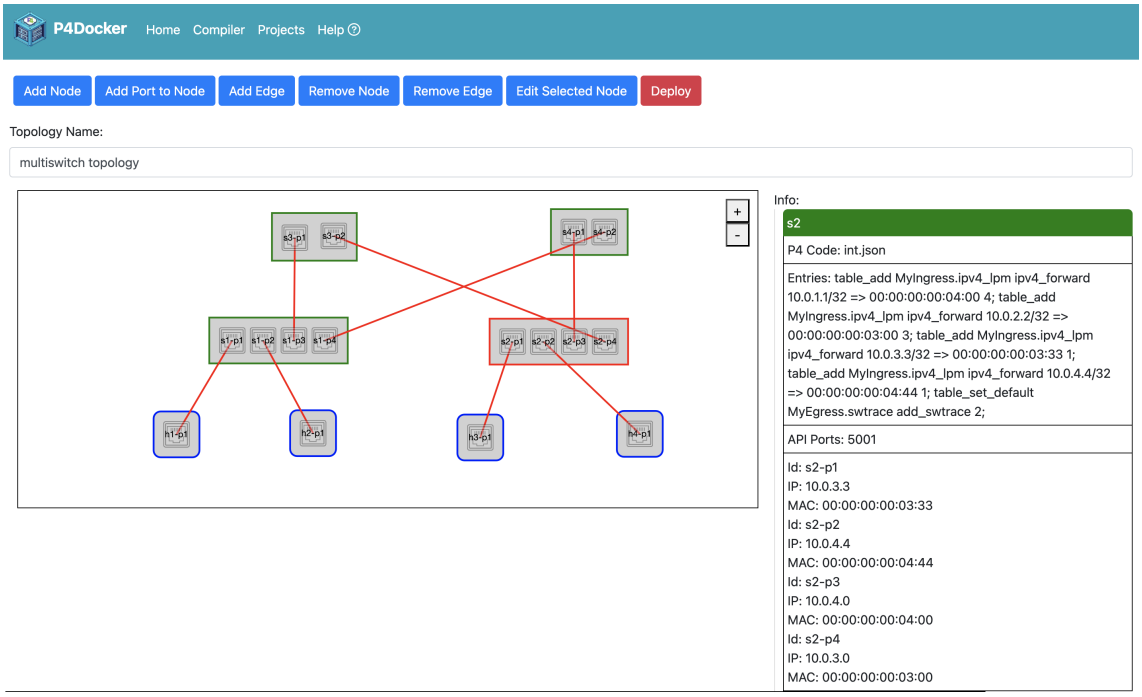
4.3 P4Docker Topology Editor

The Topology Editor is implemented by P4Docker's (SILVA et al., 2024) native editor that allows for connecting network nodes through virtual ports by specifying their internal IP's and virtual MAC addresses. This topology is saved into a JSON file, which can be manually edited or read from, and then converted into scripts that instantiate the virtual infrastructure.

In our implementation, two topology configurations were created and made available in the project repository. The created configurations can serve as base topologies to

test P4 applications on or can be used as templates for enhanced and more complex custom topologies. A more detailed explanation of the topologies implemented and provided will be provided in Section 5. Figure 4.2 demonstrates how the dashboard was used to produce a more complex topology.

Figure 4.2: P4Docker topology editor displaying a 4 switch topology



4.4 Test Definition and Traffic Generator

Given a JSON-format topology definition, the CLI implemented in *P4thTest* reads from the topology the possible hosts and ports and is able to operate on a tuple of *Sender* and *Receiver* for testing. In this definition, for any inserted topologies, a path will be tested between the sender and receiver, relying on the IP protocol between nodes to handle routing. Using the CLI application format, complex tests can be scripted together, allowing the sender and receiver hosts to be exchanged and different network paths to be tested. The tester CLI expects the following test definition:

Listing 4.2: Input arguments for CLI interaction

```
$ ./test.py
usage: test.py [-h] [-f FILE] [-s SPEED] [-m MTU] [-p PAYLOAD_SIZE]
[-u {B,KB,MB,GB}] [-v] SOURCE TARGET
```

A detailed explanation on the properties of the test definition arguments is provided in Table 4.1. The arguments received on the CLI will be utilized as Traffic Generator parameters in the test scenario.

The implemented traffic generator constructs packets with a randomized string payload and configures the packet's INT header with a node counter of zero. When any of the monitors fills in a new header, the node counter field is incremented. This process is done in order to be able to completely reconstruct the payload afterwards by removing the inserted headers.

Table 4.1: Test Definition Properties

Argument	**Description**
Source	Source host name for sending the probe, a host's name is defined in the P4 Docker topology file.
Target	Target host name for sending the probe, a host's name is defined in the P4 Docker topology file.
Speed	Amount of fragments sent per second.
MTU	Maximum transmission unit for fragment payload, in bytes (defaults to 1024).
Payload Size	Size of complete randomized payload, default unit is bytes (if payload size is larger than MTU, payload will used fragments of maximum size).
Payload Size Unit	Unit for payload size, one of B, KB, MB, or GB.
Verbose	Optional verbose flag for whether packet content should be displayed on screen.

4.5 Deployer and Docker API

The Deployer interface abstracts communication between the test manager environment and the test scenario via the Docker API. The P4Docker tool sets up provisioning scripts for each of the hosts and switches, and utilizes the configured node names for the Docker environment for naming the containers. Using these same environment variables, the Deployer interface connects via SSH to each container to send commands.

A first connection is set up with the *Receiver* container, where the *Traffic Receiver* application is run. A second connection is set up with the *Sender* container, where the Traffic Generator arguments are sent to. The established connections are then used by

the Collector interface to listen for events and capture then on both of the sides of the testing path. Although separating the pair of containers by Receiver and Sender, the implementation still allows container to be used interchangeably in any of the two.

4.6 In-band Network Telemetry Metrics Definition

Each monitor is configured to embed switch metrics into packets and forward them according to its pre-configured IP table. The INT implementation used for the monitors comes from open P4 examples from a laboratory on INT from Github (ALMEIDA, 2021). The metrics collected for the implementation are defined at Table 4.2.

Different monitor behaviors are provided through different implementations of the P4 code for the switches. The default behavior is to expect INT-formatted packets and forward input according to its routing table. This default behavior is ideal for the environment where the monitor switches are strategically placed in points where network behavior is to be monitored, such as next to switches of interest.

Another possibility is that of running the Monitor code in the same switch as the tested code, such as in the example demonstrated on Figure 4.1 for the Architecture.

4.7 Metric Collector

The Metric Collector interface abstracts the process of reading the output of the testing environment (emitted by *stdout* of the SSH communication between the Test Manager and the test containers).

For obtaining the telemetric information, a text parser reconstructs the header information extracted from the packets. The payload in the sent packet from the sender host is then compared to the payload received in the receiver to check for discrepancies.

In order to obtain usage metrics for the containers, the Docker API is also used to collect several resource metrics for the running containers, including for each of the switches, with this being one of the advantages of using the Docker infrastructure in the tests. The exposed resource information for containers are: the number of processes running in the container, network IO usage, memory usage, and CPU usage.

The information is aggregated into a JSON structured output in the following format:

Table 4.2: Metrics collected using INT

Metric	**Length** (bits)	**Description**
Switch ID	31	Unique ID for the Switch. On the testing environment, integers starting from 1 were used.
Ingress Port	9	Number of the interface where the packet was received at the switch.
Egress Port	9	Number of the interface where the packet was sent from the switch.
Ingress Timestamp	48	Timestamp of the moment the packet arrived at the switch.
Egress Timestamp	32	Timestamp of the moment the packet was sent from the switch.
Enqueue Timestamp	32	Timestamp of the moment the packet was enqueued on the switch.
Enqueue Depth	19	Build-up of traffic in the ingress buffer of the device, in packets.
Dequeue Time	19	Time taken for the packet to be buffered in the egress buffer, in milliseconds.
Dequeue Depth	19	Build-up of traffic in the egress buffer of the device, in packets.

Listing 4.3: Example of collected data

```
{
  "errors": [],
  "source": "host1",
  "target": "host2",
  "speed": 5,
  "mtu": 1024,
  "payload_size": 128,
  "unit": "B",
  "received_packet": {
    // INT headers are returned here
  },
  "duration": 2.940906047821045,
  "container_metrics": {
    "sw1": {
      "processes": "14",
      "network_io": "0B / 0B",
```

```
      "memory_usage": "0.27%",
      "cpu_usage": "1.42%"
    },
  }
}
```

4.8 InfluxDB Exporter

In order to simplify storing and visualizing data into observability dashboards, an InfluxDB exporter was implemented in *P4thTest*, allowing each of the output structures to be sent individually as time series entries in the database. To establish the connection with a database instance, the exporter utilizes client credentials supplied through environment variables.

InfluxDB was chosen as the time series data platform to be leveraged in this solution as it surpasses the features expected from a database and also provides a user dashboard for the interactions, being comparable to monitoring systems such as Prometheus, but being lighter and more suitable for simpler time series functionality such as this approach entices. Another advantage to using InfluxDB is that it allows for a simpler exporting, due to its implementation of a schemaless design concept (INFLUXDATA, 2024), in which data can be sent over without requiring major formatting or configuration from the recipient side.

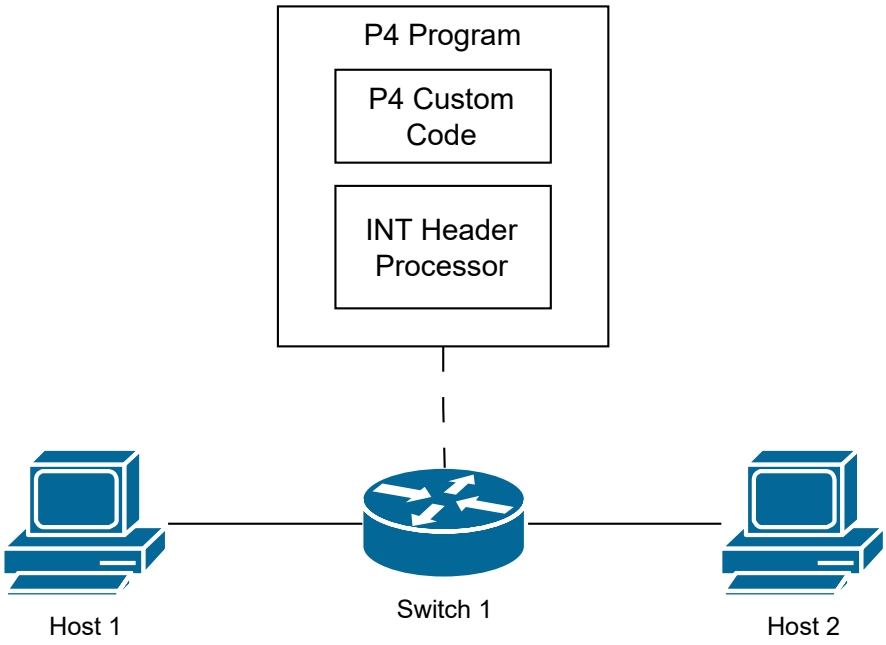
5 EVALUATION

In this section, we present the evaluations performed in order to test the usability and feasibility of our approach to simplify P4 network development and engineering tasks. To accomplish this, test topologies were created using the P4Docker framework to simulate daily development usage. In addition, test scripts were written to exemplify the features the provided solution provides. Lastly, an InfluxDB dashboard was configured in order to validate displaying exported results in real time.

5.1 Basic Test Topology

The first ideal topology should be the minimum testable configuration for the aforementioned approach. To do that, the first example from the P4Docker repository was modified to contain proper edge connections, and an INT header processor was added to the sole P4 switch. This highlights the first utilization procedure for *P4thTest*, which is done by updating the tested switch code with INT processing being inserted at the beginning of the parser stages.

Figure 5.1: Basic Topology



This represented an insertion of approximately 80 lines of code, but no deletions, since the P4 processing pipeline allows for a linear processing approach, where stages

are ideally independent of each other. The needed modifications can be considered as simple steps, since they were based on adding header definitions at the declarations stage of the P4 code, adding a sole egress action for embedding switch information, and adding a deparser step for the new headers. These steps should not change between codebases and are a static process.

The configured topology is represented in Figure 5.1 and made available in *topologies/basic* on the project repository (OLIVEIRA, 2024).

5.2 Multi Switch Test Topology

In order to analyze the feasibility of using the approach for more complex networks, for the second test, a topology was created from scratch. Using the P4Docker interface, a test example from the p4lang tutorial repository has been reconstructed in the new topology system. The process served as a proof of concept for the simplicity of the P4Docker solution as a means of prototyping and setting up test environments.

The topology of choice was the Firewall Example (P4 Language Organization, 2019), where four switches are set up in a grid-like system, one of them working as a firewall maintaining an internal network composed of two hosts, with two other hosts connected to the mesh without a firewall. This showcased a different approach for implementing the testing environment, where the three forwarder switches can be replaced with INT forwarder switches and the firewall can be left as-is, in order to serve as target switch for testing.

The configured topology is represented in Figure 5.2 and made available in *topologies/firewall* on the project repository (OLIVEIRA, 2024).

5.3 Validating results of custom applications

This experiment helped to diagnose the issue in requiring the usage of the traffic generator and collectors for creating the INT formatted packets, since this makes it difficult to test applications that require receiving clean payloads. To solve that, a feature was implemented to forward the payloads received from the metric collector to a configurable port on the loopback device of the host, allowing for a workflow as represented in Figure 5.3.

Figure 5.2: Multi Switch Topology

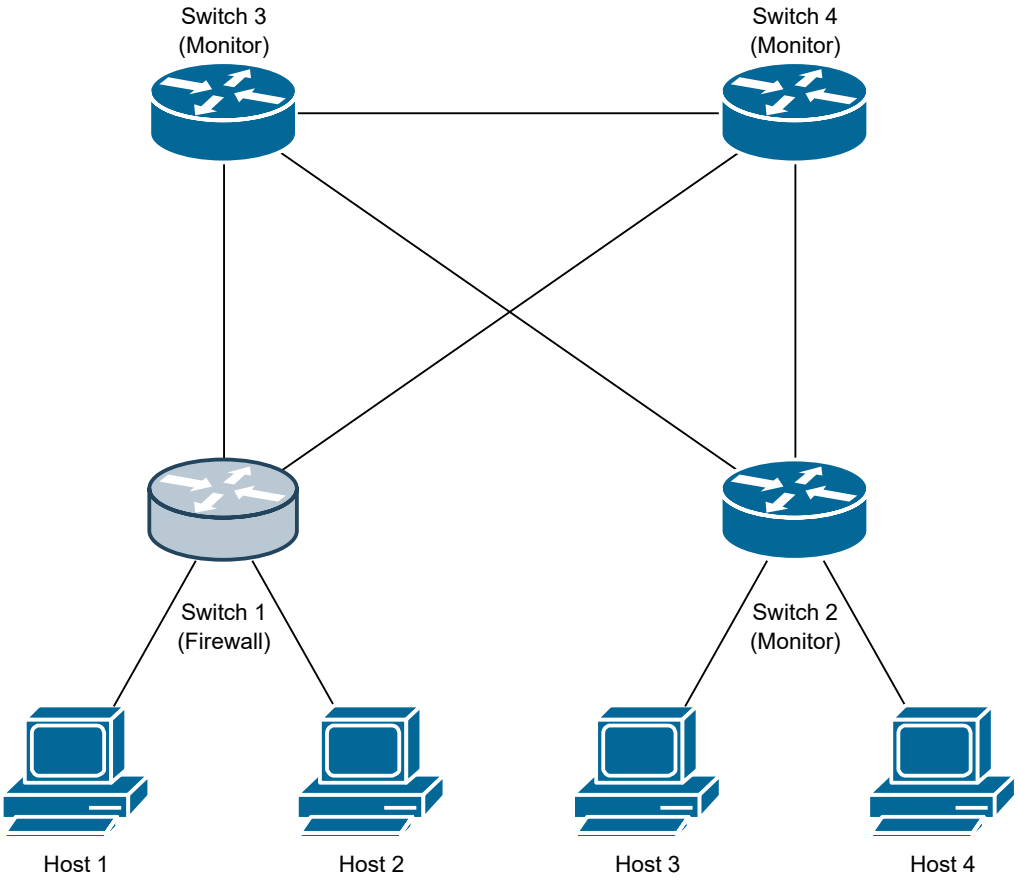
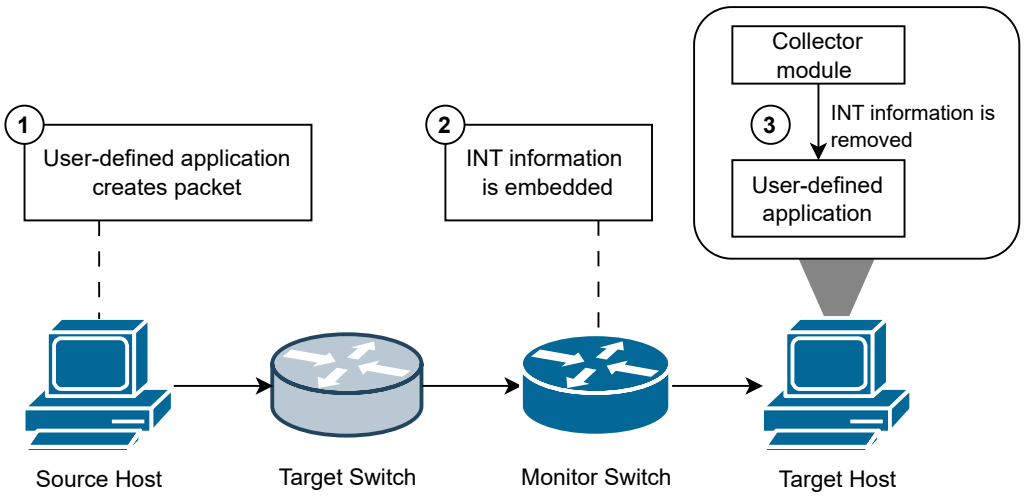


Figure 5.3: Workflow for removing telemetry data at destination



This allowed for creating a test configuration without any modifications in the P4 code, requiring only manipulations on the topology itself to replace monitor switches in the environment, while still maintaining the behavior from the original experiment.

5.4 Time Series Visualization

To validate the approach's InfluxDB exporter, a template dashboard has been created and is pictured in Figure 5.4. In the example, the target switch's CPU and memory usages are displayed in percentages, along with test duration and the switch's internal dequeue time. The created dashboard showcases metrics for a single switch, so a method for filtering the data from a single container was used. To simulate frequent usage of the environment, tests were executed in a loop for a time with the default testing parameters.

This highlights the possibility of using such approach into continuous integration pipelines of P4 Code, to allow for continuous testing of developed code in a virtualized condition, with the usage of collected metrics as predefined KPIs

Figure 5.4: Created visualization dashboard

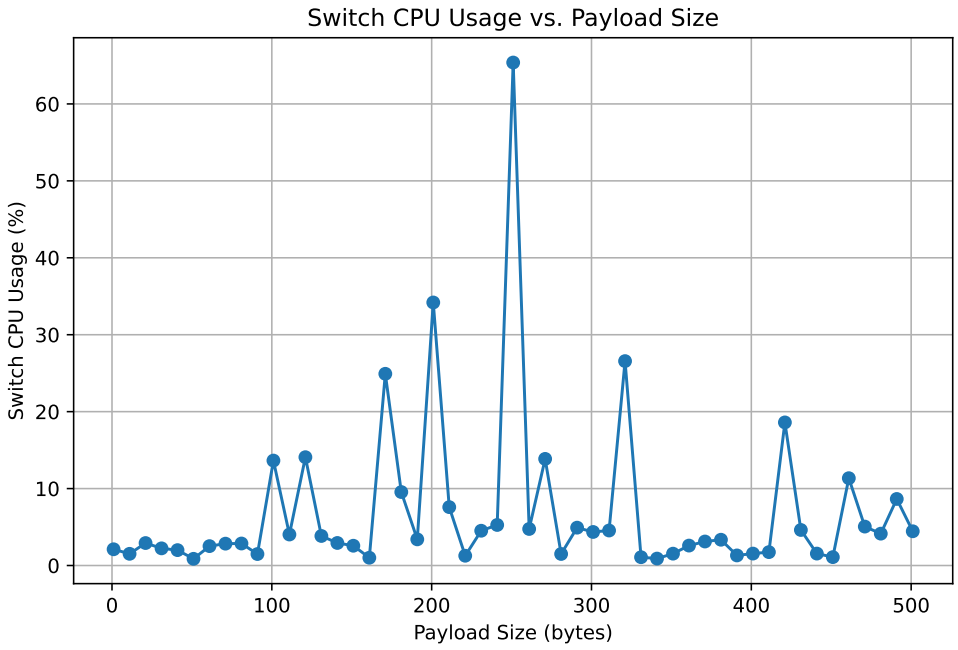
InfluxDB proved itself capable of being used in a prototyping setting, with its installation being simplified by simply instantiating a Docker container in the local machine and its resource usage was not noticeable under normal development settings. The schema-less design provided by the tool proved itself useful to allow for easier prototyping, not requiring schema alterations whenever a new field would be inserted.

5.5 Metrics Validation

In order to review the usefulness of the metrics considered in the proposed setting, tests were performed using the *P4thTest* by varying the parameters of the traffic generator in the two defined topologies, while using the visualization dashboards to aggregate data in a longer time period. Lastly, the single switch topology was then used to create graphs of measured scenarios to illustrate behaviors in data.

While most of the collected metrics proved useful, some resource indicators did not yield meaningful results. For example, Docker API's network input/output metric consistently showed zero throughput in all tests, which may indicate a limitation or bug in P4Docker's network emulation. Nevertheless, since the traffic generator's throughput can be set directly, the lack of network I/O metrics does not impede experiments.

Figure 5.5: Switch CPU usage in according to traffic payload size



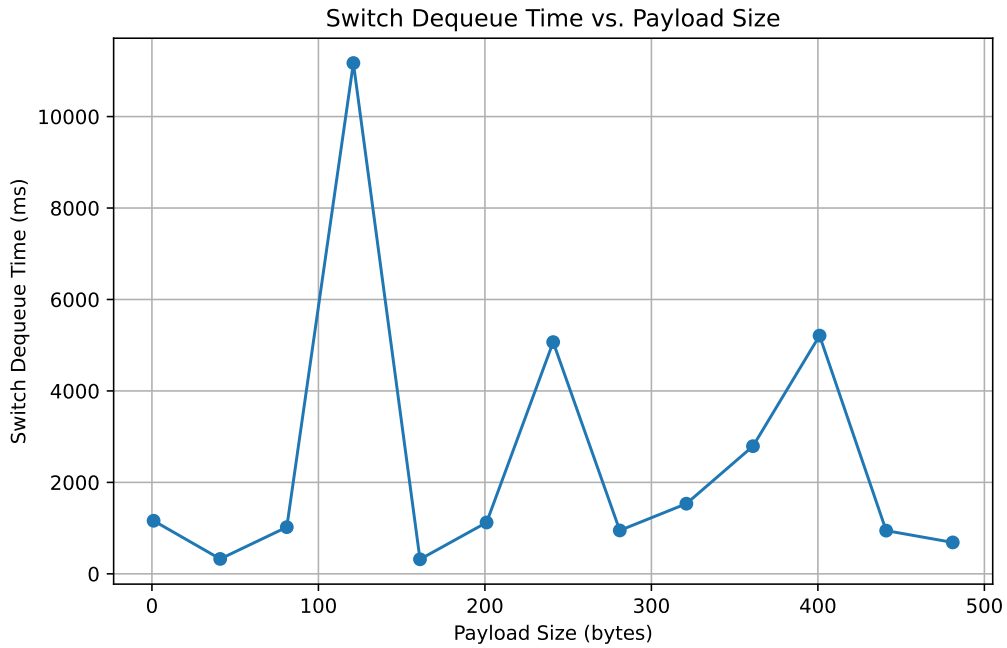
The generated traffic caused no noticeable changes in the switches' allocated memory, which remained near 0.3%. Although inconclusive, this might suggest that the tested P4 code was neither complex nor parallelized enough to demand increased memory allocation. Conversely, the CPU usage for the virtualized switches did increase in proportion to the network load, as shown in Figure 5.5

For Docker API metrics collected, generalized container benchmarks have already been made (N et al., 2015), and other works have investigated resource usage of P4 appli-

cations with different virtualization approaches (SAUERESSIG et al., 2024). This leaves space for further study to be done benchmarking P4 applications specifically within containerized environments such as P4Docker.

INT metrics (e.g., queueing times, queue sizes) did not demonstrate conclusive correlations within the small sample sizes tested, and primarily presented fluctuations, as shown in Figure 5.6. This could also be related to the relative simplicity of the P4 programs tested, or from insufficiently stressed network conditions. Larger-scale experiments or more complex P4 code might better reveal meaningful patterns

Figure 5.6: Time taken to dequeue packets in according to traffic payload size



Overall, these experiments highlight a number of avenues for future investigation. Aside from Docker API's network I/O metric, the collected data show promise for deeper analysis. Although the results for INT metrics were not definitive, their availability, along with collected switch IDs, enables identification of potential bottlenecks in a multi-switch topology, as initially demonstrated in the INT framework experiments (KIM et al., 2015).

6 CONCLUSION AND FINAL REMARKS

In this bachelor thesis, we introduced and evaluated *P4thTest*, an automated testing framework that integrates INT capabilities and containerized topologies for P4 applications. Using a modular architecture and leveraging the usability of P4Docker, our approach simplifies the process of designing, deploying, and testing P4 programs on-scale. The contributions are summarized as follows.

- **INT-Enabled Telemetry Embedding:** The approach allows test topologies to embed and extract telemetry data for measuring performance metrics, such as timestamps, ingress/egress ports, and queue depths. This allows to analyze network delays and bottlenecks for both simple and more complex multi-switch deployments.
- **Collector and Visualization:** Our solution incorporates a flexible metrics collector that gathers INT data and container resource usage in real-time. By exporting results to InfluxDB, it becomes easy to visualize network and container-level performance insights on custom dashboards.
- **Modular Architecture:** The proposed architecture separates core functionalities (Traffic Generation, INT Header Processing, Metric Collection, and Exporting) into independent, reusable components. This modular design lowers the complexity of debugging and fosters rapid prototyping, allowing for enhancements to also be done modularly to each component.

Simplifying daily development tasks is essential in order to approximate the fields of software defined networking and software engineering. Through the design, implementation, and evaluation of *P4thTest*, we demonstrated how INT-enabled telemetry and containerized automation can streamline the development process for P4-based applications. We envision that continued enhancements to *P4thTest*, P4Docker, and related solutions can further empower researchers and developers to innovate in P4-based networks and accelerate the adoption of programmable data planes in production environments.

As limitations, the proposed solution is not as easily integrated as simply adopting it in a project, some topology alterations and fine-tuning of configurations is still required to set up a project for testing. This indicates that further work should still be done in order to simplify the configuration and adoption of the tooling.

The proof-of-concept templates primarily used publicly available P4 tutorials and labs. Broader integration tests comparing against various P4 targets (e.g., Tofino, FPGA-

based P4) would further validate the generality of the approach.

As future work, we envision improvement of different elements of *P4thTest*, for example: *(i)* enhancing the usability of the testing environment by simplifying the addition of monitors into topologies, whether by integrating the solution into P4Docker or providing code templates to not require code alteration to embed INT headers in some cases. *(ii)* creating or adapting a framework to define and assert the test formats specified in this approach, removing the need for developers to script their own tests. *(iii)* providing comparisons with P4 hardware targets to validate the precision of the simulations. *(iv)* incorporating the proposed test environments into continuous integration pipelines of real P4-based projects and reviewing the advantages and caveats of using this in a development pipeline.

REFERENCES

ALBAB, K. D. et al. SwitchV: automated SDN switch validation with P4 models. In: **Proceedings of the ACM SIGCOMM 2022 Conference**. [S.l.]: ACM, 2022. p. 365–379.

ALMEIDA, L. C. de. **InbandNetworkTelemetry-P4**. 2021. <https://github.com/leandrocalmeida/InbandNetworkTelemetry-P4>.

BADOTRA, S.; PANDA, S. N. Software-defined networking: A novel approach to networks. In: **Handbook of Computer Networks and Cyber Security: Principles and Paradigms**. Cham: [s.n.], 2020. p. 313–339.

BAKNI, M.; CARDINALE, Y.; MORENO, L. M. Experiences on Evaluating Network Simulators: A Methodological Approach. **Journal of Communications**, p. 866–875, 2019.

BALDIN, I. et al. Fabric: A national-scale programmable experimental network infrastructure. **IEEE Internet Computing**, v. 23, n. 6, p. 38–47, 2019.

BALDONI, R. et al. A survey of symbolic execution techniques. **ACM Comput. Surv.**, ACM, v. 51, n. 3, 2018.

BERDE, P. et al. Onos: towards an open, distributed sdn os. In: **Proceedings of the Third Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: Association for Computing Machinery, 2014. p. 1–6. Available from Internet: <https://doi.org/10.1145/2620728.2620744>.

BOSSHART, P. et al. P4: programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul 2014.

CISCO. **Netflow**. <https://www.cisco.com/c/pt_br/tech/quality-of-service-qos/netflow>.

CISCO. **Cisco Packet Tracer**. 2022. <https://www.netacad.com/cisco-packet-tracer>.

DIX, P. **InfluxDB**. 2013. <https://github.com/influxdata/influxdb>.

DUGAN, J. et al. **iPerf - The ultimate speed test tool for TCP, UDP and SCTP**. <https://iperf.fr/>.

Galaxy Technologies, L. **Graphical Network Simulator (GNS3)**. 2022. <https://www.gns3.com/>.

GOMEZ, J. et al. A survey on network simulators, emulators, and testbeds used for research and education. **Computer Networks**, v. 237, p. 110054, dec. 2023.

INFLUXDATA. **InfluxDB design principles**. 2024. <https://docs.influxdata.com/influxdb/cloud/reference/key-concepts/design-principles>.

INTEL. **Intel Tofino. P4-programmable Ethernet switch ASIC that delivers better performance at lower power**. 2021. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html>.

JACOBSON, V. **traceroute**. <https://linux.die.net/man/8/traceroute>.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: **ACM SIGCOMM**. [S.l.: s.n.], 2015. v. 15, p. 1–2.

KIM, H. et al. Experience-driven research on programmable networks. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 51, n. 1, p. 10–17, 2021.

KREUTZ, D. et al. Software-Defined Networking: A Comprehensive Survey. **Proceedings of the IEEE**, IEEE, v. 103, p. 14–76, 2015.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In: **9th ACM SIGCOMM Workshop on Hot Topics in Networks**. Monterey, California: [s.n.], 2010. p. 1–6.

MCKEOWN, N. et al. OpenFlow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, Association for Computing Machinery, v. 38, p. 69–74, 2008.

N, P. E. et al. Evaluation of docker containers based on hardware utilization. In: **2015 International Conference on Control Communication Computing India (ICCC)**. [S.l.: s.n.], 2015. p. 697–700.

NöTZLI, A. et al. p4pktgen: Automated Test Case Generation for P4 Programs. In: **Proceedings of the Symposium on SDN Research**. [S.l.]: ACM, 2018. p. 1–7.

OLIVEIRA, E. V. de. **p4thtest**. 2024. <https://github.com/ernestovaz/p4thtest>.

Open Networking Foundation. **OpenFlow Switch Specification**. 2012.

OpenSim, L. **Objective Modular Network Testbed in C++ (OMNeT++)**. 2001. <https://omnetpp.org>.

P4 Language organization. **P4Lang Behavioral Model**. 2016. <https://github.com/p4lang/bmv2>.

P4 Language organization. **P4Lang Turorials**. 2016. <https://github.com/p4lang/tutorials>.

P4 Language Organization. **Implementing A Basic Stateful Firewall**. 2019. <https://github.com/p4lang/tutorials/tree/master/exercises/firewall>.

P4 Language Organization. **P4 reference compiler**. 2021. <https://github.com/p4lang/p4c>.

PROMETHEUS INT(INBAND NETWORK TELEMETRY) EXPORTER. <https://github.com/serkantul/prometheus_int_exporter>.

RILEY, G. F.; HENDERSON, T. R. The ns-3 network simulator. In: **Modeling and tools for network simulation**. [S.l.]: Springer, 2010. p. 15–34.

RODRIGUEZ, F. et al. BB-Gen: A Packet Crafter for P4 Target Evaluation. In: **Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos**. ACM, 2018. p. 111–113. Available from Internet: <https://dl.acm.org/doi/10.1145/3234200.3234229>.

ROUGHAN, M. Fundamental bounds on the accuracy of network performance measurements. **SIGMETRICS Perform. Eval. Rev.**, Association for Computing Machinery, v. 33, n. 1, p. 253–264, jun. 2005.

RUFFY, F. et al. P4Testgen: An Extensible Test Oracle For P4. In: **Proceedings of the ACM SIGCOMM 2023 Conference**. [S.l.]: ACM, 2023. p. 136–151.

SAUERESSIG, M. et al. FEVER: Intelligent Behavioral Fingerprinting for Anomaly Detection in P4-based Programmable Networks. In: **38th International Conference on Advanced Information Networking and Applications (AINA-2024)**. Kitakyushu, Japan: [s.n.], 2024. p. 1–12.

SILVA, D. et al. P4Docker: Enabling Efficient P4 Switch Testbeds with Docker Integration. In: **Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)**. [S.l.]: SBC, 2024. p. 1–8. ISSN: 2177-9384.

SOUNDCLOUD. **Prometheus**. 2012. <https://github.com/prometheus/prometheus>.

TAN, L. et al. In-band Network Telemetry: A Survey. **Computer Networks**, Elsevier, v. 186, p. 107763, 2021.

The P4.org Applications Working Group. **In-band Network Telemetry (INT) Dataplane Specification - Version 2.1**. 2020. Available from Internet: <p4.org/p4-spec/docs/INT_v2_1.pdf>.

TU, N. V.; HYUN, J.; HONG, J. W.-K. Towards onos-based sdn monitoring using in-band network telemetry. In: **2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)**. [S.l.: s.n.], 2017. p. 76–81.

TU, N. V. et al. INTCollector: A High-performance Collector for In-band Network Telemetry. In: **14th International Conference on Network and Service Management (CNSM 2018)**. Rome, Italy: IEEE/IFIP, 2018. p. 10–18.

WHITE, B. et al. An integrated experimental environment for distributed systems and networks. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 36, n. SI, p. 255–270, 2002.

ZHU, Y. et al. Packet-level telemetry in large datacenter networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 45, n. 4, p. 479–491, aug. 2015.