OST
Eastern Switzerland
University of Applied Sciences

Green Networking

Visibility, a first step towards sustainable networking

Bachelor Thesis FS2024

Advisor:	Prof. Laurent Metzger
Co-Advisor:	Severin Dellsperger
Proofreader:	Prof. Dr. Daniel Patrick Politze
Partner:	Dr. Alexander Clemm

Authors: Ramon Bister
Reto Furrer

INS Institute for Network and Security

Submission: 14. June 2024
Version: 1.0

1. Abstract

This thesis is a follow-up project to our term paper that proposed green networking metrics aimed at enhancing the energy efficiency of networking infrastructures. The initial study highlighted the lack of visibility into network energy efficiency, which hampers efforts to optimize sustainability.

The primary objective of this thesis is to demonstrate a comprehensive use case in a virtualized environment where the energy efficiency indicators, developed in the earlier study, are exported and visualized. This aims to provide network operators with the tools necessary to improve network efficiency.

The research involves setting up a proof of concept within a simulated network environment. This includes implementing an IPFIX exporter on network switches to gather efficiency data, establishing collecting servers for persistent storage of this information, and creating dashboards to visualize the network's current state. Additionally, an automation solution is implemented to dynamically configure and update the simulation network.

The project successfully developed a virtualized demo application that simulates an energy efficiency-enabled network, as proposed in the term paper. The demonstration shows that exporting flow efficiency information using IPFIX is straightforward and feasible.

The study concludes that while the export and visualization of efficiency data are straightforward, the collection of such data and the implementation of additional data plane functionalities (such as the IOAM protocol) require support from vendors and must be advocated at the IETF. The research demonstrates that significant information can be extracted from the network with a manageable processing overhead, paving the way for more sustainable networking practices.

2. Management Summary

2.1. Introduction

In today's digital age, the vast network infrastructure supporting our internet and communication systems is a significant consumer of energy. Despite the critical role of networks, there is currently no effective method for retrieving information about the carbon intensity and energy efficiency of network paths and devices. This lack of visibility hampers efforts to identify and mitigate inefficiencies, making it challenging to reduce the overall environmental impact of these systems. This project addresses this gap by proposing a method to export and visualize network telemetry data, which will provide insights into the carbon footprint of networks at both the path and flow levels. The ultimate goal is to lay the groundwork for future improvements in network sustainability.

2.2. Project Objectives and Solution

The primary objective of this project is to develop a solution that can capture and visualize network energy efficiency metrics with minimal processing overhead. The project focuses on implementing a proof of concept within a simulated environment. This proof of concept involves several key components:

IPFIX Exporting Process The project implements an IPFIX (IP Flow Information Export) Exporting Process on network switches. This process captures and exports data related to network traffic and energy usage, facilitating the analysis of network efficiency.

Data Collection and Visualization A collection system utilizing the TIG stack (Telegraf, InfluxDB, Grafana) is set up to store and visualize the efficiency data. This stack enables the creation of comprehensive dashboards that provide real-time insights into the network's energy performance.

Data Plane Optimization with IOAM The project also includes the optimization of network devices' data planes through the implementation of the IOAM (In-situ Operations, Administration, and Maintenance) Aggregation Option protocol extension. This allows for detailed tracking and aggregation of performance metrics, including energy efficiency, across different network paths. Error handling and dynamic aggregator selection are incorporated to ensure accurate and reliable statistics.

The successful implementation of these components demonstrates that it is feasible to obtain detailed energy efficiency insights from network operations with minimal additional overhead.

Figure 2.1 illustrates the solution based on a simple network topology briefly explaining the related components and metrics.

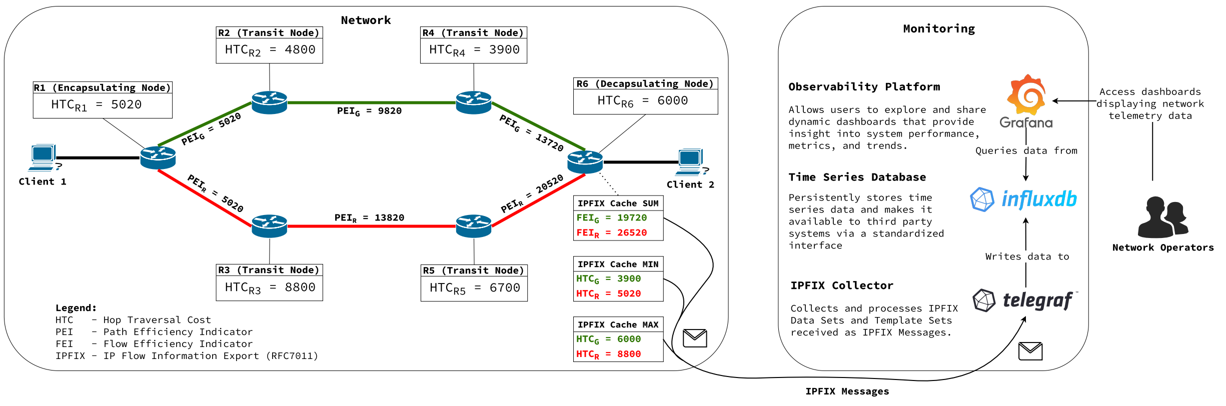


Figure 2.1.: Proof of Concept Overview

2.3. Value of the Solution

This project provides significant value by demonstrating a practical method for obtaining and visualizing energy efficiency data from network infrastructure. Key benefits of the solution include:

Enhanced Visibility The project showcases a method for gaining deep insights into network efficiency, which is currently not possible with existing systems. This visibility is crucial for identifying both efficient and inefficient network components and paths.

Feasibility and Low Overhead The proof of concept confirms that retrieving energy efficiency information from networks is achievable with little processing overhead. This ensures that the proposed solution can be integrated into existing network operations without significant performance degradation.

Path to Practical Implementation The next steps involve exploring how the proposed solution can be adapted for real-world networks. This includes presenting the results at industry conferences, such as those focused on network softwarization, to garner support for standardizing the IOAM Aggregation Trace Option as a Request for Comments (RFC). Additionally, increasing interest among network device vendors to implement this standard is crucial for widespread adoption.

2.4. Conclusion

This project addresses a critical issue in the realm of network sustainability. As computer networks are substantial energy consumers, the ability to monitor and optimize their efficiency is essential for reducing their carbon footprint. The study demonstrates that it is possible to retrieve valuable energy efficiency data from networks in a simulated environment, marking a significant first step toward making networks more sustainable. The insights gained from this research pave the way for future developments aimed at creating a more environmentally friendly network infrastructure. By providing a foundation for better network efficiency management, this work contributes to the broader goal of minimizing the environmental impact of our digital infrastructure.

The findings from this project not only highlight the potential for immediate improvements in network efficiency but also emphasize the importance of continuing to explore and develop sustainable network technologies. As the demand for network connectivity grows, the implementation of such solutions will become increasingly crucial in our efforts to build a more sustainable and ecologically responsible future.

3. Acknowledgement

We would like to express our sincere gratitude to the following individuals and organizations who have supported us throughout the process of completing this bachelor thesis.

First, we are deeply thankful to our thesis advisor, **Professor Laurent Metzger**, for his support during this thesis. His commitment, passion and expertise helped and motivated us a lot.

Thanks to **Severin Dellsperger**, for his support during this thesis. We greatly appreciated the constructive discussions, the respectful interaction, his expertise and commitment to our bachelor thesis.

A special thanks goes to our external partner, **Dr. Alexander Clemm** founder of Sympotech, for his invaluable guidance, encouragement, and expertise. His insightful feedback and constructive criticism played a pivotal role in shaping the direction of our research. We also greatly appreciated the fact that we were involved in the standardisation work and were given the opportunity to work on the implementation of the IOAM Aggregation Option protocol extension in our simulated network environment. Additionally, we would like to thank him for introducing us to IPFIX and providing excellent guidance while writing the paper for the NetSoft 2024 IEEE conference, which goes beyond the scope of this thesis. His support was instrumental in enhancing the quality and reach of our work. We were very pleased with the uncomplicated nature of the collaboration.

We would also like to thank **Jan Untersander** for his prompt technical support and the provisioning of virtual servers for our simulated environment. His assistance ensured the smooth progress of our project and was greatly appreciated.

Ramon Bister, Reto Furrer
OST - Eastern Switzerland University of Applied Sciences
14. June 2024

4. Important Terms and Abbreviations

The efficiency indicators including FEI, HEI, HTC, LEI and PEI are defined and further described in chapter 1 in the elaboration part.

Control Plane Is a logical component of a network device and is responsible for the management and routing decisions that determine how data packets traverse the network

Data Plane Is a logical component of a network device and is responsible for the real-time forwarding and processing of network packets

FEI Flow Efficiency Indicator

HEI Hop Efficiency Indicator

HTC Hop Traversal Cost

IOAM In situ Operations, Administration, and Maintenance RFC 9197

IPFIX IP Flow Information Export (IPFIX) Protocol RFC 7011

LEI Link Efficiency Indicator

P4 Programming Protocol-independent Packet Processors (P4) is a domain-specific language to define the forwarding pipeline of the data plane of network devices

PEI Path Efficiency Indicator

5. Introduction

The task description written by our external partner Alexander Clemm introduces the project accomplished in this bachelor thesis very well.

5.1. Background

Alexander Clemm, Sympotech, Los Gatos, California/USA
February 26th, 2024

In the previous project, we developed a system that instruments a network to provide carbon metrics for networking paths. As part of this, a new protocol was implemented that allows to aggregate telemetry data along a path. Different types of aggregation are supported, including (but not limited to) the sum of telemetry data items of nodes that are encountered along the path. The protocol allows to piggyback on data packets, which ensures path congruency, i.e. the hops for which data is aggregated are indeed the same hops that are being traversed by the underlying production traffic. The carbon path metric provided by the PoC is a new metric referred to as Path carbon Efficiency Indicator (PEI), defined as an aggregate as Hop carbon Efficiency Indicators (HEIs), another newly introduced metric that applies at the node level.

There are a number of possibilities to build on the original project to provide additional features and a more comprehensive system. The following are some of the possibilities that we discussed:

Development of IPFIX-based export of PEI data Instead of exporting data in a proprietary format using simple API calls, exported data would be formatted as IPFIX records for easier ingestion by existing IPFIX collectors.

Support for additional path metrics These metrics might involve other node data requiring additional computation on nodes, hence more suitable to be used in combination with dedicated probe traffic instead of production traffic. A corresponding probe implementation would be included as part of the project.

Development of a more comprehensive demo app One application would involve an application that maintains a real-time carbon intensity matrix for paths whose carbon intensity has been assessed.

Support for flow metrics Extending the system to support not only carbon metrics for paths, but carbon metrics for flows, e.g. the sum of power consumption that can be attributed to the set of all packets that constitute a flow. For example, a flow carbon cost indicator could be introduced that is the sum of PEIs provided by the packets. Support for such metrics would facilitate, for example, new accounting and pricing schemes that are based on the degree of pollution caused as opposed to (or in addition to) other metrics such as traffic volume.

The follow-up project will build on the earlier project, focusing on three main aspects:

IPFIX-based export to the demo application from an egress node Data to be exported using IPFIX formatting. However, in order to avoid that a full IPFIX exporter would need to be implemented, the exporter will support only a minimal, fixed set of Information Elements, including source IP, destination IP, PEI ID, and PEI. Optionally a few more Information Elements could be supported, notably source port and destination port.

PEI ID and PEI are new Information Elements that will require allocation of a new IE ID. Standardized IEs are registered here: https://www.iana.org/assignments/ipfix/ipfix.xhtml; in addition, enterprise-specific IEs can be assigned as well (which would require an enterprise ID registered with IANA, here: https://www.iana.org/assignments/enterprise-numbers/). For the purposes of this project, we can simply use IE IDs which have not been assigned, e.g. 5050 for PEI ID and 5051 for PEI.

As HEIs and hence PEIs can be configured and customized by users, there is no single semantics per PEI. In order to support different PEIs, users can assign a PEI ID to custom PEI versions by which they can be differentiated.

The export format will be specified using an implied IPFIX template that is assumed to be preprovisioned at the IPFIX exporter. Note, no IPFIX templates need to be actually implemented; they only serve to define the format according to which data gets exported. The following is an example for a candidate IPFIX template that could be used:

Listing 5.1: Example IPFIX Template (ID: 256)

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Set ID = 2         |      Length = 24 octets       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Template ID 256        |        Field Count = 4        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|    sourceIPv6Address = 27   |       Field Length = 16       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0| destinationIPv6Address = 28 |       Field Length = 16       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|         PEI ID = 5050       |       Field Length = 4        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|           PEI = 5051        |       Field Length = 4        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

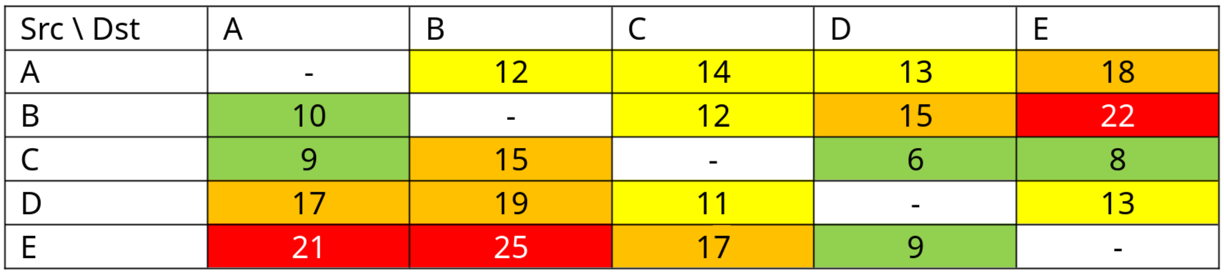
Using the above hyptothetical temple, exported data would then be formatted as follows (containing 2 records):

Listing 5.2: Example Data Set using Template 256

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Set ID = 256         |          Length = 84          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          sourceIPv6                           |
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           destIPv6                            |
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             PEI ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              PEI                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          sourceIPv6                           |
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           destIPv6                            |
:                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             PEI ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              PEI                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Support for configurable HEI/PEI determined by IOAM control plane The current implementation fundamentally already allows for configurable HEIs and hence PEIs. However, it does not allow for multiple different HEIs to be supported concurrently. In this extension, a user (i.e. network administrator) will be able to configure different HEIs to be aggregated by different PEIs under different PEI IDs. A user might in fact collect different PEIs for the same path, each aggregated using a separate IOAM packet.

Enhanced demo app One possibility here concerns maintaining a matrix for the network depicting the PEIs for different paths, here: different combinations of sources and destinations. This will be similar to a traffic matrix, but individual cells will contain PEIs and time information. An example of an efficiency matrix to be elaborated in this bachelor thesis is depicted in figure 5.1.



Src \ Dst	A	B	C	D	E
A	-	12	14	13	18
B	10	-	12	15	22
C	9	15	-	6	8
D	17	19	11	-	13
E	21	25	17	9	-

Figure 5.1.: Example Efficiency Matrix

Such a matrix could facilitate sustainability analysis for a network. For example, it could be used to show a "heat map". It would also allow to analyze the impact of topology changes on PEIs.

The demo app will presumably also need to include an orchestrator app to ensure that each node periodically sends traffic used for probing to different destinations.

To make for a good demo, a sandbox testbed should include enough nodes such that the impact of topology changes can be shown (e.g. initiated by a link cut or some intermediate nodes going into or coming out off sleep mode, etc).

Stretch goal: Support for additional components It would be useful to allow additional parameters to be incorporated as components into HEIs. One possibility includes data related to the ingress and egress interfaces of the packet, such as port power consumption. This will allow PEIs to not only reflect sustainability data concerning traversed nodes as a whole, but to provide a more differentiated picture. Parameters might include management data such YANG data nodes or MIB objects. Support for such parameters would presumably involve extern functions and come with the caveat that it should be collected not at line rate, but e.g. via occasional probing packets without high QoS demands.

5.2. Thesis Composition

The documentation of the bachelor thesis is organized in four main parts based on the RUP (Rational Unified Process) project method. The goal of the documentation is to give insight into the most important concepts and challenges of the project. In order to follow along this documentation, one should be familiar with the fundamental computer networking concepts. Especially the terms control plane and data plane in the context of network devices should be clear to the reader. Additionally it is helpful to have read the preceded term paper with the title Green Networking - Visibility, a first step towards sustainable networking for insights into the elaborated concepts regarding efficiency indication in computer networks.

5.2.1. Inception

The inception part of the documentation outlines the existing research, project planning with user stories, requirements analysis, and risk assessment.

5.2.2. Elaboration

In the elaboration part conceptual decisions related to the network virtualization system, the monitoring system and the configuration update system are described. Furthermore a theoretical part introduces the efficiency indicators elaborated, highlighting how they could be used in practice to reduce the carbon footprint of computer networks and the associated challenges for an in production deployment.

5.2.3. Construction

The construction part contains the most important implementation details about the core components such as the BMv2 cache implementation, the IPFIX exporter, the IPFIX collector, the time series database, the traffic generator and the dashboards of the monitoring system. Furthermore data plane optimization introduced in the bachelor thesis and the Wireshark extension for the IOAM Aggregation Option protocol extension are documented.

5.2.4. Transition

The transition part contains a conclusion and discussion. It also outlines possible future work. Additionally a demo case is worked out that shows the complete system in operation digging into a sample scenario clarifying important concepts of this thesis.

5.2.5. Appendix

The appendix lists additional documents such as the poster about the bachelor thesis and the NetSoft 2024 conference paper that emerged from the term paper.

Contents

Part I.

Inception

1. Initial Situation

Sustainability has become a critical concern across various sectors, including computer networking. The substantial energy consumption associated with network infrastructure significantly contributes to the overall carbon footprint of the digital economy. As networks continue to expand and evolve, optimizing their energy efficiency has become paramount to achieving sustainable growth. However, there is currently a significant lack of visibility into the energy efficiency of computer networks. This lack of transparency poses a major challenge to efforts aimed at improving network energy efficiency and, by extension, reducing the carbon footprint of these networks.

1.1. The Need for Energy Efficiency Metrics

To address this challenge, the Internet Engineering Task Force (IETF) has published a draft titled *Green Networking Metrics*. This document underscores the necessity for network instrumentation capable of assessing power consumption, energy efficiency, and the carbon footprint associated with networks, their equipment, and the services they support. [4] The draft outlines a variety of metrics that are essential for evaluating and enhancing the energy efficiency of networking infrastructures. These metrics serve as critical tools for network operators and researchers seeking to optimize network performance and sustainability.

1.2. Previous Work and Proposed Solutions

In a previous semester, we developed a term paper titled Green Networking - Visibility, a first step towards sustainable networking. This paper introduced a framework for enhancing visibility into network energy efficiency, proposing innovative metrics for assessing and improving network performance. Our research aimed to provide a foundation for sustainable networking practices by offering new ways to measure and optimize energy usage across network infrastructures.

Building on the insights and findings of our term paper, we subsequently authored a paper titled *Towards Sustainable Networking: Unveiling Energy Efficiency Through Hop and Path Efficiency Indicators in Computer Networks*. This paper, which presents the results of our earlier research, was accepted for presentation at the 10th IEEE International Conference on Network Softwarization in Saint Louis, USA. We will present our findings at the conference in June 2024, showcasing our innovative approach to measuring and improving network energy efficiency through the use of hop and path efficiency indicators.

The term paper proposed innovative metrics for indicating network efficiency, designed to be measured as packets traverse the network. This approach aligns with the metrics outlined in the IETF's Green Networking Metrics draft. Specifically, we introduced the concept of the Path Efficiency Indicator (PEI), categorized in the draft as an *Energy Metric related to Paths*. The PEI provides a detailed measure of the energy efficiency of network paths, facilitating the identification of both efficient and inefficient routes.

Additionally, our term paper suggested a method for discovering the most efficient and inefficient hops along a network path based on the Hop Efficiency Indicator (HEI), classified under the draft as *Metrics related to Equipment*. By identifying these critical points within the network, we aim to target specific areas for improvement, thereby enhancing overall network efficiency.

To implement these metrics, we utilized the IOAM (In-situ Operations, Administration, and Maintenance) Aggregation Option protocol extension. This protocol allows for embedding network telemetry data within the packets themselves, enabling real-time monitoring of network conditions. The IOAM Aggregation Option is currently in draft status, but it is anticipated that it will be standardized in the near future. Our external partner, who published the draft on the IOAM Option, provided the foundational support for our implementation in the term paper.

1.3. Industry Insights and External Contributions

Our exploration into network energy efficiency metrics is complemented by insights from industry leaders and academic institutions:

- At the MPLS SD & AI Net World Congress in Paris, Bart Janssens from Colt Technology Services demonstrated a practical solution for carbon-aware routing within their network. Their approach utilizes a simple efficiency indicator on routers, defined as the ratio of current power utilization to available bandwidth. This efficiency information is then sent to a central system via streaming telemetry for further processing. This solution highlights the practical application of efficiency metrics in achieving carbon-aware routing, a concept that aligns with the goals of our research.
- The University of Oxford presented a notable study at the RIPE conference, focusing on the benefits of carbon-aware routing. Their research aimed to quantify the potential advantages of such an approach, introducing the Carbon-Aware Traffic Engineering (CATE) methodology. This study analyzed various metrics and traffic patterns to assess the impact of carbon-aware routing on specific regions in Europe, demonstrating significant potential for reducing the carbon footprint of network operations.

1.4. Optimizing and Expanding on Previous Work

Building on the foundation laid by our term paper and the subsequent IEEE conference paper, the current project seeks to further optimize the proof of concept (PoC) for the efficiency indicator and propose additional *Green Metrics related to Flows*, as outlined in the Green Networking Metrics draft. Our goal is to enhance the granularity and accuracy of the metrics, thereby providing deeper insights into network performance and energy efficiency.

A critical challenge that the project addresses is the accessibility of collected telemetry data. Until now, this data has not been accessible outside the network, limiting its utility for broader analysis and optimization. To overcome this limitation, the project proposes exporting the collected telemetry data using IPFIX (IP Flow Information Export). This will enable external stakeholders to access detailed network efficiency metrics, providing a comprehensive view of network performance.

1.5. Leveraging Modern Technologies for Data Management

To efficiently retrieve, store, and visualize the exported data, the project will leverage modern data management technologies such as the TIG Stack. The TIG Stack, which includes Telegraf, InfluxDB, and Grafana, is well-suited for handling network telemetry data. Telegraf will be used for data collection, InfluxDB for data storage, and Grafana for data visualization. This combination of tools will facilitate the creation of detailed dashboards that provide real-time insights into network efficiency, supporting ongoing efforts to optimize network performance and reduce energy consumption.

2. Requirements

This chapter contains the functional and non-functional requirements of the project. The requirements are specified based on the *FURPS* classification method. *FURPS* is an acronym and stands for:

Functionality Capability, Reusability, Security

Usability Human Factors, Aesthetics, Consistency

Reliability Availability, Recoverability, Accuracy

Performance Speed, Efficiency, Resource Consumption, Scalability

Supportability Maintainability, Testability, Flexibility

The information above is based on the *FURPS* Wikipedia entry [7].

2.1. Functional Requirements

In our project, functional requirements are essential and represent the *F* in the *FURPS* model, which stands for functionality. They describe exactly what tasks and functions the system must fulfill. To organize our development process, we have used Epics to categorize and order these requirements. Each Epic consolidates a group of related features, which are accompanied by corresponding User Stories, providing a user-centric way to articulate needs and perspectives.

2.1.1. Epics

ID	EP1
Subject	Energy Efficiency Insight for Network Paths
Description	As a network operator, I need a comprehensive solution to optimize energy usage across network paths. This epic entails the development of tools and algorithms to analyze and compare the energy efficiency of different paths within the network. By achieving this, operators can identify optimal paths to place in standby mode during low traffic periods, thereby maximizing energy savings while maintaining network functionality.
Remark	Partially achieved in preliminary work (term paper) with the elaboration of the hop efficiency indicator (HEI) and path efficiency indicator (PEI).

ID	EP2
Subject	Energy Efficiency Insight for Network Flows
Description	This epic aims to empower network operators with insights into the energy efficiency of network flows. It involves developing tools and features to visualize the efficiency of flows. By providing operators with clear visibility into the energy consumption patterns of network flows, they can make informed decisions to optimize energy usage and enhance overall network performance.

ID	EP3
Subject	Standardized Export and Centralized Management of Energy Efficiency Data
Description	This epic focuses on establishing a standardized approach for exporting energy efficiency data and centralizing its management within a designated system. It encompasses the development of mechanisms for collecting, processing, and persisting energy efficiency data from various sources into a centralized repository. By implementing this epic, network operators can streamline the management of energy efficiency information, enabling better analysis, reporting, and decision-making processes.

ID	EP4
Subject	Proof of Concept Simulation for Energy Efficiency Features
Description	This epic is dedicated to creating a proof of concept simulation environment for the energy efficiency features outlined in previous epics. It involves developing a virtualized network infrastructure capable of simulating network traffic and dynamically adjusting the hop efficiency of individual routers. The primary goal is to demonstrate the feasibility and effectiveness of energy optimization strategies.

2.1.2. User Stories

ID	US1
Epic	EP1
Subject	Hop Efficiency Indicator (HEI) Processing
Description	As a network operator, I need an efficiency indicator on the hop level, so that path and flow level energy efficiency metrics can be processed.

ID	US2
Epic	EP1
Subject	Link Efficiency Indicator (LEI) Processing
Description	As a network operator, I need an indicator which can be added to the HEI and considers ingress and egress link efficiency, so that the final value stored in the aggregate makes a better statement about the over all energy efficiency of the selected path of the data packet. Not all links necessarily have the same efficiency (WAN/LAN link, number of repeaters, wireless connections, fibre, copper, etc.).

ID	US3
Epic	EP1
Subject	Simultaneous Collection of Several Different Hop Efficiency Indicators (HEI)
Description	As a network operator, I want to have the possibility to collect various Hop Efficiency Indicators simultaneously, so that the information loss through aggregation can be minimized.

ID	US4
Epic	EP1
Subject	Wireshark Dissector
Description	As a network operator, I want to view the network telemetry data stored in the packet inside the IOAM Aggregation Option in Wireshark, so that there is better observability and troubleshooting possibilities in efficiency indicator enabled networks.

ID	US5
Epic	EP1
Subject	View Path Statistics and Identify Inefficient Paths and Nodes
Description	As a network operator, I want to view path statistics such as, the hops traversed, the indicator value, the source and destination IP address grouped by indicator type, so that I can take that information to identify inefficient paths and nodes to be able to apply targeted improvements to the efficiency of my network.

ID	US6
Epic	EP2
Subject	Visualize Efficiency of Flows in a Heat Map
Description	As a network operator, I want to view the efficiency of flows in a heat map, so that I can see the efficiency of point to point communications at a glance.

ID	US7
Epic	EP2
Subject	View Flow Statistics and Identify Efficient and Inefficient Flows Between Hosts
Description	As a network operator, I want to view flow statistics such as, efficiency indicator values, flow duration, number of packets and flow endpoints so that I have a more detailed insight into the efficiency of flows as available in the heat map.

ID	US8
Epic	EP3
Subject	IPFIX Metering and Exporting Process for the Export of Path Specific Efficiency Data
Description	As a network operator, I want to use an IPFIX Metering and Exporting Process to export path specific energy efficiency data, so that the data can be imported on a remote system with a standard compliant IPFIX Collector.

ID	US9
Epic	EP3
Subject	IPFIX Metering and Exporting Process for the Export of Flow Specific Efficiency Data
Description	As a network operator, I want to use an IPFIX Metering and Exporting Process to export flow specific energy efficiency data, so that the data can be imported on a remote system with a standard compliant IPFIX Collector.

ID	US10
Epic	EP3
Subject	IPFIX Collecting Process for the Import of Path Specific Efficiency Data
Description	As a network operator, I want to use an IPFIX collector to import the path specific efficiency data in a standardized way, so that the data can be used by a third party application for further analysis.

ID	US11
Epic	EP3
Subject	IPFIX Collecting Process for the Import of Flow Specific Efficiency Data
Description	As a network operator, I want to use an IPFIX collector to import the flow specific efficiency data in a standardized way, so that the data can be used by a third party application for further analysis.

ID	US12
Epic	EP3
Subject	Persistent Storage for Energy Efficiency Data
Description	As a network operator, I want to persist network energy efficiency data in a time series database, so that it can be used for history purposes.

ID	US13
Epic	EP3
Subject	Dashboard for Energy Efficiency Data Visualization
Description	As a network operator, I want to display the network energy efficiency history data from the time series database in a customizable dashboard to have all network energy efficiency data available at a glance.

ID	US14
Epic	EP4
Subject	Automated Deployment of Simulation Network with Custom Topology
Description	As a project engineer, I want to be able to simulate an energy efficiency indicator enabled network with an arbitrary topology based on an infrastructure as code (IaC) approach, so that I can test the system in different environments without the need to manually create the Mininet topology file and BMv2 control plane definitions.

ID	US15
Epic	EP4
Subject	Network Traffic Simulation
Description	As a project engineer, I want to be able to generate network traffic and send it through the simulated network in order to retrieve network efficiency telemetry data.

ID	US16
Epic	EP4
Subject	Simulation of Dynamic Hop Efficiency Indicator values
Description	As a project engineer, I want to be able to simulate a dynamic environment by adjusting the HEI values on the routers, so that specific scenarios (e.g. day/night) can be simulated and more interesting efficiency indicator data can be obtained from the network.

2.2. Non Functional Requirements

Based on the FURPS classification the non functional requirements are categorized in the following categories:

- Usability
- Reliability
- Performance
- Supportability

ID	NFR1
Subject	Efficiency Indicator Processing Performance
Category	Performance
Priority	High
System	Network Virtualization System
Description	Whether the efficiency indicator processing is enabled or disabled in a network should not affect the actual data forwarding. Packets which can not carry additional metadata in the header because they already reached there maximum size shall be forwarded without the efficiency indicator processing.
Justification	Determining the efficiency of a network is not the core responsibility of a network. The reliability of a network is one of the most important requirements. This should not be undermined with the determination of the efficiency of the network.

ID	NFR2
Subject	Flow Efficiency Indicator Comparability
Category	Usability
Priority	High
System	Network Virutalisation System and Monitoring System
Description	The number of packets in a flow is variable and the FEI is an aggregation of the PEI carried by each packet. To ensure the FEIs of different flows are comparable the aggregation must consider the variable amount of packets.
Justification	To create meaningful statistics the FEIs must be comparable.

ID	NFR3
Subject	Interruptionless Operation of the Simulation Environment
Category	Reliability
Priority	High
System	Network Virutalisation System and Monitoring System
Description	All components within the network virtualization system and the monitoring system need to run for at least two weeks with no interruption.
Justification	The dashboards not only include current data but also historical data. To be able to show how the network efficiency changed over time a reasonable amount of history data must be available.

ID	NFR4
Subject	Dynamic Aggregator Selection
Category	Usability
Priority	High
System	Network Virtualization System
Description	The dynamic aggregator selection by the ingress node ensures the availability of both the overall path statistics and the discovery data of the most efficient and inefficient hop by path.
Justification	The periodic switching of the aggregator would add additional complexity to operate the system and with the dynamic selection approach there is a high probability that the efficiency data with every aggregator is available for each flow, which is a benefit for the statistical insights.

ID	NFR5
Subject	Efficiency Indicator Type Selection
Category	Usability
Priority	High
System	Network Virtualization System
Description	The indicator type of interest should be settable by the network operator and changeable at any point in time.
Justification	The availability of multiple efficiency indicator types each used for a specific efficiency value solves the issue of the loss of information when multiple efficiency values are aggregated to one efficiency indicator. The static configuration of the IOAM data param in the data plane is not an option.

ID	NFR6
Subject	Reliable Processing and Export of Efficiency Indicator Metadata
Category	Reliability
Priority	Medium
System	Network Virtualization System
Description	The IPFIX implementation called by the data plane through an extern function shall be able to export the metadata of at least 90% of the packets being forwarded with the configurations present in our simulation network which results in approximately 1000 packets / minute being sent by each of the 11 hosts.
Justification	To meaningfully represent the current efficiency of the network the metadata of a certain amount of packets is required but some loss is acceptable.

ID	NFR7
Subject	Constant Cache Entry Size for Aggregated Flow Record
Category	Performance
Priority	High
System	Network Virtualization System
Description	The IPFIX cache implementation should be designed that the size of an individual cache entry to store a flow record for the aggregated export is constant no matter how many packets belong to a specific flow.
Justification	The memory resources on a network device are limited.

ID	NFR8
Subject	Usage of Standardized IPFIX Format
Category	Usability, Supportability
Priority	High
System	Network Virtualization System
Description	To ensure interoperability with existing monitoring solutions and follow network telemetry data export best practices the standardized export format IPFIX shall be used.
Justification	The usage of a proprietary export format would be an additional hindrance for use in production.

ID	NFR9
Subject	Configuration Update on Switches
Category	Usability, Performance
Priority	High
System	Configuration Update System
Description	All switches in a topology can be updated issuing a single command and the execution takes less than 3 seconds per switch given the updated configuration files.
Justification	During the configuration update the network is in inconsistent (unconverged) state which should be as short as possible. Additionally configuration updates are carried out frequently.

ID	NFR10
Subject	Configuration Update on Arbitrary Topology
Category	Supportability
Priority	Medium
System	Configuration Update System
Description	Given the inventory specification of the target topology the configuration of switches within that topology can be updated.
Justification	The configuration update utility is used for at least two topologies. The development and simulation network.

ID	NFR11
Subject	Configuration Update at Switch Runtime
Category	Usability, Reliability
Priority	High
System	Configuration Update System
Description	The configuration update of the switches must be possible without a reboot.
Justification	A reboot of the switches to update the configuration is not acceptable. Otherwise network efficiency data stored in the caches would be lost and additionally the network would be completely down until the update is completed.

ID	NFR12
Subject	Self Explaining Dashboards
Category	Usability
Priority	Medium
System	Monitoring System
Description	The dashboards must be well organized and the most of it should be self explaining.
Justification	A new user (network operator) should need less than 30 minutes of introduction to understand the dashboards.

ID	NFR13
Subject	Monitoring System Startup
Category	Usability
Priority	Medium
System	Monitoring System
Description	All components of the monitoring system should be started by issuing a single command.
Justification	The deployment or restart should be as simple as possible for the network operators.

ID	NFR14
Subject	Storage Management to Avoid Starvation
Category	Reliability
Priority	High
System	Monitoring System
Description	The time series database must be able to store the telemetry data for at least two weeks.
Justification	A storage starvation on the monitoring system is not acceptable. Otherwise the system will be temporarily interrupted and unable to accept IPFIX export messages.

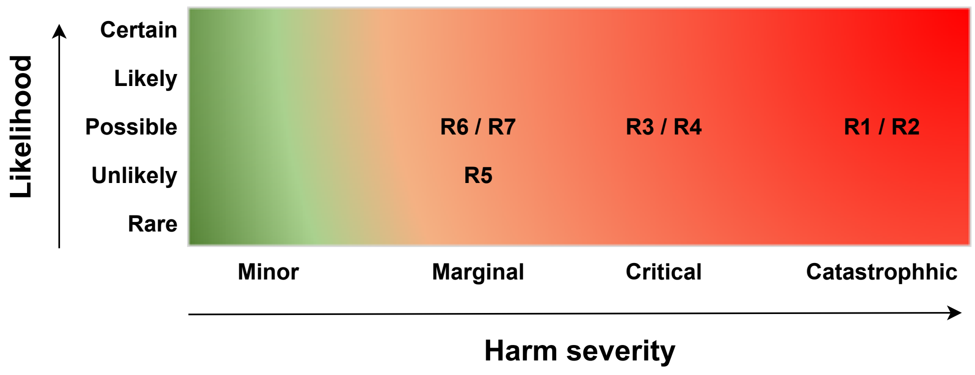
ID	NFR15
Subject	Robust Collection and Delivery
Category	Reliability
Priority	High
System	Monitoring System
Description	The IPFIX collector must be able to process at least 90% of the telemetry data exported by the network virtualization system with the settings applied in our simulation network within a 2h measurement period.
Justification	The monitoring system must be reliable, especially the IPFIX collector must be able to handle the load of the IPFIX exporter. Otherwise, the exported telemetry from the IPFIX exports will be lost.

3. Risk Assessment

In this risk assessment, potential risks to our project are recorded and evaluated according to their respective degree of severity. If a new risk is identified, it is recorded and classified using the risk matrix [13], a common tool for risk assessment.

3.1. Technical Risks

The following risks were identified as possibly disruptive to the project.



ID	R1
Subject	Impossible to extend the BMv2 control plane with a cache management system to temporarily store flow records before they are exported with IPFIX.
Description	The cache stores flow records which are continuously updated during the lifetime of a flow.
Risk	Very high *(Possible/Catastrophic)*
Mitigation	During the elaboration phase of the project, a tracer shall be implemented which demonstrates, that the BMv2 control plane can be extended with the required caching functionality.
Mitigated	Yes

ID	R2
Subject	Impossible to implement a custom extern function on the BMv2 control plane which is callable from the P4 data plane.
Description	External functions are used to trigger actions on the control plane or return external properties or values that are outside of the data plane. In our particular scenario, we plan to use an extern function to pass energy efficiency indicator metadata of the forwarded packet to the control plane. The call of that extern function should then trigger an update of the specific flow record based on the metadata to process.
Risk	Very high *(Possible/Catastrophic)*
Mitigation	During the elaboration phase of the project, a tracer shall be implemented which demonstrates, that the BMv2 control plane can be extended with an extern function which can be called from the P4 data plane.
Mitigated	Yes

ID	R3
Subject	Unable to send IPFIX messages to the monitoring system from within the Mininet environment.
Description	The collected energy efficiency data must be sent to a monitoring server as IPFIX messages, which is either internal or external to the Mininet environment. That monitoring server will persist the collected data within a time series database.
Risk	High *(Possible/Critical)*
Mitigation	There are two possible options on how to transmit IPFIX messages from the BMv2 targets to the monitoring system. • Operate the monitoring server on a host within Mininet. In case this approach is chosen it must be verified that the web interface of the monitoring server can be exposed to systems outside of Mininet. • Operate the monitoring server outside of Mininet. In case this approach is chosen it must be verified that the BMv2 software switches are capable of sending traffic via interfaces outside of Mininet.
Mitigated	Yes. We decided to go for the second approach because the deployment of the monitoring system is more flexible as it is decoupled from the Mininet environment.

ID	R4
Subject	Unable to transmit IPFIX packets on BMv2 targets.
Description	On flow expiry the last hop within the flow exports the cached data and sends it to the IPFIX collector. The BMv2 targets must be capable to craft new valid IPFIX messages.
Risk	High *(Possible/Critical)*
Mitigation	• Implement a tracer which demonstrates that BMv2 targets are capable of crafting and sending UDP datagrams containing an IPFIX message as payload. • Implement a separate application which acts as an adapter between the BMv2 targets and the monitoring system. For that purpose Scapy could be used to translate flow record data received from the BMv2 targets to valid IPFIX messages.
Mitigated	Yes. We decided to go for the first approach and successfully demonstrated that the BMv2 software switches control plane can be extended to be capable to transmit IPFIX messages.

ID	R5
Subject	Unable to simulate realistic patterns in network traffic by traffic generation.
Description	It is necessary to create enough network traffic to produce significant results and statistics about the network's energy efficiency.
Risk	Medium *(Unlikely/Marginal)*
Mitigation	• Use TRex, Cisco's traffic generator to simulate network traffic. • Development of a package generator with Python and Scapy for our specific use case that fulfills our requirements.
Mitigated	Yes. We decided to go for the second approach in order to adapt the functionality of the package generator to our needs.

ID	R6
Subject	The simulation scenario lacks sufficient significance.
Description	In order to achieve significant results, the process of collecting energy efficiency data on the network must be conducted over an extended period of time with the possibility to change network paths and adjust efficiency indicator related values on the control plane.
Risk	High *(Possible/Marginal)*
Mitigation	Setup the Mininet environment on a server and define a topology based on a real service provider network. Run the Mininet simulation over an extended period of time and periodically update the configuration of the BMv2 control plane at runtime and periodically update the configuration of the control plane at runtime.
Mitigated	Yes. We designed a simulation network topology based on a real service provider network topology. Additionally we setup two servers, one server as Mininet simulation network host and the other as monitoring system host. Finally a configuration update system was elaborated which can be used to update the configuration of the BMv2 software switches at runtime.

ID	R7
Subject	Third party application does not provide the required features to achieve our visualization goal.
Description	Visualize the network's energy efficiency data in a Grafana dashboard, particularly using a heatmap to compare the efficiency of end to end communications.
Risk	Medium *(Possible/Marginal)*
Mitigation	Provide simple dashboards from the beginning of the construction phase and improve them continuously. Discuss the dashboards and the visualizations with the stakeholders to achieve continuous improvement.
Mitigated	Yes. Furthermore, we discovered a Grafana plugin called *ESNET Matrix Panel* that fulfills our need to have a comprehensive end to end flow statistics heatmap.

Part II.

Elaboration

1. Efficiency Indicators

1.1. Definitions

HEI The Hop Efficiency Indicator (HEI) is an arbitrary number indicating the efficiency of a hop. There can be several HEI values at the same time, which cover different aspects of a hop's energy efficiency. Which indicator to use is decided by the ingress hop by setting the HEI identifier in the data param header field of the IOAM aggregation option. By randomly selecting the HEI to be used, one statistically obtains an overall view after a certain time, which covers the various aspects of the different HEIs equally. One HEI may indicate the current power to current bandwidth ratio, an other HEI may indicate the ratio of energy retrieved from renewable sources to non-renewable sources. There may be many more HEI values to consider and implement on network devices.

LEI The Link Efficiency Indicator (LEI) is a dedicated value to indicate the efficiency of an interface. For example a 10GBit/s copper interface via a twisted pair cable could be indicated to be less expensive in the means of energy efficiency compared to an interface connected to a 10GBit/s long-haul fiber connection.

HTC The Hop Traversal Cost (HTC) is the result of accumulating the LEI of the ingress link, the HEI and the LEI of the egress link. This is the value added to the aggregate of the IOAM protocol.

PEI The Path Efficiency Indicator (PEI) is the accumulation of HTC values in case the SUM aggregator is used. It indicates the efficiency of the path the packet traversed.

FEI The Flow Efficiency Indicator (FEI) is depending on the aggregator used the average PEI (SUM aggregator), the minimum HEI (MIN aggregator) or maximum HEI (MAX aggregator) considering the network telemetry data of all packets corresponding to the specific flow.

1.2. Efficiency Indicator Processing

The processing of efficiency indicators on network devices is critical in regards to performance. Any additional computation which reduces the overall performance of a device and furthermore has a negative impact on the device's energy efficiency must be kept at a minimum. It must be ensured that the gain out of optimizations is larger than the pain associated with the efficiency indicator related processing.

It is a strict requirement, that the efficiency indicator processing can run at line rate in a real world implementation.

i Information

The line rate or physical-layer frame rate is the maximum capacity to send frames of a specific size at the transmit clock frequency of the device under test. RFC 8238

1.2.1. Processing in Data Plane

Due to the requirement to process the efficiency indicators at line rate, the processing overhead in the data plane should be reduced to a minimum because an inefficient data plane directly impacts the data forwarding performance. During the term paper we followed the approach to gather efficiency indicator related information from the control plane and process the HEI based on that information in the data plane. Doing so results in reprocessing the HEI for every single packet. This could make sense if the underlying values in the control plane change very frequently, but else it is a waste of processing power.

1.2.2. Processing in Control Plane

One of the findings when writing the paper for the IEEE NetSoft 2024 conference about the research results from the term paper, was that it would make more sense to preprocess the HEI value in the control plane. Like that the data plane only has to query the value from the control plane and it has to perform the aggregation operations to store the updated value as network telemetry data in the IP packet. The HEI value recalculation may be triggered in case the underlying values change only. This approach also provides more flexibility in terms of adding multiple independent HEI values, with the data plane then querying the HEI value to be used based on the identifier set in the data param header field of the IOAM aggregation option, or as specified in the configuration in the case of an ingress node when no IOAM header data is available.

1.3. Limitations

The collection of efficiency indicator network telemetry data in transit requires modifications to both the control plane and the data plane of a network device.

1.3.1. Data Plane Extensibility (Production Devices)

As of today only very few programmable network devices are operated in the wild. In a presentation by Arista, the advantages of using P4 in the data plane were mentioned, but it was also emphasized that the use of the technology in practice is still in its infancy. [6] Most of the network devices use fixed function chips which means that the data plane is implemented in hardware. On such devices no modifications to the data plane can be made. This means that it takes a long time until new protocols are generally supported on production devices.

1.3.2. Control Plane Extensibility (Production Devices)

Furthermore the control plane would need to be adjusted with functionality to calculate and expose HEI and LEI values to the data plane using dedicated lookup tables. To ensure interoperability between vendors standards would need to be defined which define which indicator types exist and how they are calculated and made available to the data plane.

1.3.3. PoC Environment

To get around these challenging circumstances we use a simulated network environment with programmable switches. Those switches have a programmable data plane, customizable control plane tables and control plane functionality can be added using plugins. One limitation of the PoC environment is, that no real efficiency data is available, which is why we provide the HEI and LEI values from outside. In other words we mock energy efficiency data. For demonstration purposes this work around is considered good enough.

2. System Overview

This chapter gives context about the systems elaborated in this bachelor thesis. According to the epics defined in section 2.1, the main objective is to collect energy efficiency data of a simulated network environment and to export this data to a central monitoring system where it is made available to a network operator. The people, systems and their relationships relevant to achieve the main objective are visualized in the context diagram in figure 2.1 and the typical system interactions are described in order in the enumeration below.

1. The **network operator** declaratively defines the desired state of the **network virtualization system**.
2. The **network operator** triggers the provisioning of the **network virtualization system**.
3. The **network virtualization system** initializes the topology and starts the traffic simulation process.
4. The **network virtualization system** starts to export network telemetry data to the **monitoring system** using IPFIX messages.
5. The **network operator** accesses the dashboards on the **monitoring system** to get insight into current and history energy efficiency network telemetry data.
6. On demand as the **network operator** wants to simulate a change of efficiency of routers or modify network paths he modifies the desired state of the network virtualization system and triggers the regeneration of the configuration files.
7. The **network operator** triggers the execution of the **configuration update system** to update the configuration of the software switches based on the regenerated configuration stored in the network virtualization system.
8. The **configuration update system** reads the configuration files from the **network virtualization system**.
9. The **configuration update system** sends the updated configuration to the software switches by connecting to the specific gRPC endpoint.
10. The **network operator** observes the changes of the network efficiency in the **monitoring system**.
11. The **network operator** triggers the deprovisioning of the **network virtualization system** once it is not required anymore.

The following chapters describe the concepts behind the individual systems in more detail.

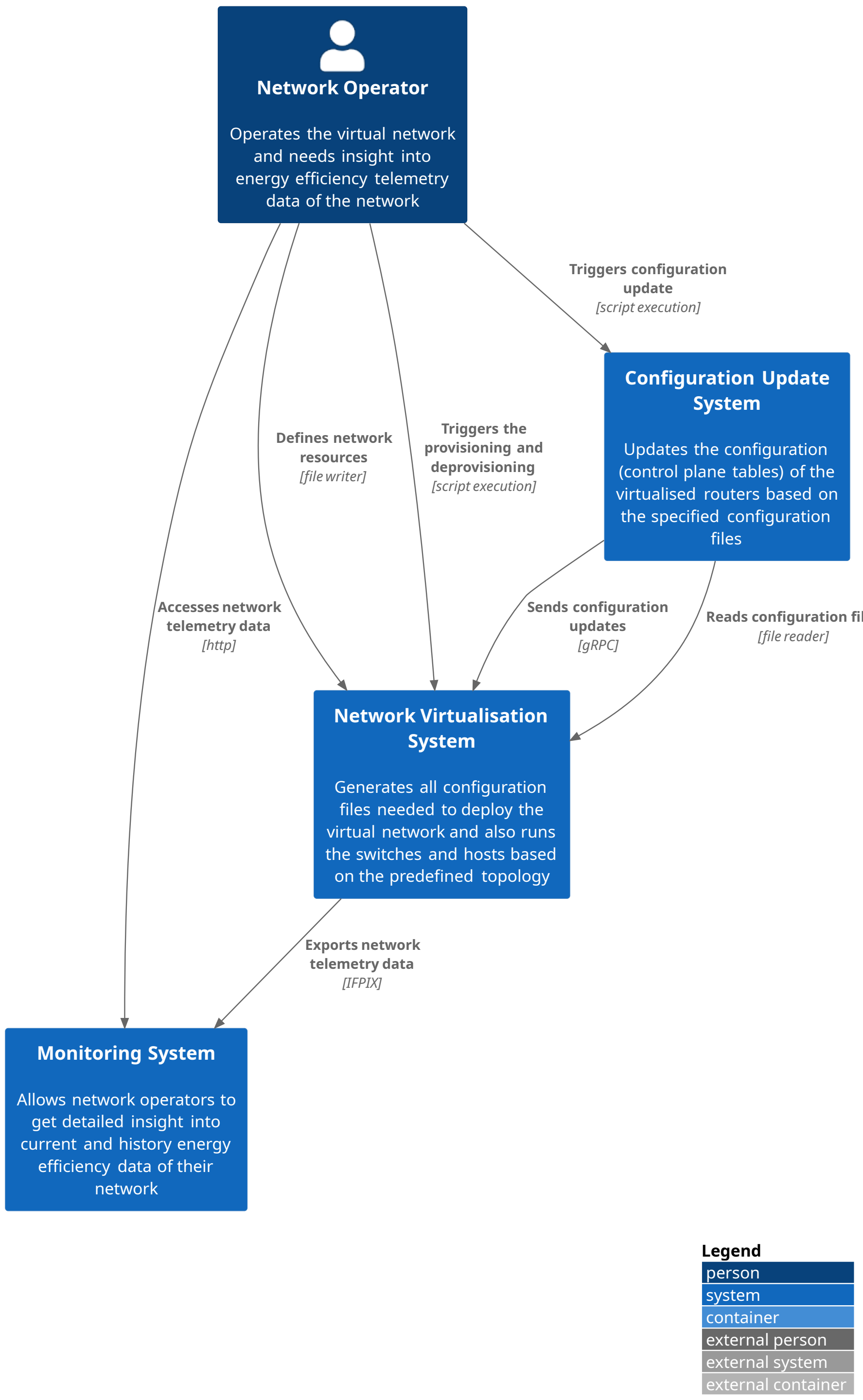

Network Operator
Operates the virtual network and needs insight into energy efficiency telemetry data of the network
Triggers configuration update
[script execution]
Configuration Update System
Updates the configuration (control plane tables) of the virtualised routers based on the specified configuration files
Defines network resources
[file writer]
Triggers the provisioning and deprovisioning
[script execution]
Accesses network telemetry data
[http]
Sends configuration updates
[gRPC]
Reads configuration files
[file reader]
Network Virtualisation System
Generates all configuration files needed to deploy the virtual network and also runs the switches and hosts based on the predefined topology
Exports network telemetry data
[IFPIX]
Monitoring System
Allows network operators to get detailed insight into current and history energy efficiency data of their network
Legend
person
system
container
external person
external system
external container

Figure 2.1.: System Context Diagram

3. Network Virtualization System

This chapter describes the concept how the network virtualization system has been extended in order to achieve IPFIX export functionality. The network virtualization system is run with Mininet, which is a software emulator that creates virtual networks for developing, testing, and experimenting with network applications and protocols. The base configuration of the Mininet environment is based on the setup elaborated in the term paper attempted in the previous semester. More details about Mininet in regards to installation and configuration related to our scenario can be found in the term paper.

Figure 3.1 is a C4 container diagram which illustrates the composition of the network virtualization system. The virtualization network system is composed of programmable network switches, and Linux hosts which are interconnected with links. The core responsibilities of the individual containers are:

Links transmit data between endpoints.

Linux Hosts simulate network traffic.

Programmable Network Switches are used for traffic forwarding in general and specifically for:

- Determination of the energy efficiency indicator value
- Attachment of energy efficiency indicator related data to the packet header as network telemetry data
- Export of all collected metadata related to the energy efficiency indicator via IPFIX (on egress switch only)
- Removal of attached energy efficiency indicator network telemetry data (on egress switch only)

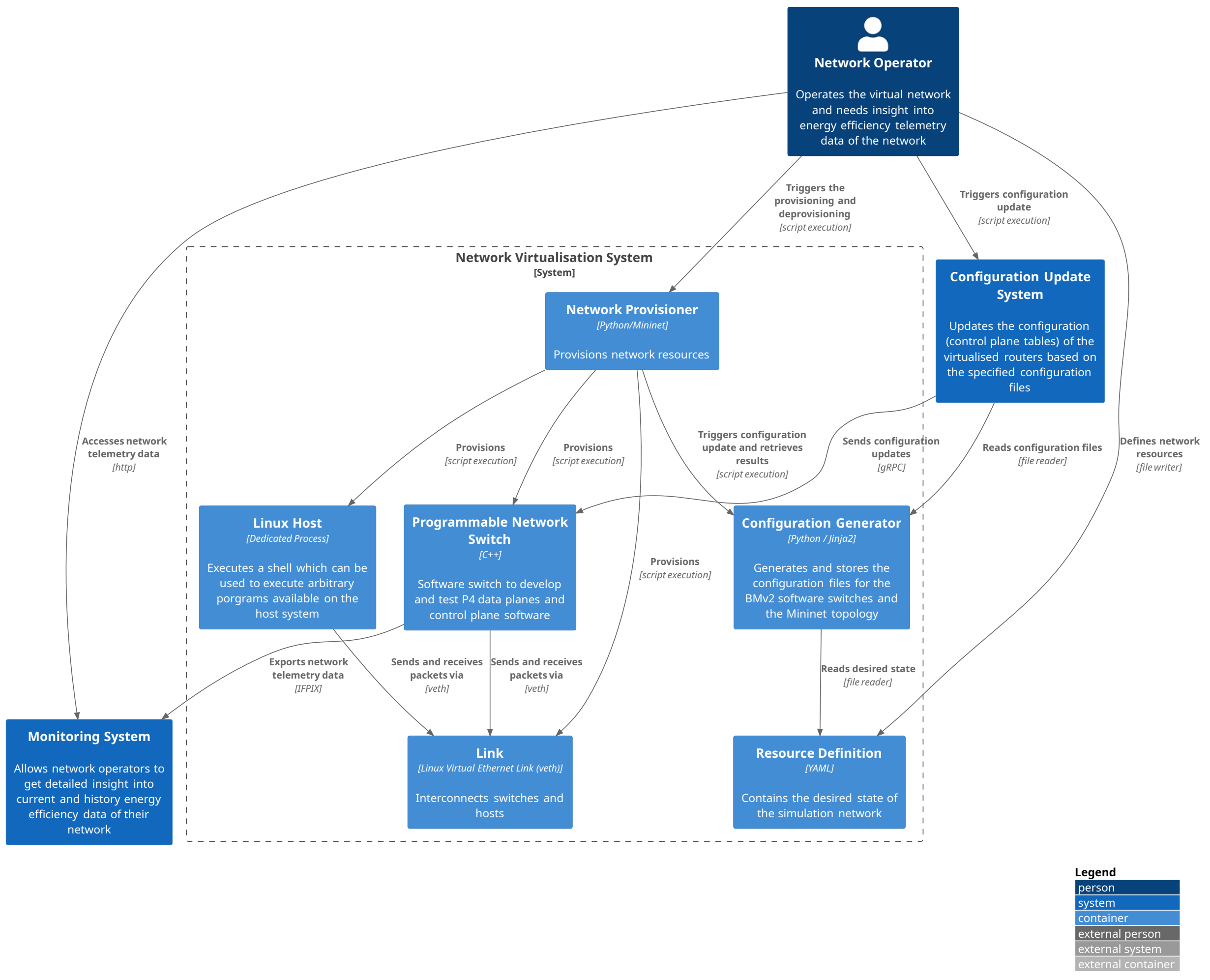

Network Operator
Operates the virtual network and needs insight into energy efficiency telemetry data of the network
Triggers the provisioning and deprovisioning [script execution]
Triggers configuration update [script execution]
Network Virtualisation System [System]
Configuration Update System
Updates the configuration (control plane tables) of the virtualised routers based on the specified configuration files
Network Provisioner [Python/Mininet]
Provisions network resources
Accesses network telemetry data [http]
Provisions [script execution]
Provisions [script execution]
Triggers configuration update and retrieves results [script execution]
Sends configuration updates [gRPC]
Reads configuration files [file reader]
Defines network resources [file writer]
Linux Host [Dedicated Process]
Executes a shell which can be used to execute arbitrary porgrams available on the host system
Programmable Network Switch [C++]
Software switch to develop and test P4 data planes and control plane software
Configuration Generator [Python / Jinja2]
Generates and stores the configuration files for the BMv2 software switches and the Mininet topology
Provisions [script execution]
Exports network telemetry data [IFPIX]
Sends and receives packets via [veth]
Sends and receives packets via [veth]
Reads desired state [file reader]
Monitoring System
Allows network operators to get detailed insight into current and history energy efficiency data of their network
Link [Linux Virtual Ethernet Link (veth)]
Interconnects switches and hosts
Resource Definition [YAML]
Contains the desired state of the simulation network
Legend
person
system
container
external person
external system
external container

Figure 3.1.: Network Virtualization System Container Diagram

3.1. Network Topology

As specified in the functional requirements in section 2.1 in US14 the deployment of an arbitrary network topology should be completely automated based on an infrastructure as code (IaC) approach. To achieve that we use a declarative YAML configuration file, some Python logic and Jinja2 templating to generate all configurations required to deploy the specified network in the Mininet environment. The configuration generation is described in more detail in section 3.2. This approach allows us to define the desired state of our network in a clear, human-readable format. By using this configuration file, we can quickly and efficiently adjust our network settings, ensuring immediate updates. Additionally, this solution provides the flexibility to create completely new network topologies. The declarative format of the files ensures consistency and reduces the risk of configuration errors, resulting in a more reliable and manageable network environment.

3.1.1. Development Network Topology

The network topology depicted in figure 3.2 was used in the term paper and in the beginning of the bachelor thesis before we decided to reconstruct a more realistic simulation network topology.

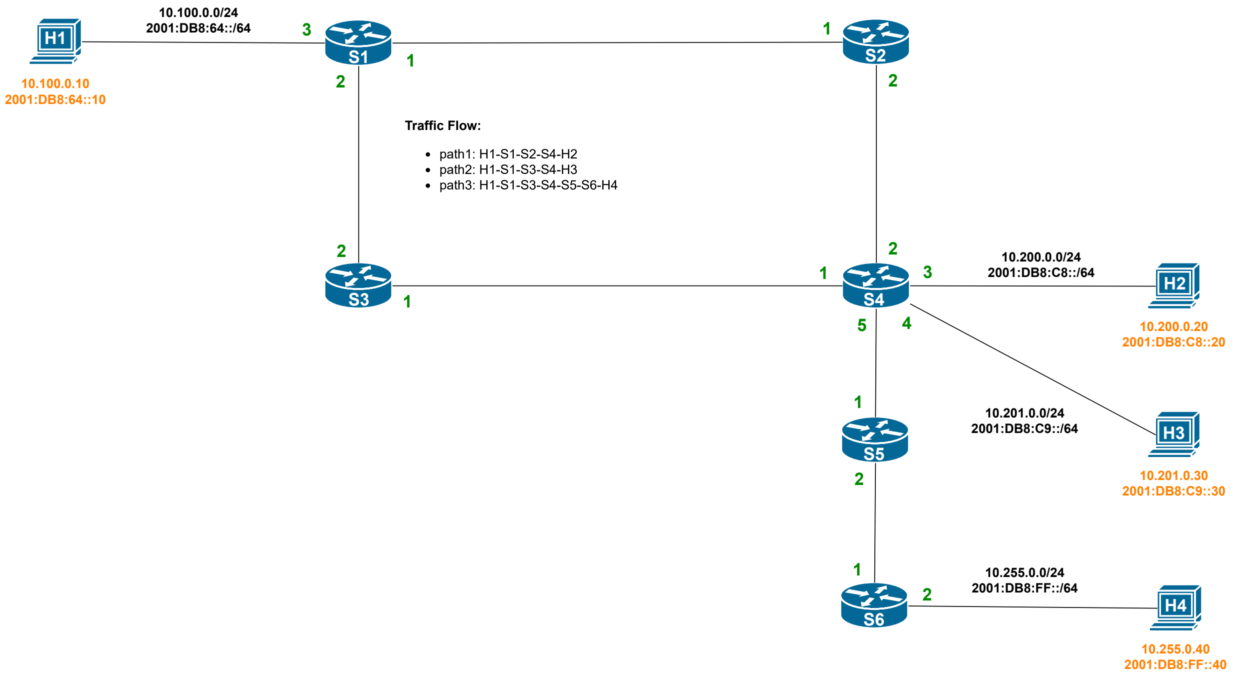


Figure 3.2.: Development Network Topology

3.1.2. Simulation Network Topology

The extended network topology depicted in figure 3.3 is based on the IP ranges and peerings of a real service provider network. It is used to simulate a realistic network in order to test the:

- Implementation of the IOAM Aggregation Trace Option for network efficiency indicators
- IPFIX export mechanism on the BMv2 targets
- Collecting process on behalf of the monitoring system
- Telemetry data generation as the base for the data visualization on the monitoring system

ℹ Information

Referring to figure 3.3 we used the website https://bgp.tools/ to reverse engineer the IP ranges and peerings of AS8758. The underlying physical topology is not representing the actual topology of the service provider network. We chose the topology to have redundant links which gives the possibility to flexibly adjust paths to illustrate the impact of the rerouting of traffic via more efficient paths to the energy efficiency of the network.

3.1.2.1. Links Inside the Autonomous System

The links inside the autonomous system are colored depending on the chosen link efficiency indicator (LEI). For the sake of simplicity the LEI is set according to the expected geographical distance of the routers which are interconnected by the specific link.

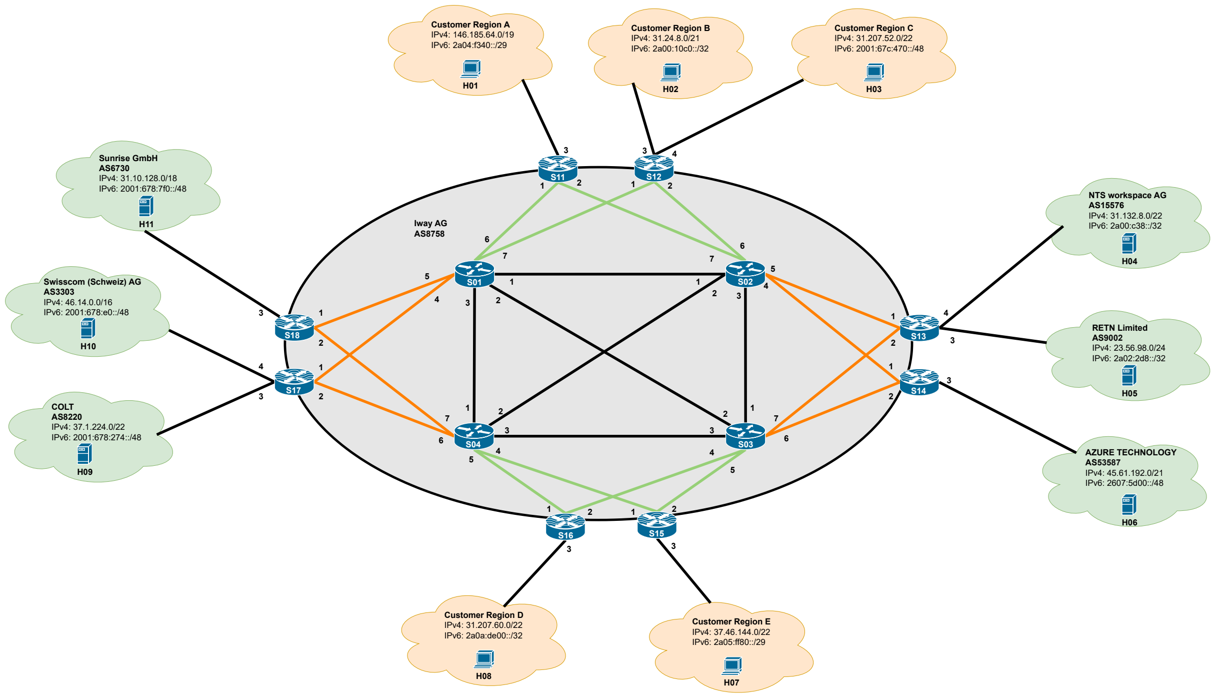


Figure 3.3.: Network Simulation Topology

Black Are links between core switches with a low distance. In our simulation scenario, we consider these links to be the most efficient ones, as the core switches are located in places with a short geographical distance

Green Are links between core switches and edge switches connecting to customer sites. In our simulation scenario, we consider these links to be less efficient than the black links, as the geographical distance is higher and more power is required for the transmission.

Orange Are links between core switches and edge switches connecting to other autonomous systems. In our simulation scenario, we consider these links to be the least efficient links, as the geographical distance is the highest and more power is required for transmission.

3.1.2.2. Router Types

Core Switches (s01-s04) Provide connectivity inside the core network.

Edge Switches (s11-s18) Connect customers and other autonomous systems to the provider core network. These switches run the IPFIX export extension. Efficiency indicator network telemetry data is being processed and sent as IPFIX messages to the IPFIX collector which is part of the monitoring system periodically.

3.1.2.3. IP Addressing

Switches Use unnumbered ports which do not have an IP address assigned. The layer 3 networks inside the AS are limited on the scope of a link between two switches.

Hosts Are configured dual stack which means that they have both an IPv4 and IPv6 address assigned. There is only one host per network and it always uses the 10th host address of the specific IPv4 and IPv6 range respectively.

3.2. Configuration Generator

The configuration generator is designed to read the desired state for a virtual network from a YAML file so that it can generate the configurations that can either be read when the network is initialized or can be pushed at runtime to update the configuration if required.

The container diagram in figure 3.4 zooms into the configuration generator.

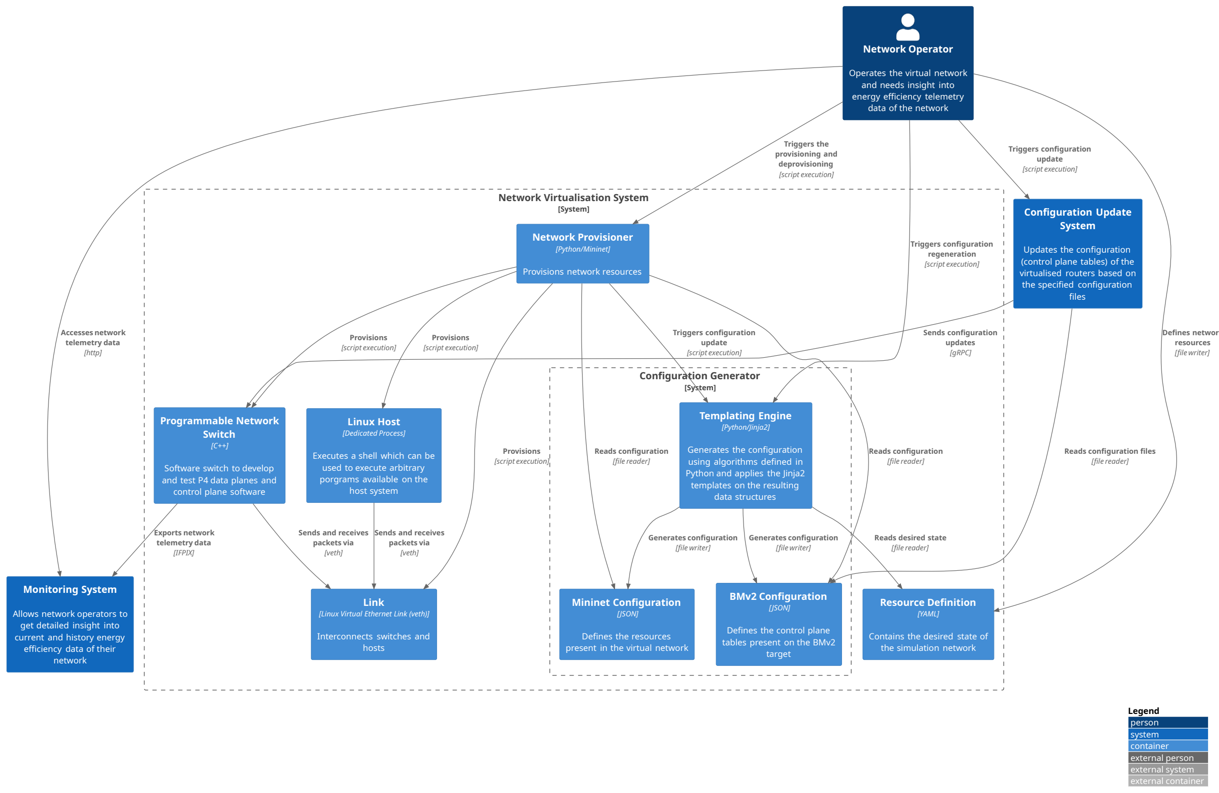

Network Operator
Operates the virtual network and needs insight into energy efficiency telemetry data of the network
Triggers the provisioning and deprovisioning [script execution]
Triggers configuration update [script execution]
Network Virtualisation System [System]
Network Provisioner [Python/Mininet]
Provisions network resources
Triggers configuration regeneration [script execution]
Configuration Update System
Updates the configuration (control plane tables) of the virtualised routers based on the specified configuration files
Accesses network telemetry data [http]
Provisions [script execution]
Provisions [script execution]
Triggers configuration update [script execution]
Sends configuration updates [gRPC]
Defines network resources [file writer]
Configuration Generator [System]
Programmable Network Switch [C++]
Software switch to develop and test P4 data planes and control plane software
Linux Host [Dedicated Process]
Executes a shell which can be used to execute arbitrary porgrams available on the host system
Provisions [script execution]
Reads configuration [file reader]
Templating Engine [Python/Jinja2]
Generates the configuration using algorithms defined in Python and applies the Jinja2 templates on the resulting data structures
Reads configuration [file reader]
Reads configuration files [file reader]
Exports network telemetry data [IFPIX]
Sends and receives packets via [veth]
Sends and receives packets via [veth]
Generates configuration [file writer]
Generates configuration [file writer]
Reads desired state [file reader]
Monitoring System
Allows network operators to get detailed insight into current and history energy efficiency data of their network
Link [Linux Virtual Ethernet Link (veth)]
Interconnects switches and hosts
Mininet Configuration [JSON]
Defines the resources present in the virtual network
BMv2 Configuration [JSON]
Defines the control plane tables present on the BMv2 target
Resource Definition [YAML]
Contains the desired state of the simulation network
Legend
person
system
container
external person
external system
external container

Figure 3.4.: Configuration Generator Component Diagram

3.2.1. Requirements

The configuration generator needs to fulfill the following requirements derived from user story 14 associated to epic 4 as described in chapter 2 in the inception phase.

Virtual Network Deployment at Scale The manual definition of the BMv2 runtime (control plane tables) is tedious work and does not scale for larger topologies. In order to deploy large virtual network topologies these configurations need to be generated automatically.

Declarative Definition of Desired State There must be an abstraction layer for the specification of the information relevant to deploy a virtual network. Additionally there shall be a single source of truth for this information which can be used by the related software components.

3.2.2. Configuration Generation Process

The configuration generation process is run through in two cases. Both descriptions refer to figure 3.4.

3.2.2.1. Network Virtualization System Startup

Each time the network virtualization system is started the configuration is regenerated based on the information specified in the resource definition YAML file. When the system starts up the following steps are taken:

1. The **network operator** defines the required state in the **resources yaml** file.
2. The **network operator** triggers the provisioning by executing the make target to start the **network provisioner**.
3. The **network provisioner** triggers the generation of the BMv2 and Mininet **configurations** by the execution of the **templating engine** script.
4. The **templating engine** reads the **resources yaml** file.
5. The **templating engine** generates the BMv2 and the Mininet **configurations**.
6. The **network provisioner** reads the BMv2 and the Mininet **configurations**.
7. The **network provisioner** bootstraps the virtual network including switches, hosts and links, based on the **configurations**.

3.2.2.2. Configuration Update

In case the BMv2 switches need to be reconfigured at runtime, the following steps are taken:

1. The **network operator** redefines the required state in the **resources yaml** file.
2. The **network operator** triggers the generation of the BMv2 and Mininet **configurations** by the execution of the make target which launches the **templating engine** script.
3. The **templating engine** reads the **resources yaml** file.
4. The **templating engine** generates the BMv2 and the Mininet **configurations**.
5. The **config updater** reads the BMv2 programmable switch **configurations**.
6. The **config updater** pushes the updated **configurations** to the **programmable network switches**.

3.2.3. Resource Definition

The resource definition allows the specification of all required information in YAML format to deploy the virtual network. Part of the yaml definition are:

- File path to P4 data plane program
- Network paths between hosts
- Host details
- Switch details
- Table details

The resource definition format is discussed in chapter **TODO: Add reference** in the construction part.

3.3. Traffic Generator

The traffic generator is designed to generate network traffic within the simulation network with unpredictable behavior by the selection of a random destination host, flow label and amount of packets per flow. The traffic generator is a software component which is part of the Linux host (see figure 3.1). Each host in the simulation network topology will send different flows with a various number of packets per flow to all other hosts.

3.3.1. Requirements

The traffic generator needs to fulfill the following requirements derived from user story 15 associated to epic 4 as described in chapter 2 in the inception phase to ensure versatile simulation scenarios.

Declarative Configuration The configuration of the traffic generator must be dynamic and in a declarative way to ensure that the traffic generator runs in different simulation network topologies.

Automated Start The traffic generator should start automatically with a delay of a configurable amount of seconds after the virtualization system is up and running.

Sending Packet Each host can send a random number of packets to all other hosts.

Flow Label A random and unique flow label is selected for each new connection.

Destination Host The sending host selects a random and valid destination host.

Number of Packets A random amount of packets in a predefined range is chosen and sent.

Source Port A random ephemeral source port from a range of well known UDP ports will be taken.

Transmission Timeout A random number of seconds is selected for the time between flow transmission.

Command Line Parameters The traffic generator is an automated solution but it is necessary that it can be run manually and all the above requirements options can be specified via command line parameters. This ensures that the network operator is able to perform targeted and specific network analysis by sending specific flows without the random factors that would otherwise interfere. For example, the network operator can specify a flow label, source host, destination host, and the number of packets, and is now able to trace the path of the flow and the applied efficiency indicator values.

3.3.2. Traffic Generator Startup

Each time the network virtualization system is started the configuration for the traffic generator is regenerated based on the information specified in the resource definition YAML file. When the system starts up the following additional steps are taken compared to the previously described process in section 3.2.2.1:

1. The **templating engine** generates the traffic generator **configuration**.
2. After the bootstrap process is triggered by the **network provisioner** and the BMv2 switches are up and running the traffic generator is starting up based on the traffic generator **configuration**.

After a delay of a configurable amount of seconds all hosts in the simulation network topology start to generate traffic.

ℹ Information

The further usage as well as the implementation and configuration specific information can be found in the construction phase in section 1.3 traffic generator.

3.3.3. Configuration

A separate traffic generator configuration file is generated for each host in the simulation network topology based on the current resource definition YAML file, which is therefore the single source of truth for all resources inside the simulation network. The resource definition YAML file is further described in chapter 1 in the construction part.

3.3.4. Logfiles

Each host creates a log file with its name and the current traffic generator start-up timestamp. The following information is logged for each flow and can be used for further analysis:

- Flow label
- Source host
- Destination host
- Amount of packets
- Delay between sending a new flow

3.4. Programmable Network Switch

The programmable network switch used in this project, is the second version of the reference P4 software switch Behavioral Model (BMv2). It is written in C++ and the forwarding pipeline of the data plane can be defined using the P4 programming language. The project is open source and available on GitHub: https://github.com/p4lang/behavioral-model.

The C4 component diagram in figure 3.5 visualizes the relationships of the involved systems with a detailed view of the programmable network switch system. The programmable network switch is composed of four components:

Data Plane Component written in P4. The data plane is a forwarding pipeline which takes network packets as inputs, parses the headers and performs so called match actions where header fields are updated and metadata information is set. The data plane retrieves forwarding and energy efficiency indicator information from the control plane by table lookups.

Control Plane Component written in C++ which exposes key value stores called lookup tables to the data plane. All information required to perform the forwarding operation including energy efficiency indicator information is stored in the control plane tables.

Configuration Update Interface Exposes a gRPC interface to update the information stored in the control plane. The configuration update system uses this interface to change the configuration of the programmable switches at runtime.

IPFIX Extension Implements an extern function to temporarily cache and export energy efficiency indicator metadata with the BMv2 programmable software switch. It collects energy efficiency indicator information on a per flow basis and stores it in flow records in a local in memory cache. Once the records expire, they are sent as an IPFIX message to the monitoring system. The implementation of this component is documented in chapter 1 in the construction part.

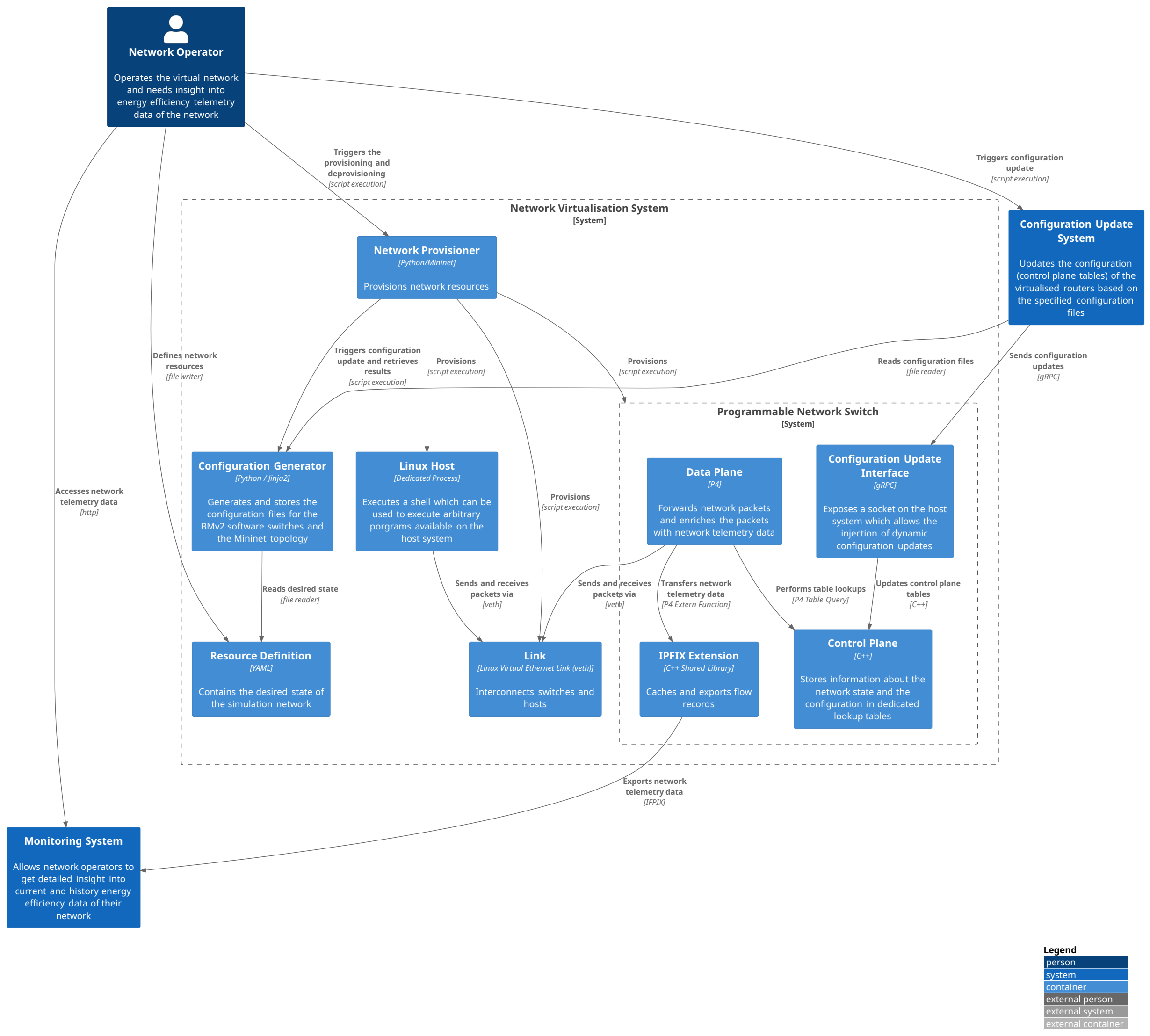

Network Operator
Operates the virtual network and needs insight into energy efficiency telemetry data of the network
Triggers the provisioning and deprovisioning [script execution]
Triggers configuration update [script execution]
Network Virtualisation System [System]
Network Provisioner [Python/Mininet]
Provisions network resources
Configuration Update System
Updates the configuration (control plane tables) of the virtualised routers based on the specified configuration files
Defines network resources [file writer]
Triggers configuration update and retrieves results [script execution]
Provisions [script execution]
Provisions [script execution]
Reads configuration files [file reader]
Sends configuration updates [gRPC]
Programmable Network Switch [System]
Accesses network telemetry data [http]
Configuration Generator [Python / Jinja2]
Generates and stores the configuration files for the BMv2 software switches and the Mininet topology
Linux Host [Dedicated Process]
Executes a shell which can be used to execute arbitrary porgrams available on the host system
Provisions [script execution]
Data Plane [P4]
Forwards network packets and enriches the packets with network telemetry data
Configuration Update Interface [gRPC]
Exposes a socket on the host system which allows the injection of dynamic configuration updates
Reads desired state [file reader]
Sends and receives packets via [veth]
Sends and receives packets via [veth]
Transfers network telemetry data [P4 Extern Function]
Performs table lookups [P4 Table Query]
Updates control plane tables [C++]
Resource Definition [YAML]
Contains the desired state of the simulation network
Link [Linux Virtual Ethernet Link (veth)]
Interconnects switches and hosts
IPFIX Extension [C++ Shared Library]
Caches and exports flow records
Control Plane [C++]
Stores information about the network state and the configuration in dedicated lookup tables
Exports network telemetry data [IFPIX]
Monitoring System
Allows network operators to get detailed insight into current and history energy efficiency data of their network
Legend
person
system
container
external person
external system
external container

Figure 3.5.: Programmable Network Switch System Component Diagram

3.5. IP Flow Information Export (IPFIX) Protocol

IPFIX (Internet Protocol Flow Information Export) is a standard protocol designed for the collection and export of network flow information. It operates on a client-server model, where network devices generate flow records containing information about network traffic, such as source and destination IP addresses, ports, packet counts, and timestamps. These flow records are then exported to a collector server using IPFIX messages, enabling centralized monitoring and analysis of network traffic. The main purpose of IPFIX is to provide a standardized method for efficiently exporting flow data from various network devices, facilitating network traffic analysis, troubleshooting, and capacity planning. The IPFIX protocol is standardized in RFC 7011. [2]

Information

This section exclusively describes the IPFIX Metering and IPFIX Exporting Process which are in the responsibility of the network virtualisation system. Refer to chapter 4 for elaboration details about the IPFIX Collecting Process.

Metering Process The Metering Process generates Flow Records. Inputs to the process are packet headers, characteristics, and Packet Treatment observed at one or more Observation Points. The Metering Process consists of a set of functions that includes packet header capturing, timestamping, sampling, classifying, and maintaining Flow Records. The maintenance of Flow Records may include creating new records, updating existing ones, computing Flow statistics, deriving further Flow properties, detecting Flow expiration, passing Flow Records to the Exporting Process, and deleting Flow Records RFC 7011. [2]

Exporting Process The Exporting Process sends IPFIX Messages to one or more Collecting Processes. The Flow Records in the Messages are generated by one or more Metering Processes RFC 7011. [2]

As can be seen in figure 3.1 IPFIX is used to export network telemetry data from the Programmable Network Switch (Exporting Process) to the Monitoring System (Collecting Process). To facilitate the exportation of network telemetry data using IPFIX, a standardized format known as Templates is indispensable, as well as caches which are essential components of the Exporting Process, serving as temporary storage for Flow Records.

3.5.1. Templates

As specified in RFC 7011 a Template is an ordered sequence of <type, length> pairs used to completely specify the structure and semantics of a particular set of information that needs to be communicated from an IPFIX Device to a Collector. Each Template is uniquely identifiable by means of a Template ID. [2]

Information

In the two sections below the templates are explained independently from each other. In the actual implementation though the two Templates are exported in one Template Set.

3.5.1.1. Aggregated Data Export

As specified in the functional requirements in section 2.1 in US7 the flow statistics shall be made available to a network operator emphasizing the capability of the generation of a heat map containing endpoint to endpoint network efficiency information. For that purpose the Template with the ID 256 was elaborated to export and collect the relevant data. The Template structure is depicted in listing 3.1.

The following two fields are introduced in the context of this bachelor thesis and are directly related to the collection of network energy efficiency data. They are not registered by IANA by the time of writing this thesis.

indicatorID Identification number of the energy efficiency indicator being collected. It is needed in order to distinguish different efficiency indicators and to know how to interpret the indicatorValue. The value corresponds to the Auxil-data Node-ID field of the Aggregation Trace Option for IOAM specified in draft-cxx-ippm-ioamaggr-00. [3]

indicatorValue Value of the collected indicator. The value corresponds to the Aggregate field of the Aggregation Trace Option for IOAM specified in draft-cxx-ippm-ioamaggr-00. [3]

indicatorAggregator Aggregator used to calculate the indicatorValue on transit. The value corresponds to the Aggregator field of the Aggregation Trace Option for IOAM specified in draft-cxx-ippm-ioamaggr-00. [3]

ioamAggrFlag1Count Counts the number of packets of a flow which have the flag1 set in the IOAM header. The flag is used to indicate that the specified **aggregator was unsupported** on a node on the path. The flag is part of the Aggregation Trace Option for IOAM specified in draft-cxx-ippm-ioamaggr-00. [3]

ioamAggrFlag2Count Counts the number of packets of a flow which have the flag2 set in the IOAM header. The flag is used to indicate that the specified **data param was unsupported** on a node on the path. The flag is part of the Aggregation Trace Option for IOAM specified in draft-cxx-ippm-ioamaggr-00. [3]

ioamAggrFlag3Count Counts the number of packets of a flow which have the flag3 set in the IOAM header. The flag is used to indicate that the specified **namespace was unsupported** on a node on the path. The flag is part of the Aggregation Trace Option for IOAM specified in draft-cxx-ippm-ioamaggr-00. [3]

ioamAggrFlag4Count Counts the number of packets of a flow which have the flag4 set in the IOAM header. The flag is used to indicate that an **other error** occurred on a node on the path. The flag is part of the Aggregation Trace Option for IOAM specified in draft-cxx-ippm-ioamaggr-00. [3]

Listing 3.1: Template Set for Aggregated Data Export

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Set ID = 2          |       Length = 68             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Template ID 256        |       Field Count = 10        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|     flowLabelIPv6 = 31      |       Field Length = 4        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|    sourceIPv6Address = 27   |       Field Length = 16       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0| destinationIPv6Address = 28 |       Field Length = 16       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|   sourceTransportPort = 7   |       Field Length = 2        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|   dest.TransportPort = 11   |       Field Length = 2        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|     indicatorID = 5050      |       Field Length = 4        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|    indicatorValue = 5051    |       Field Length = 8        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|  indicatorAggregator = 5052 |       Field Length = 1        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|  ioamAggrFlag1Count = 5053  |       Field Length = 8        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|  ioamAggrFlag2Count = 5054  |       Field Length = 8        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|  ioamAggrFlag3Count = 5055  |       Field Length = 8        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|  ioamAggrFlag4Count = 5056  |       Field Length = 8        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|    packetDeltaCount = 2     |       Field Length = 8        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0| flowStartMilliseconds = 152 |       Field Length = 8        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|  flowEndMilliseconds = 153  |       Field Length = 8        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

For the definition of the following fields refer to RFC 7011. [2]

- Set ID
- Length
- Template ID
- Field Count
- Field Length

For the definition of the following fields refer to IANA IP Flow Information Export (IPFIX) Entities. [10]

- flowLabelIPv6
- sourceIPv6Address
- destinationIPv6Address
- packetDeltaCount
- flowStartMilliseconds
- FlowEndMilliseconds

3.5.1.2. Raw Data Export

As specified in the functional requirements in section 2.1 in US5 the path statistics shall be made available to a network operator emphasizing the identification possibilities of inefficient paths and nodes within a network. The Template used for Aggregated Data Export lacks the ability to identify the path, the measured indicator value is related to. For that purpose the Template with the ID 257 was elaborated to perform a raw data export as proposed in draft-spiegel-ippm-ioam-rawexport-07. [14]

Information

The "Raw export of IOAM data" mode enables nodes to export received IOAM data without interpretation, aggregation, or reformatting, facilitating a decoupled operational model from the encapsulation, updating, and decapsulation processes (IOAM data-plane operation). This separation of concerns allows nodes focused on data-plane operations to offload the interpretation task to specialized IOAM data processing systems, ensuring scalability and efficient handling of IOAM telemetry. The IOAM node handles data-plane operations, while the IOAM data processing system interprets, aggregates, and formats the IOAM data for further analysis, with potential scalability to export data to multiple processing systems. For more information refer to draft-spiegel-ippm-ioam-rawexport-07. [14]

The structure depicted in listing 3.2 is based on the example *Fixed Length IP Packet* in draft-spiegel-ippm-ioam-rawexport-07. [14] IOAM data is part of the *ipHeaderPacketSection*. The field *ioamReportFlags* is introduced by the draft and is not registered by IANA by the time of writing this thesis.

Listing 3.2: Template Set for Raw Data Export

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Set ID = 2          |       Length = 24             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        Template ID 257        |       Field Count = 4         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|    ioamReportFlags = 5060   |       Field Length = 1        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|    forwardingStatus = 89    |       Field Length = 1        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0| sectionExportedOctets = 410 |       Field Length = 2        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0| ipHeaderPacketSection = 313 |       Field Length = 96       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

For the definition of the following fields refer to RFC 7011. [2]

- Set ID
- Length
- Template ID
- Field Count
- Field Length

For the definition of the following fields refer to IANA IP Flow Information Export (IPFIX) Entities. [10]

- forwardingStatus
- sectionExportedOctets
- ipHeaderPacketSection

3.5.2. Caches

In IPFIX, there are different strategies on how to export telemetry data about a network flow. Some of them are based on a per flow packet sampling followed by a direct export not requiring a local cache. Others are based on the aggregation of flow data during the lifetime of a flow followed by an export on flow expiry. In order to perform aggregation on the exporting device, a cache-like memory must be available.

An other scenario where caches are required in IPFIX is the batch processing of multiple records at a time. An individual set may contain multiple records and in order to be able to export multiple records at a time in the same IPFIX message, a local buffer is required.

As introduced earlier in the IPFIX templates description, we introduced the two types of export which require a different caching approach.

Aggregated Export Caching The cache has one entry for each flow which is identified with the flow key. During the flow life time the values in the entries are updated (aggregated) with the corresponding aggregator. The export is triggered on flow expiry which means when there was no packet received of a particular flow for a predefined amount of time.

Raw Export Caching The raw export caching does not rely on aggregation. Instead the whole or part of the IP header of a packet is exported directly. The export is triggered based on a preconfigured sampling rate on a per flow basis. The caching is in this case only required in case batch processing (export of multiple records in one IPFIX message) should be done.

3.6. Design Decisions

- In the context of the network telemetry data export system, facing the need to transfer network telemetry data associated to a network flow to a remote system, we decided for the IP Flow Information Export (IPFIX) Protocol and against the usage of a custom HTTP API or messaging system, to achieve that the solution is natively supported on network devices and that already existing IPFIX data collectors can be used, accepting that part of the IPFIX protocol needs to be implemented on the behavioral model software switches.

- In the context of the network telemetry data export system, facing the need to transport IPFIX messages, we decided to use the User Datagram Protocol as transport protocol and against the usage of the Transmission Control Protocol or Stream Control Transmission Protocol, to achieve that the solution is lightweight and straight forward to implement, accepting that there is no reliability guarantee that exported IPFIX messages are received by the IPFIX collector.

- In the context of the network telemetry data export system, facing the need to send IPFIX messages from the BMv2 software switch (IPFIX exporter) to the monitoring system (IPFIX collector), we decided to send the IPFIX messages via a network interface outside of the isolated Mininet environment and against the transmission of IPFIX messages within Mininet, to achieve that the monitoring system can be setup and operated completely independent outside of Mininet, accepting that two independent systems need to be operated in order to run the infrastructure.

- In the context of the network telemetry data export system, facing the need to add a custom extern function which is callable from the P4 data plane program, we decided to implement the IPFIX export functionality in a decoupled software module which is added as a shared object to the BMv2 software switch, and against the implementation of the new feature as a part of the current BMv2 software, to achieve as much independence of the existing software project as possible and to reduce the time required for compilation because there is no need to recompile the whole project when modifications to the IPFIX extension are made, accepting that the startup configuration of the BMv2 software switches needs to be modified so that the IPFIX extension is loaded.

- In the context of the network telemetry data export system, facing the need to implement the IPFIX extension in C++, we decided to do the implementation in procedural C-like style, and against an object oriented approach in combination with the usage of advanced C++ features, to achieve a fast learning curve in writing C++ code, accepting that the implementation does not comply with C++ best practices, which is acceptable because the written code is for demo purposes only and will never run in production.
- In the context of the network telemetry data export system, facing the need to export data which can be used to generate a heatmap showing endpoint to endpoint energy efficiency, we decided to cache and aggregate all telemetry data associated to a network flow during the lifetime of a flow locally followed by an export of the aggregated data on flow expiry, and against a direct export of network telemetry data in combination with a packet sampling, to achieve the possibility to include the network telemetry data of all packets of a flow and to be able to do batch processing by exporting multiple flows together which expired during the same period of time, accepting that more local memory resources and dedicated cache management are required.
- In the context of the network telemetry data export system, facing the need to export data which can be used to get energy efficiency information on path level, we decided to additionally do a sampled raw export of the IPv6 header including all extension headers on a per flow basis and against the addition of path information to the aggregated flow export, to achieve the opportunity to retrieve a maximum amount of information out of the network by combining the data from the raw export and the aggregated export on the collector, accepting that extra development effort is required to implement an additional export mechanism.
- In the context of the network telemetry data export system, facing the need to do batch processing by exporting multiple flow records within a single IPFIX message, we decided to implement a reactive message splitting algorithm which splits a message into multiple messages in case it is too large, and against the static configuration of the maximum transmission unit (MTU), to achieve that too large messages are handled correctly by the exporting process, accepting that the maximum message size needs to be determined dynamically.
- In the context of the network telemetry data export system, facing the need to discover expired flow records, trigger the export of the expired flow records and to clean up the cache by deleting exported flow records, we decided to use a detached background thread which executes a function that takes care of the discovery, export and deletion of expired flow in a configurable interval of time, and against handing over this task to the extern function which is called for every packet being processed, to achieve performant and reliable cache management, accepting that a detached background thread needs to be executed with no possibility to join nor to terminate it without the shutdown of the software switch itself.
- In the context of the network telemetry data export system, facing the need to discover expired flow records, we decided that a flow is considered expired if the reception of the last packet of a specific flow was longer than a predefined amount of time ago and against the export when reaching a specific amount of packets, to achieve that flows of arbitrary length are exported properly, accepting that the timestamp of the reception of the last packet must be stored in the flow record cache.

- In the context of the network telemetry data export system, facing the need to export flow records in an efficient manner, we decided to use batch processing for the export of aggregated flow records which expired during the same period of time and against the transmission of each flow record in a single IPFIX message, to achieve a more efficient processing by saving bandwidth and compute overhead overhead on export of import a huge amount of individual messages, accepting a slightly more complex implementation due to the necessity of handling message of a dynamic size.

- In the context of the network telemetry data export system, facing the need to export flow records in an efficient manner, we decided to send one raw flow record per IPFIX message in the current implementation and against batch processing, to achieve memory savings because there is no need to cache the huge raw flow records and to get more accurate results in the collecting process in the means of consensus of the message timestamp and the time when the network telemetry data was captured, accepting the overhead of the transmission of a single flow record in a dedicated IPFIX message.

- In the context of the network telemetry data export system, facing the need to have the ability to transmit multiple raw records in a single IPFIX message for future performance optimizations through intelligent batch processing, we decided to implement the raw flow record exporter to take a list of raw records as input, to achieve the readiness to create IPFIX messages containing multiple raw flow records in a future version, accepting the slightly more complex implementation.

- In the context of the traffic generator, facing the need to to generate realistic network traffic within the network virtualization system, we decided to develop our own traffic generator using Python and Scapy and against using an existing traffic generator such as Cisco's traffic generator named TRex, to achieve that the traffic generator is adapted to our needs, accepting that additional development effort is required to implement the traffic generator.

4. Monitoring System

This chapter explains the concept of the monitoring system. As specified in the functional requirements in chapter 2 in the inception part in US5, US6, and US7, it is necessary to build a comprehensive monitoring system that shows several visualizations that provide insight into the network efficiency statistics. It is crucial to have significant network efficiency statistics because without them, it is difficult to gain insight into the huge amount of network efficiency telemetry data. Each component of the monitoring system runs in a separate Docker container for consistency and isolation. Docker ensures that each application runs the same everywhere, so deployment issues are reduced, what is necessary for the monitoring system because the deployment has to run on various hosts.

Figure 4.1 is a C4 container diagram which illustrates the composition of the monitoring system. The monitoring system is composed of the three main components IPFIX collector, time series database and a monitoring dashboard.

IPFIX Collector receives the IPFIX templates and exported network efficiency data from the network virtualization system.

Timeseries Database stores the time-stamped network efficiency data.

Monitoring Dashboard provides insights into system performance, metrics and trends.

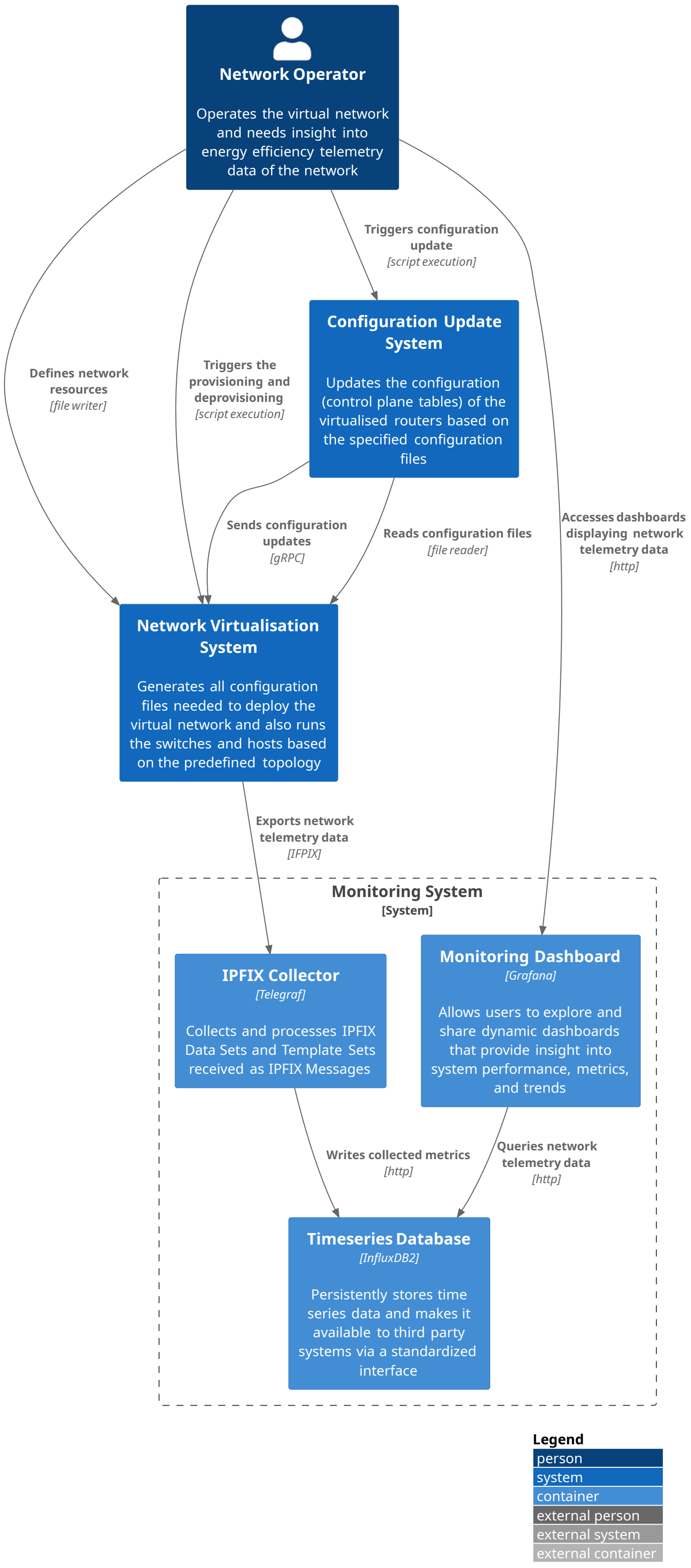

Network Operator
Operates the virtual network and needs insight into energy efficiency telemetry data of the network
Triggers configuration update
[script execution]
Configuration Update System
Updates the configuration (control plane tables) of the virtualised routers based on the specified configuration files
Defines network resources
[file writer]
Triggers the provisioning and deprovisioning
[script execution]
Sends configuration updates
[gRPC]
Reads configuration files
[file reader]
Accesses dashboards displaying network telemetry data
[http]
Network Virtualisation System
Generates all configuration files needed to deploy the virtual network and also runs the switches and hosts based on the predefined topology
Exports network telemetry data
[IFPIX]
Monitoring System
[System]
IPFIX Collector
[Telegraf]
Collects and processes IPFIX Data Sets and Template Sets received as IPFIX Messages
Monitoring Dashboard
[Grafana]
Allows users to explore and share dynamic dashboards that provide insight into system performance, metrics, and trends
Writes collected metrics
[http]
Queries network telemetry data
[http]
Timeseries Database
[InfluxDB2]
Persistently stores time series data and makes it available to third party systems via a standardized interface
Legend
person
system
container
external person
external system
external container

Figure 4.1.: Monitoring System Container Diagram

4.1. IPFIX Collector

An IPFIX collector is responsible for receiving flow record packets, acquiring data from the flow records, preprocessing, and transmit the flow record to a various destination. We decided to use Telegraf as IPFIX collector. Telegraf is a versatile open-source agent developed and maintained by InfluxData and integrates seamlessly with InfluxDB for storing time series data. The advantage of Telegraf is that it has a plugin based configuration setup. That means Telegraf includes *input plugins* to collect data from various sources, *processing plugins* to filter, normalize or transform the collected metrics and *output plugins* writes the metrics to various destinations. The Telegraf agent runs in a dedicated Docker container and loads the customized *telegraf.conf* file during start up. The configuration and deployment process is further described in the construction part in the chapter 2 in the construction part.

4.2. Time Series Database

A time series database (TSDB) is a specialized type of database optimized for storing and querying time-stamped data points such as the export of network efficiency data from the IPFIX export. We decided to use InfluxDB as time series database. The InfluxDB runs in a dedicated Docker container and the data is stored persistently in Docker volumes. InfluxDB allows to add custom configuration options with environment variables and allows to run a custom script that configures e.g. InfluxDB via Influx CLI during the startup process. The configuration and deployment process is further described in the construction part in the chapter 2 in the construction part.

4.2.1. Query Language

Flux Query Language (Flux) is a functional data scripting language designed by InfluxData for querying data from the InfluxDB time series database.

4.2.2. Buckets

A bucket is a named location where time series data is stored within the InfluxDB. [9] The network virtualization system performs two types of IPFIX exports, further described in detail in section 3.5. The two different IPFIX export types are necessary because the aggregated data export lacks the ability to identify the path, the measured indicator value is related to. For that purpose the raw data export was elaborated. To differentiate the two types of exported telemetry data in a simple and structured way, two buckets are created.

Aggregated Data Export stores the efficiency data from the aggregated data export

Raw Data Export stores the efficiency data from the raw data export

4.3. Monitoring Dashboard

As specified in the functional requirements in chapter 2 in US13 in the elaboration part the exported efficiency data from the network virtualization system should be visualized in several dashboards. We decided to use Grafana as monitoring software to build comprehensive visualizations. Grafana runs in a dedicated Docker container and the data is stored persistently in Docker volumes. During the start up Grafana will load the configuration, the data sources and the predefined dashboards. The configuration and deployment process is further described in the construction part in chapter 2 in the construction part.

4.3.1. Dashboards

We decided to create four dashboards, which are grouped according to the content of each dashboard. For each statistic, the most appropriate type of visualization is chosen.

Flow Statistics shows flow relevant statistics (based on aggregated data export).

- Network Flow Efficiency
- End to End Flow Efficiency Matrix
- Flow Efficiency by Receiver
- Flow Efficiency Indicator Distribution
- Most Inefficient Flow by Destination

Hop Statistics shows hop relevant statistics (based on raw data export).

- Efficient Hop Discovery
- Inefficient Hop Discovery
- Minimum Hop Transition Cost per Path
- Maximum Hop Transition Cost per Path

Path Statistics shows path relevant statistics (based on raw data export).

- Network Path Efficiency
- PEI Distribution
- Current Path Statistic from Host to Host
- PEI Statistics
- Path Efficiency over Time

Simulation Network Statistics shows simulation network topology relevant statistics (based on aggregated and raw data export).

- Aggregator Used for Percentage of Packets
- Number of Packets per Flow Distribution
- Number of Packets by Destination
- Number of Packets with Unsupported Aggregators
- Number of Packets per Flow with Unsupported Aggregators
- Number of Packets with Unsupported Data Parameter
- Number of Packets per Flow with Unsupported Data Parameter
- Number of Packets with Unsupported Namespace
- Number of Packets per Flow with Unsupported Namespace
- Number of Packets with Other Errors
- Number of Packets per Flow with Other Errors
- Flow Error Overview

4.4. Design Decisions

- In the context of the monitoring system, facing the need to deploy the monitoring system on various hosts, we decided to use Docker containers and against a native development for a specific platform, to achieve that the system runs the same everywhere and that the system is simple to start, stop and clean, accepting the limitations and challenges of containerization, such as the more complex topic of persistent data storage and data loss when shutting down the environment.
- In the context of the monitoring system, facing the need to use an IPFIX collector to receive the exported network efficiency data from the network virtualization system, we decided to use Telegraf as our IPFIX collector and against another IPFIX collector and analyzer tools because Telegraf is a server based agent that is written in Go and has no external dependencies and is a common tool used in combination with the Influx time series database to achieve an individual and configurable Telegraf agent that uses specific plugins for our use case to run as an IPFIX collector accepting that the slightly more complex implementation because we have to evaluate and configure the necessary Telegraf plugins.
- In the context of the monitoring system, facing the need to store the exported network efficiency data in a time series database, we decided to use InfluxDB and against another time series database such as Prometheus to achieve a performant and well-working system in combination with the Telegraf IPFIX collector, that is developed and maintained by the same company accepting that we have to understand and learn the query language Flux, that is a functional data scripting language of InfluxDB for querying, analyzing and acting on data.
- In the context of the monitoring system, facing the need to visualize the collected network efficiency data, we decided to use Grafana and against any other monitoring dashboard software to achieve a performant and well-working monitoring dashboard system in combination with the InfluxDB as data source accepting that Grafana uses substantial resources in data-rich environments.

5. Configuration Update System

According to user story 14 there is a need to have the possibility to update the control plane table definitions of the BMv2 software switches at runtime. The possibility to modify the configuration at runtime allows to do more sophisticated simulations as network paths and efficiency indicator related parameters may be changed on demand.

The configuration update system is designed to read the JSON configurations of the BMv2 software switches and to upload the configurations to the specific software switch. In the use case of this bachelor thesis the configuration update system is used hand in hand with the configuration generator which is part of the network virtualization system.

i Information

In general the configuration update system is designed to be usable with any other project using the BMv2 software switches. We consider to publish the configuration update system on GitHub under an open source license so that it can be used by other P4 developers.

5.1. Requirements

The configuration update system needs to fulfill one specific functional requirement.

Modification of Parameters at Switch Runtime In order to be able to simulate a realistic scenario the energy efficiency values and paths in the network must be changed from time to time. This update process must be possible without the need to re-initialize the virtual network.

5.2. Configuration Update Process

The configuration update process is run through when the *Network Virtualization System* is already running and the configuration of the BMv2 software switches needs to be modified. The first two steps are to prepare the updated BMv2 configuration files and are not directly related to the configuration update system. In order to follow the configuration update refer to figure 3.4.

1. The **network operator** modifies the **resource definition**.
2. The **network operator** manually triggers the **configuration generator** so that the latest **BMv2 configurations** are available.
3. The **network operator** manually triggers the **configuration updater**.
4. The **configuration updater** connects to each individual **programmable network switch** and uploads the specific **BMv2 configuration**.

5.3. Design Decisions

- In the context of the configuration update system, facing the need to transfer configurations stored locally to the software switches, we decided to use the **p4 runtime library** which is already in use as part of the initial configuration process at start up, and against the usage of the **p4 runtime shell** with a custom configuration update implementation, to achieve a consistent configuration state on configuration update with a straight forward implementation strategy, accepting that partial configuration updates are not possible.
- In the context of the configuration update system, facing the need to specify an inventory, connection parameters, user interface etc., we decided to use the **Nornir automation framework**, to achieve the availability of commonly used network automation functionalities out of the box and decrease the development effort, and neglected the implementation of our own automation solution, accepting the external dependency.

Part III.

Construction

1. Network Virtualization System

This chapter contains implementation details of all related components inside the network virtualization system.

1.1. Resource Definition

One of the requirements for the configuration update system is, that there is a layer of abstraction between the configuration files and the place where the relevant information is set. Additionally the information is only specified once in a structured declarative format. We decided to define the resources in a dedicated file using the YAML format which is then used as the single source of truth to generate and update the configuration of the virtual network.

At the top of the resource definition the path to the P4 data plane program is specified which should be loaded on the programmable switches. Afterwards the resource file contains the paths followed by the hosts and the switches and finally there is some meta information about the tables to configure. More information about the specific parts are available in the following sections.

1.1.1. Path Specification

One essential part of the resulting configuration on the software switches are the control plane table entries used for IPv4 and IPv6 forwarding. The setup of these tables by hand is very time intensive even for a small topology. For each table entry the correct outgoing port and destination MAC address must be specified.

To be able to dynamically reconfigure the network paths there must be an abstraction to specify the paths with as little overhead as possible. A path entry takes four values as shown in listing 1.1.

from Is the name of the originating host.

to Is the name of the destination host.

via Is an ordered list of hops a packet traverses when traveling from the source to the destination.

return_route A boolean flag which indicates whether a symmetric path in the opposite direction shall be created. With this option only half of the paths need to be specified explicitly. The remaining paths are implicit.

Listing 1.1: Path Specification Example

```yaml
paths:
  - from: h01
    to: h02
    via: [s11, s01, s12]
    return_route: true
```

1.1.2. Host Specification

To automatically add hosts to the virtual network they have to be specified in the format shown in listing 1.2. The host configuration includes the following information:

ipv4 A dictionary with the information relevant to configure the IPv4 address and routing table on the host.

ipv6 A dictionary with the information relevant to configure the IPv6 address and routing table on the host.

mac The MAC address of the host.

commands A list of commands which should be executed on host startup.

Listing 1.2: Host Specification Example

```
hosts:
  h01:
    ipv4:
      ip: 146.185.64.10
      net: 146.185.64.0
      prefix_len: 19
    ipv6:
      ip: 2a04:f340::a
      net: "2a04:f340::"
      prefix_len: 29
    mac: 08:FF:00:00:00:01
    commands:
      - "route add default gw 146.185.64.1 dev eth0"
      - "arp -i eth0 -s 146.185.64.1 08:EE:00:00:00:11"
      - "python3 ./dev-network/utils/testing/packet_generator.py --ipv6 --src 'h01'
    --infinite --startup-delay 15 --logfile &"
```

1.1.3. Switch Specification

The switch specification allows to:

1. Set generic switch specific settings

   mac The MAC address of the switch

2. Set energy efficiency indicator related parameters

   hei List of available indicator types including the identification number and the current value.

   ioam Dictionary with parameters to configure the network telemetry collection based on the IOAM protocol such as the active efficiency indicator (data param) or the list of active aggregators.

3. Define the network graph

   ports For each port the specific neighbor and the efficiency of the link is set and the node id.

Listing 1.3: Switch Specification Example

```
switches:
  s01:
    mac: 08:CC:00:00:00:01
    hei:
      - data_param: 255
        value: 1000
    ioam:
      node_id: 1
      aggregators: # 1 = SUM / 2 = MIN / 4 = MAX
        - 1 # selected if last two bits of payload size are [00]
        - 2 # selected if last two bits of payload size are [01]
        - 1 # selected if last two bits of payload size are [10]
        - 4 # selected if last two bits of payload size are [11]
      data_param: 255
    ports:
      1:
        neighbor: s02
        lei: 10
      2:
        neighbor: s03
        lei: 20
      3:
        neighbor: s04
        lei: 30
      4:
        neighbor: s17
        lei: 40
      5:
        neighbor: s18
        lei: 50
      6:
        neighbor: s11
        lei: 60
      7:
        neighbor: s12
        lei: 70
```

1.1.4. Tables Specification

In the tables specification section metadata about the control plane tables present in the BMv2 runtime are defined. An example is given in listing 1.4 where the table for the IPv4 forwarding table is defined.

The tables specification includes the following information:

name The name of the table to be defined.

action_name The name of the target action called in case an entry has matched.

default_action (optional) The name of the default action which is called in case there was no matching table entry. In case there is no default action this field can be omitted.

match_key The name of the field which is used during the match operation.

Listing 1.4: Tables Specification Example

```
tables:
  ipv4_forwarding:
    name: MyIngress.ipv4_lpm
    action_name: MyIngress.ipv4_forward
    default_action: MyIngress.drop
    match_key: hdr.ipv4.dstAddr
```

1.2. Configuration Generator

The configuration generator is written in Python. It reads the information specified in the resource definition, applies data structure reformatting based on dedicated logic further described in the following sections. Once the data structures are ready they are handed over to predefined Jinja2 templates where the custom configuration is brought into the format accepted by Mininet and the BMv2 software switches respectively.

1.2.1. Mininet Topology

The Jinja2 template used to craft the Mininet Topology is the *mininet_topology.j2* file. It defines the structure of the topology file used to specify the hosts, switches and links present in the network. It is read by the *Network Virtualization System* at startup. The switches and hosts, as defined in the resources file, can be passed to the Jinja2 template without any modifications. The links though are not explicitly defined in the resources file. They are implicitly defined as part of the properties of a switch by the specification of the port neighborships. The next section will go into more detail about the algorithm used for the link discovery.

1.2.1.1. Link Discovery Algorithm

The link discovery algorithm has to transform the port neighborship specification into a specification of links. Listing 1.5 contains some of the port specifications of s01, s02 and s03 which is the input for the link discovery algorithm.

Listing 1.5: Ports Specification (Algorithm Input)

```
s01:
    <-- details omitted -->
    ports:
      1:
        neighbor: s02
        lei: 10
      2:
        neighbor: s03
        lei: 20
      3:
        neighbor: s04
        lei: 30
    <-- details omitted -->
  s02:
    <-- details omitted -->
    ports:
      1:
        neighbor: s01
        lei: 10
      2:
```

```
      neighbor: s04
      lei: 20
    3:
      neighbor: s03
      lei: 30
  <-- details omitted -->
s03:
  <-- details omitted -->
  ports:
    1:
      neighbor: s02
      lei: 10
    2:
      neighbor: s01
      lei: 20
  <-- details omitted -->
```

Listing 1.6 contains the resulting link specification of s01, s02 and s03 which is the output of the link discovery algorithm. The link specification is in JSON format and part of the Mininet topology definition.

Listing 1.6: Links Specification (Algorithm Output)

```
<-- details omitted -->
"links": [
  [
    "s01-p1",
    "s02-p1"
  ],
  [
    "s01-p2",
    "s03-p2"
  ],
<-- details omitted -->
```

The algorithm iterates over all switches in the given input. For each switch it iterates over all ports. If the neighbor of the current port is a switch and the switch number of the neighbor is greater than the local switch number, the port on the neighbor switch is discovered which connects to the local switch. Once the port has been discovered on the neighbor switch a new link is added with the local port of the local switch and the discovered port on the neighbor switch. In case a host is connected to the port of the local switch, the specific link is added right away as no port discovery is required on the remote host.

1.2.2. BMv2 Runtime

The Jinja2 template used to craft the BMv2 runtime is the *bmv2_runtime.j2* file. It defines the structure of the BMv2 control plane table specification. It is read by the *Network Virtualization System* at startup. The tables specification explained in 1.1.4 as well as configuration parameters for the switches, can be passed to the Jinja2 template without any modifications. The forwarding information though is specified in an abstracted format using paths described in section 1.1.1. Before the Jinja2 template can be applied the abstract forwarding information specified as paths must be translated into concrete forwarding entries containing MAC addresses and egress ports for each individual switch on the path. The next section will go into more detail about the algorithm used for the forwarding information translation algorithm.

1.2.2.1. Forwarding Information Translation Algorithm

To clarify the operation of the forwarding information translation algorithm we look at how the abstract path definition in listing 1.7 is translated into forwarding information which can be uploaded to configure the control plane of the BMv2 software switches. Refer to figure 3.3 in the elaboration part.

Listing 1.7: Example Abstract Path Definition

```
paths:
  - from: h01
    to: h02
    via: [s11, s01, s12]
    return_route: true
```

As with the algorithm described in section 1.2.1.1 the algorithm's job is to transform the abstract data specified in the resource yaml file in an extended data structure which can be used by the Jinja2 templating engine to generate the configuration file.

i Information

The listings 1.8, 1.9 and 1.10 only contain the information relevant to explain the algorithm. The only forwarding table entry contained, is the one for *h02*. Also other switch specific information is omitted.

The elements present in the extended data structure as the output of the translation algorithm are:

ip The destination IPv4 or IPv6 address prefix. Used as part of the key in the control plane table.

mac The MAC address of the next hop. Used as a value in the control plane table. This value is used by the data plane to update the destination MAC address header field.

port The outgoing port for the forwarding operation. Used as a value in the control plane table. This value is used by the data plane to update egress port as part of the standard metadata.

prefix_len The length of the prefix defined in the ip field. Used as part of the key in the control plane table.

route_type Specifies the type of route. Used as a value in the control plane table. This value is used by the data plane to determine whether the switch is an egress switch or not. In case it is an egress switch the network telemetry data is passed to IPFIX exporter. Additionally IOAM header fields could be stripped off which is currently not implemented as it is not required for demo purposes and it would reduce troubleshooting capabilities. The possible route types are:

0 Directly connected route (identifies the switch to be an egress)

1 Static route

Listing 1.8: s01 Extended Datastructure

```
's01': {
  'ipv4_forwarding': {
    'h02': {
      'ip': '31.24.8.0',
      'mac': '08:EE:00:00:00:12',
      'port': 7,
      'prefix_len': 21,
      'route_type': 1}
    },
    'ipv6_forwarding': {
      'h02': {
        'ip': '2a00:10c0::',
        'mac': '08:EE:00:00:00:12',
        'port': 7,
        'prefix_len': 32,
        'route_type': 1}
  }
}
```

Listing 1.9: s11 Extended Datastructure

```
's11': {
  'ipv4_forwarding': {
    'h02': {
      'ip': '31.24.8.0',
      'mac': '08:CC:00:00:00:01',
      'port': 1,
      'prefix_len': 21,
      'route_type': 1}
  }
  'ipv6_forwarding': {
    'h02': {
      'ip': '2a00:10c0::',
      'mac': '08:CC:00:00:00:01',
      'port': 1,
      'prefix_len': 32,
      'route_type': 1}
  }
}
```

Listing 1.10: s12 Extended Datastructure

```
's12': {
  'ipv4_forwarding': {
    'h02': {
      'ip': '31.24.8.10',
      'mac': '08:FF:00:00:00:02',
      'port': 3,
      'prefix_len': 32,
      'route_type': 0}
  },
  'ipv6_forwarding': {
    'h02': {
      'ip': '2a00:10c0::a',
      'mac': '08:FF:00:00:00:02',
      'port': 3,
      'prefix_len': 128,
      'route_type': 0}
  }
}
```

In order to get from the abstract path definition to the extended data structures the algorithm iterates over all switches and for each switch it iterates over all paths for each path it performs the following steps:

1. Verifies if the switch is part of the current path. If not it continues with the next path.
2. If the switch is part of the path it attempts to generate an IPv4 and IPv6 forwarding entry.
3. Loads the information of the destination host and the current switch from the resources yaml file.
4. Verifies if the switch is the last hop.
5. It initializes the following fields:

   ip Sets the destination ip prefix of the specified host. In case of a directly connected route the actual IP of the host is set else the network prefix is set.

   mac If the switch is the last hop the MAC address of the host is set. Else the MAC address of the next hop is set based on the next entry in the hop list in the abstract path definition.

   port Determined by looping over the local ports and finding the port where the neighbor matches the next hop.

   prefix_len Set according to the specification in the resources yaml file. In case of a directly connected route the prefix is set to /32 in the case of IPv4 and to /128 for IPv6.

   route_type If the switch is the last hop the table entry is initialized with a route type of 0 else with a route type of 1.

ℹ Information

Be aware that information in the extended datastructures not present in the abstract path definition, are taken from the resource definition yaml file.

Once the data structures are initialized as in listings 1.8, 1.9 and 1.10 the Jinja2 template is applied to produce the following output. Listing 1.11 shows how the final IPv4 and IPv6 table entries will look like given the example input specified in listing 1.7

Listing 1.11: s01 IPv4 and IPv6 Table Entries

```
<-- details omitted -->
{
  "table": "MyIngress.ipv4_lpm",
  "match": {
    "hdr.ipv4.dstAddr": [
      "31.24.8.0",
      21
    ]
  },
  "action_name": "MyIngress.ipv4_forward",
  "action_params": {
    "mac": "08:EE:00:00:00:12",
    "port": 7,
    "route_type": 1
  }
```

```
    },
    <-- details omitted -->
    {
      "table": "MyIngress.ipv6_lpm",
      "match": {
        "hdr.ipv6.dstAddr": [
          "2a00:10c0::",
          32
        ]
      },
      "action_name": "MyIngress.ipv6_forward",
      "action_params": {
        "mac": "08:EE:00:00:00:12",
        "port": 7,
        "route_type": 1
      }
    },
    <-- details omitted -->
```

1.2.3. Traffic Generator Configuration

As described in section 3.3 in the elaboration part the traffic generator uses a config file that is generated based on the current resource yaml file. The config file is regenerated each time the network virtualization system is started to support a variety of simulation topologies. The following two elements are created within the *packet-generator-config.json* file for each host that is specified in the topology resource yaml file of the simulation network that is loaded. As can be seen in listing 1.12 a mapping from source host to all possible destination hosts is created and in listing 1.13 for each host, the relevant forwarding information such as source mac, destination mac (of the next hop), ipv4 and ipv6 addresses are specified.

Listing 1.12: Mapping Source host to possbile destination hosts

```
<-- details omitted -->
"dst_hosts": {
  "h01": [
    "h02",
    "h03",
    "h04",
    "h05",
    "h06",
    "h07",
    "h08",
    "h09",
    "h10",
    "h11"
  ],
<-- details omitted -->
```

Listing 1.13: Source Host Information

```
<-- details omitted -->
"hosts": {
  "h01": {
    "src_mac": "08:FF:00:00:00:01",
    "dst_mac": "08:EE:00:00:00:11",
    "ipv4": "146.185.64.10",
    "ipv6": "2a04:f340::a"
  },
<-- details omitted -->
```

1.3. Traffic Generator

The traffic generator is written in Python and uses Scapy to craft and send network packets. The traffic generator runs on each Mininet host and generates simulated network traffic in the network virtualization system. The configuration is derived from the resource definition yaml file. In case the resource definition is modified the traffic generator will automatically reconfigure itself after a restart.

It depends on the following library:

Scapy Scapy is a powerful Python library used for packet manipulation, network scanning and network analysis. It allows users to create, send, receive and dissect network packets. Scapy is further described in the official Scapy documentation.

1.3.1. Functionality

The traffic generator sends a flow with a random number of packets to various hosts, whereby the flow label is always unique. The traffic generator runs for an infinite amount of time depending on the parameters given. The process of sending a new flow is always the same:

1. A random and valid destination host is chosen
2. Select a random number of packets to be sent in the flow (range 1 - 100 packets)
3. Generate a valid and unique flow label
4. Select a random source port from the ephemeral port range (49152 - 65535)
5. Send the flow to the destination host
6. Sleep a few seconds before sending a new flow (random selected sleep time, up to 30 seconds)

i Information - Flow Label

To ensure that a flow label is unique, it will be added to a *flow label* set to avoid using it more than once within a short period of time. Since our traffic generator is running infinite, the problem arises that we run out of flow labels because a flow label is a 20-bit number. To solve this problem, we have implemented a logic that randomly selects and checks if we have already used the flow identifier up to 10 times. If the random selection of a flow identifier has not worked after 10 times, the *flow label* set is cleared.

1.3.1.1. Command Line Parameters

The traffic generator is an automated solution to simulate realistic traffic in the simulation network. But its also necessary that the network operator can use the traffic generator manually with specific parameter options. To allow and support the specific use of the traffic generator the following command line parameters are implemented to ensure that targeted and specific traffic can be sent.

–ipv6 Send only IPv6 traffic

–src specifies the source host

–dst specifies the destination host

–flow-label set a specific flow label

–flow-count define number of flows to be sent

–packet-count define number of packets to be sent

–infinite specifies that the traffic generator will run for an infinite amount of time

–config path to json configuration file, default=*"./dev-network/traffic-generator-config.json"*

–log-dir specifies the log file directory, *default_logs = os.path.join(cwd, "logs")*

–logfile writes a log file (log level: warning)

–startup-delay defines a startup delay before sending traffic

1.3.2. Start Traffic Generator on Mininet Host

As specified in the functional requirements in chapter 2 in the inception part in US15 the traffic generator should generate traffic in the simulation network. The traffic generator starts with a delay of a configurable amount of seconds on all Mininet hosts in the simulation network topology and sends packets for an infinite amount of time to the other hosts. The host specification below from the *resource.yaml* file, further described in section 1.1 shows that in the *commands* option the traffic generator is started. The traffic generator needs the following parameters to start correctly on a Mininet host to send random flows to other host in the network for an infinite amount of time:

–ipv6 Send only IPv6 traffic

–src It's necessary to specify the hostname on which host the traffic generator is started to choose the correct and valid destination hosts

–infinite Specifies that the traffic generator runs for an infinite amount of time

–startup-delay The host starts sending packets after a delay of 15 seconds

–logfile A log file that logs *warnings* and *errors* is created

Listing 1.14: h01 specification in resource file

```
<-- details omitted -->
h01:
    ipv4:
      ip: 146.185.64.10
      net: 146.185.64.0
      prefix_len: 19
    ipv6:
      ip: 2a04:f340::a
      net: "2a04:f340::"
      prefix_len: 29
    mac: 08:FF:00:00:00:01
    commands:
      - "route add default gw 146.185.64.1 dev eth0"
      - "arp -i eth0 -s 146.185.64.1 08:EE:00:00:00:11"
      - "python3 ./dev-network/utils/testing/traffic_generator.py --ipv6 --src 'h01'
    --infinite --startup-delay 15 --logfile &"
<-- details omitted -->
```

Information

The ampersand (&) at the end of the command is needed to start the traffic generator in the background. If this option is not set, Mininet will stop at this point of startup process and will not continue because the command never terminates.

1.3.3. Storage Starvation Issue

We were facing the issue that we ran out of storage on the local development VM and also on the server. After disabling the logging and packet capture (creation of pcap files) of each interface on all Mininet hosts the problem with the lack of storage was better but after a few days nevertheless we run again out of storage. In the end, we found the problem. The root cause of the problem was the traffic generator, more precisely the library, Scapy that we use to craft and send packets. The problem was that each *eth0* interface on all Mininet hosts was permanently changing its interface state by entering and leaving promiscuous mode for each packet being sent. Each time an interface changes its state a log entry is written by syslog. We added this line of code to the traffic generator to set the interface promiscuous mode to *false* and the problem was fixed.

Listing 1.15: Disable Scapy Promiscious Mode

```
conf.sniff_promisc = 0
```

1.4. BMv2 IPFIX Extension

1.4.1. Challenges

As already mentioned in the project planning chapter and within the risk assessment, many challenges have been identified at the beginning of the project regarding the implementation of the BMv2 IPFIX extension. The most crucial ones are described in the following sections.

1.4.1.1. Exposal of Extern Function to Data Plane

To make the functionality implemented in the IPFIX extension available to the data plane it must be exposed using a P4 extern function. The setup of an extern function requires actions on both the data plane and the control plane.

In the BMv2 environment an extern function can be registered using the macro shown in listing 1.16. The macro takes an arbitrary number of positional arguments, whereas the first one is the name of the function to be exposed to the data plane and the following arguments, are the types and names of the arguments of the function to be exposed.

In the file *behavioral-model/externs/src/ipfix/cache.cpp* one finds the following code block which registers the extern function.

Listing 1.16: Extern Function Registration Control Plane

```
BM_REGISTER_EXTERN_FUNCTION(ProcessEfficiencyIndicatorMetadata,
                            const bm::Data &, const bm::Data &,
                            const bm::Data &, const bm::Data &,
                            const bm::Data &, const bm::Data &,
                            const bm::Data &, const bm::Data &,
                            const bm::Data &, const bm::Data &,
                            const bm::Data &, const bm::Data &);
```

On the data plane the extern function must be declared similarly it would be done in C or C++ header file to let the compiler know about the existence of concrete implementations of that function. In the file *externs.p4* which is included into the file *main.p4* one finds the following code block which declares the extern function.

Listing 1.17: Extern Function Registration Data Plane

```
extern void ProcessEfficiencyIndicatorMetadata(
                                        in ioamNodeID_t nodeID,
                                        in flowKey_t flowKey,
                                        in flowLabel_t flowLabel,
                                        in ip6Addr_t srcIPv6,
                                        in ip6Addr_t dstIPv6,
                                        in bit<16> sourceTransportPort,
                                        in bit<16> destinationTransportPort,
                                        in bit<24> indicatorID,
                                        in ioamAggregate_t indicatorValue,
                                        in bit<8> indicatorAggregator,
                                        in bit<768> raw_ipv6_header
                                    );
```

1.4.1.2. IPFIX Extension as BMv2 Plugin

In both Windows and Linux there is the possibility to dynamically load and link libraries into a statically compiled and linked project. In Windows this is achieved using dynamic linked libraries (DLL) files and in Linux with shared object (so) files. As we are working on Linux we compiled the IPFIX extension to an *so file* to add the functionality implemented in the IPFIX extension to the BMv2 project without the need to recompile the whole project.

The command used to compile the BMv2 IPFIX extension to a shared object is:

```
g++ -Wall -Wextra -g -O2 -fPIC -shared -o obj/ipfix.so <source files> -ltins
```

Especially important is the flag **-shared** provided to the linker which compiles the application to a shared object instead of a stand alone binary. To compile the extension please use the Makefile located in the externs directory. After compilation the generated shared object file can be loaded into the BMv2 software switches with the argument **–load-modules <path/to/so/file>**.

Each BMv2 software switch is started as a separate process based on the simple_switch_grpc binary. The following command is used on startup of the network virtualization system to start a new BMv2 switch.

```
simple_switch_grpc \
    -i 1@s01-eth1 \
    -i 2@s01-eth2 \
    -i 3@s01-eth3 \
    -i 6@s01-eth6 \
    -i 7@s01-eth7 \
    -i 4@s01-eth4 \
    -i 5@s01-eth5 \
    --pcap /home/boss/git/ba/efficiency-indicator-p4/pcaps \
    --nanolog ipc:///tmp/bm-0-log.ipc \
    --device-id 0 \
    build/main.json \
    --log-console \
    --thrift-port 9090 \
    -- \
    --grpc-server-addr 0.0.0.0:50051 \
    --load-modules /home/boss/git/ba/behavioral-model/externs/obj/ipfix.so
```

The double dash before *grpc-server-addr* indicates that the two arguments *grpc-server-addr* and *load-modules* are positional arguments rather than options for the simple_switch_grpc command. This is a way to ensure that these arguments are passed as intended without being parsed as options by the command-line parser. This is especially useful when the positional arguments might be mistaken for options or when arguments need to be passed to another command or script called by the main command.

1.4.1.3. Concurrent Access on Cache Data Structures

The cache data structures described in section 1.4.2 are accessed concurrently by multiple different threads. There are dedicated threads used for the export via IPFIX and one dedicated thread for each call of the extern function which update the cache with the values provided by the data plane.

To ensure data consistency in the IPFIX caches only one thread at a time is allowed to update the data structure. This constraint is implemented using a mutex to lock and unlock the cache data structures used in IPFIX. We used the mutex library which is part of the C++ standard library documented here: https://en.cppreference.com/w/cpp/thread/mutex.

Fortunately the data plane does not wait until the execution of the extern function is terminated. This means that the data forwarding is not impacted by potentially delayed executions of cache updates caused by the locked data structure.

1.4.1.4. Background Task for Regular Export

Aggregated flow records stored inside the cache and template records shall be exported on a regular basis. The discovery of expired flow records and the execution of the export task is handled by a thread which needs to run continuously.

This challenge is solved by the setup of a detached thread which executes a function which discovers expired records in a specified interval.

The background threads are started on the first execution of the extern function ProcessEfficiencyIndicatorMetadata. Listing 1.18 contains a code snippet out of the extern function which is responsible to start the background threads if not already started.

bg_threads_started Is a boolean variable which indicates whether the background threads have already been started.

flow_export_cache_manager Is a thread which executes the function to regularly discover and export expired aggregated flow records. The thread is being detached on line 11 to ensure it remains running after the termination of the current thread.

template_exporter Is a thread which executes the function to regularly export the template records. The thread is being detached on line 12 to ensure it remains running after the termination of the current thread.

Listing 1.18: Background Thread Start Snippet

```cpp
void ProcessEfficiencyIndicatorMetadata(
    const bm::Data &node_id,
    <-- details omitted -->) {

  // start background threads
  if (!bg_threads_started) {
    observation_domain_id = node_id.get_int();
    bg_threads_started = true;
    std::thread flow_export_cache_manager(ManageFlowRecordCache);
    std::thread template_exporter(ExportTemplates);
    flow_export_cache_manager.detach();
    template_exporter.detach();
  }
  <-- details omitted -->
}
```

1.4.1.5. IPFIX Message Transmission

As already identified as the technical risk (R3), the transmission of IPFIX messages using the BMv2 software switch harbours major challenges. There are two aspects which had to be clarified.

1. Craft and transmit new network packets using the BMv2 software switch.
2. Send packets to a system outside of Mininet.
3. Transmission of variable sized IPFIX message data which may exceed the maximum transmission unit MTU.

The first aspect could be clarified with the usage of an external library. The BMv2 software switch and so also the IPFIX extension are written in C++. After some research we found the C++ library libtins which is capable of crafting arbitrary network packets comparable to scapy in Python. More information about libtins is available here: https://github.com/mfontanini/libtins

The second aspect is supported by the Linux operating system out of the box. Mininet runs each of the switches as a separate process and interconnects them with virtual links. As each process has access to all network interfaces within the the same network namespace on the operating system an arbitrary network interface can be selected for the transmission of IPFIX messages.

Listing 1.19 contains a code snippet which shows how the IP packet is crafted and sent using the libtins library.

Line 4 The default interface is selected based on the routing table entries. If the default interface is selected, the interface that has configured the default route is used.

Line 5 The address configuration of the selected interface is retrieved.

Line 6-7 The IP packet carrying the IPFIX message is crafted with the following arguments:

IPFIX_COLLECTOR_IP Destination IP address

info.ip_addr Source IP address

4739 Destination UDP port

43700 Source UDP port

payload Pointer to the IPFIX message raw data

size Size of the IPFIX message raw data

Line 10 Transmission of packet

Listing 1.19: Send Function Snippet

```
1  int SendMessage(uint8_t *payload, size_t size) {
2    int result = ERR_OK;
3    BMLOG_DEBUG("IPFIX EXPORT: Sending IPFIX message");
4    NetworkInterface iface = NetworkInterface::default_interface();
5    NetworkInterface::Info info = iface.addresses();
6    IP packet = IP(IPFIX_COLLECTOR_IP, info.ip_addr) / UDP(4739, 43700) /
7                RawPDU(payload, size);
8    PacketSender sender;
9    try {
10     sender.send(packet, iface);
11   } catch (const Tins::socket_write_error &e) {
12     if (std::string(e.what()) == "Message too long") {
13       result = ERR_MESSAGE_TOO_LONG;
14     } else {
15       delete[] payload;
16       throw;
17     }
18   }
19   return result;
20 }
```

Regarding the third aspect mentioned above about the transmission of IPFIX messages which potentially exceed the MTU the following solution was found. As can be seen in listing 1.19 on line 11 a socket write error is intercepted. In case the error message corresponds to *Message too long* the error is set as the result on line 13. At the place where the SendMessage function is called the result is validated. In case the result indicates a *Message too long* error the IPFIX messages are split and stored in two separate payloads. Afterwards the SendMessage function is called twice, once with each payload. This is a recursive process. In case a message would still be too large the same process would happen and instead of two, four payloads would be generated.

1.4.2. Caches

In IPFIX, there are different strategies on how to export telemetry data about a network flow. Some of them are based on a per flow packet sampling followed by a direct export not requiring a local cache. Others are based on the aggregation of flow data during the lifetime of a flow followed by an export on flow expiry. In order to perform aggregation on the exporting device, a cache-like memory must be available.

An other scenario where caches are required in IPFIX is the batch processing of multiple records at a time. An individual set may contain multiple records and in order to be able to export multiple records at a time in the same IPFIX message, a local buffer is required.

1.4.2.1. Aggregated Data Export

The aggregated data export strategy is based on aggregation of flow data during the lifetime of a flow. It solves the following problems:

- Aggregation of flow data drastically decreases the demand of local caching capacity by storing the aggregated values only instead of the whole data series. It implies that the storage requirements for a single flow is constant independent from the number of packets the flow consists of.

- There is no need to implement sampling because the sampling rate is given implicitly by exporting the aggregated values on flow expiry.
- The batch processing of multiple flow records in a single export is done by design by exporting the flow records which expired within the same period of time simultaneously.

In order to be able to perform the aggregation, a local cache is needed which fulfills the following requirements:

- The collection of multiple different indicator types must be possible simultaneously without interference.
- Create, update and delete operations on flow records must be possible, given the indicator ID, aggregator and flow key.
- The data structure must be resistant against concurrent read and write operations by independent threads.
- On demand growing and shrinking of the data structure must be possible with very little overhead in regards to the reorganization of the data structure.

The data structure of the cache for the aggregated data export is depicted in figure 1.1. The data structure consists of two maps which are nested into each. The inner map stores a pointer to the specific flow record as a value. The flow record is stored on the heap so that it is accessible by different threads.

FlowRecordCacheIndex (outer map) Is allocated as a global variable on the stack. It stores the references to multiple FlowRecordCache data structures. It maps a given indicator ID (IOAM data param) concatenated with the aggregator to the FlowRecordCache containing the FlowRecords of that particular indicator ID and aggregator combination. With this outer map the simultaneous collection of multiple different indicator types and aggregators without interference is made possible.

FlowRecordCache (inner map) Is allocated as a global variable on the stack. It stores the references to FlowRecords. It maps a given flow label on a pointer to the FlowRecord data structure. The FlowRecord data structure complies to the definition of the *Template Set for Aggregated Data Export* specified in listing 3.1 in the elaboration part. The small empty gray boxes are place holders for other FlowRecordCaches which where omitted to simplify the illustration.

FlowRecord Is allocated on the HEAP and so made accessible to all threads of the current program.

HEAP The heap is made of reserved memory which can be used for dynamic memory allocation in a process. It is accessible by all threads of the specific process. It stores the actual flow records.

Information

A flow key uniquely identifies a flow and is the concatenation of the IPv6 source address and the flow label. The exact specification can be found in RFC 2460.

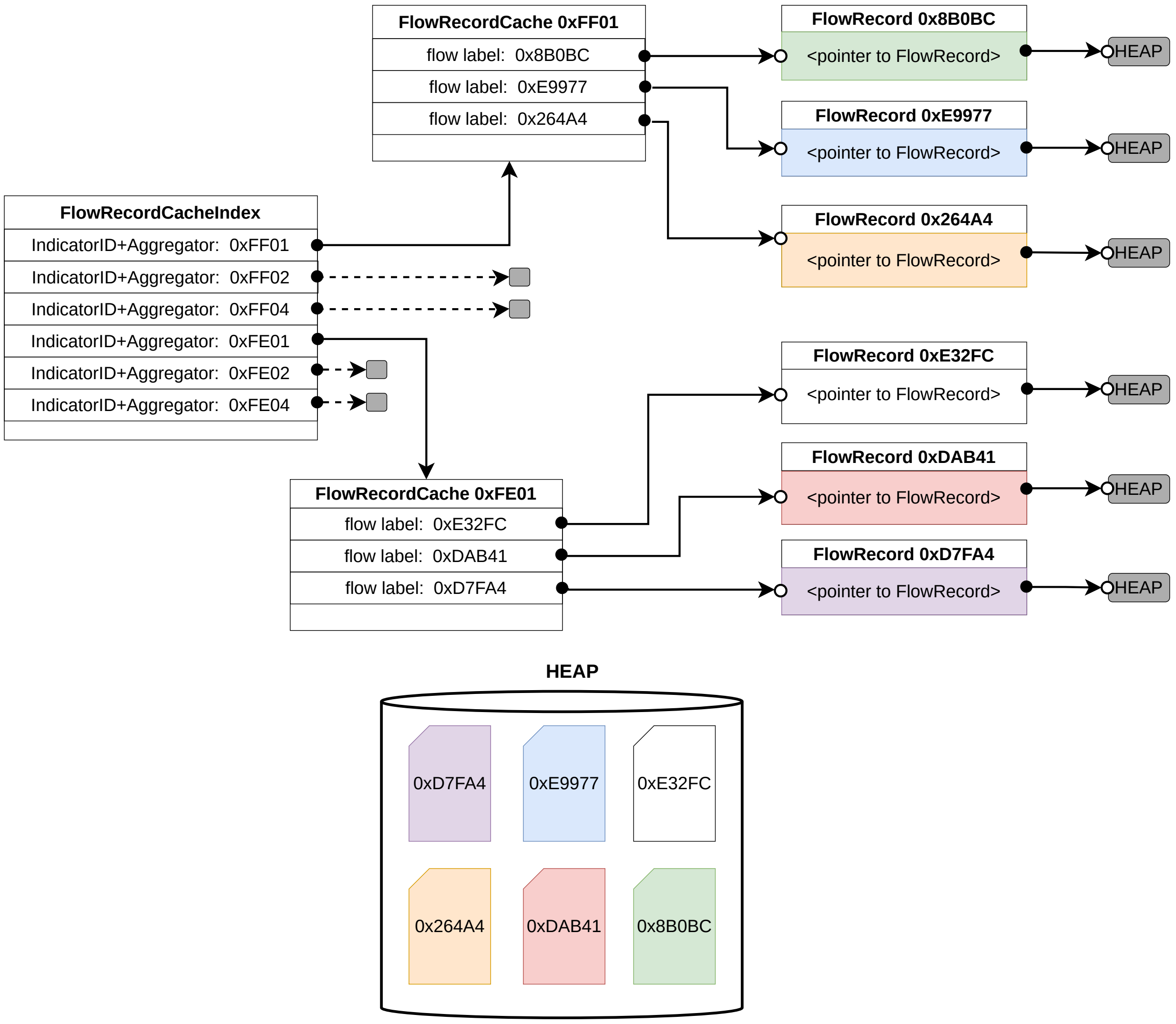


Figure 1.1.: Aggregated Export Cache Overview

1.4.2.2. Raw Data Export

The raw data export strategy is based on the direct export of flow data in combination with a user defined sampling rate per flow. It solves the following problem:

- Lack of information about a path a packet traversed. The raw export contains all information collected on behalf of the IOAM protocol which not only includes energy efficiency indicator data, but also a node list identifying the traversed path.

As mentioned in the introduction to the section about caching, the direct export does not require a local cache because the flow data is exported instantly as the data is related to a single packet. In order to be able to implement a flow based sampling to ensure there is at least one export of each flow, a mechanism is required to keep track of when the last export of a specific flow took place. As mentioned in the section about the aggregated export the FlowRecord data structure contains an additional field only used in the raw export strategy. The field stores the value of the packet delta count of the last export. That way it can be determined algorithmically whether the raw data of a given packet needs to be exported.

In order to have the possibility to batch process the export of multiple RawRecords a linked list like data structure is used to temporarily store the RawRecords pending for export. With this approach the export of both multiple RawRecords and a single RawRecord per IPFIX message is possible.

In the current exporter implementation only one RawRecord is exported at a time, because otherwise multiple exported RawRecords of probably the same flow are stored with the same timestamp inside the time series database.

1.4.3. Export Mechanism

This section contains a brief description on how the export mechanism and the related cache management is implemented for both the aggregated and raw data export of flow records. In general the export of flow records includes the following steps:

1. Determine the records to export (e.g. expiry due to flow inactivity).
2. Retrieve the list of flow records to export from the cache manager.
3. Determine the total message size based on the number of flow records to export.
4. Generate the raw payload containing the IPFIX message representation in binary format in network byte order.
5. Try to send a UDP datagram with the crafted raw payload.
6. In case the message could not be transmitted because its size exceeded the maximum transmission unit, split the message and retry the send operation. This process is repeated recursively until the send operation succeeded.

Figure 1.2 is a simplified view of the network virtualization system and only contains the IPFIX Extension system in full detail. For a more detailed insight into the structure of the network virtualization system and to see the relations to all other involved systems, refer to figure 3.5 in the elaboration part.

To get a better understanding about how the network virtualization system is implemented the following enumeration contains a step by step description on how it works referring to figure 1.2.

1. The data plane is responsible to forward network packets and update header fields. On an egress node the data plane has the additional responsibility to pass the aggregated network telemetry data to the IPFIX extension in the control plane. For that purpose the efficiency indicator metadata processor is called which is exposed to the data plane as an extern function.
2. Assuming the efficiency indicator metadata processor is called the first time it starts the aggregated export cache manager and the IPFIX template exporter as detached threads which continue to run in the background.
3. After that or if the background threads where already started the efficiency indicator metadata processor will handle the aggregated export use case by initializing a new flow record data structure with the obtained data from the data plane. The generated flow record will be inserted into the aggregated export cache. In case there is already a cache entry for the specific data param, aggregator and flow label the existing cache entry will be updated considering the selected aggregator on efficiency indicator value aggregation.
4. Once the aggregated export use case is handled the efficiency indicator metadata processor handles the raw export use case. The raw export is done based on a per flow sampling. More information about the raw export sampling is available in section 1.4.3.2. If an export is required the raw data provided by the data plane is written as a record into the raw export cache and the content of this cache is passed right away to the IPFIX data set exporter.

5. Once the raw record is expired the efficiency indicator metadata processor cleans up the raw export cache by deleting exported records.
6. The aggregated export cache manager periodically executes and algorithm to discover expired records inside the aggregated export cache.
7. The aggregated export cache manager hands expired records over to the IPFIX data set exporter.
8. Once the records where exported successfully, the aggregated export cache manager deletes exported records from the aggregated export cache.
9. The IPFIX data set exporter crafts raw IPFIX message payloads from the given records. Using those payloads including message headers etc. the IPFIX data set exporter crafts UDP datagrams which contains an IPFIX data set. This UDP datagram is then encapsulated in an IP packet and sent to the monitoring system.
10. The IPFIX template set exporter initially generates the UDP datagrams containing the IPFIX template sets based on the provided configuration. It then sends these template sets in a regular and configurable interval to the monitoring system.

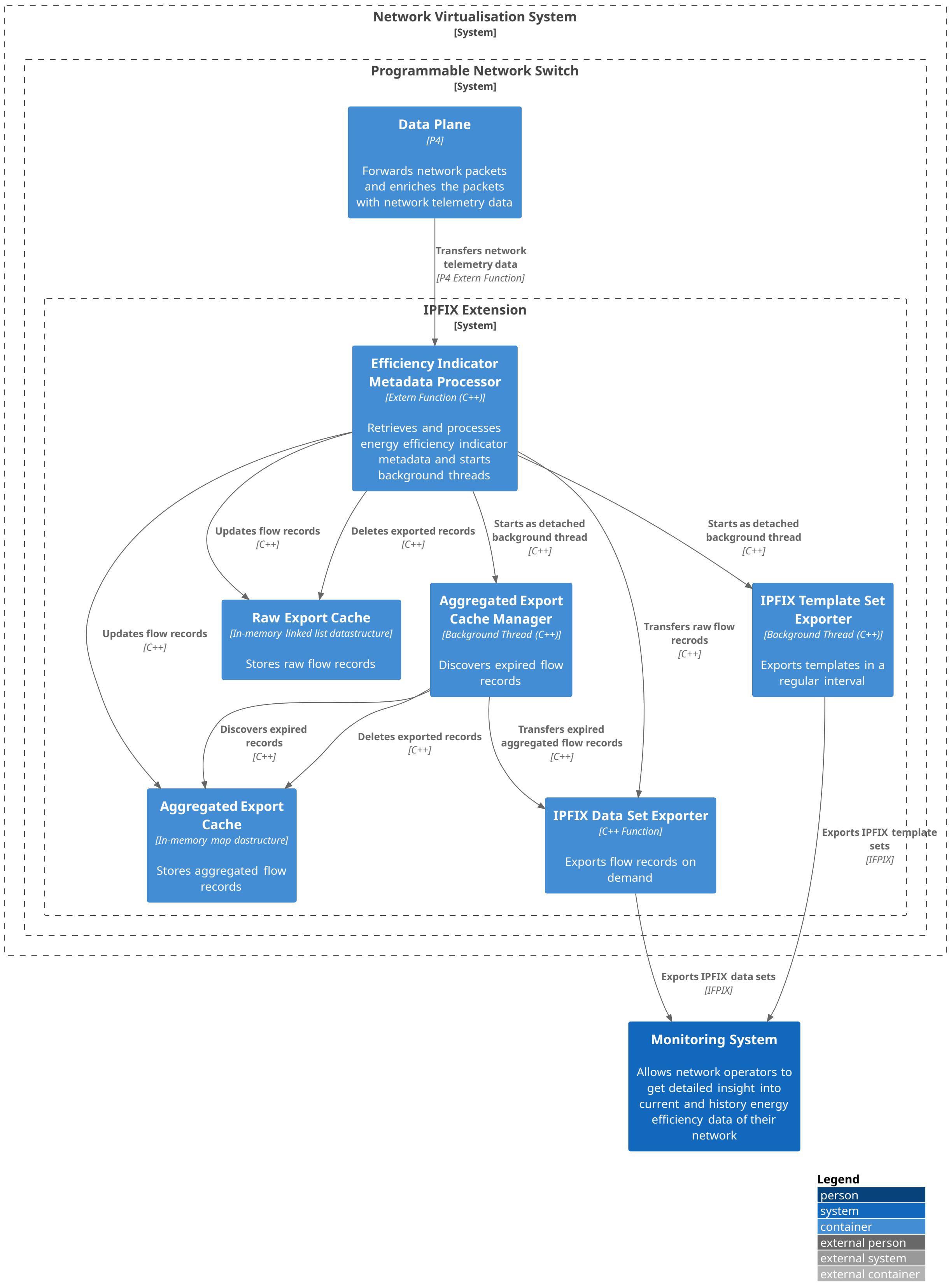

Network Virtualisation System
[System]
Programmable Network Switch
[System]
Data Plane
[P4]
Forwards network packets and enriches the packets with network telemetry data
Transfers network telemetry data
[P4 Extern Function]
IPFIX Extension
[System]
Efficiency Indicator Metadata Processor
[Extern Function (C++)]
Retrieves and processes energy efficiency indicator metadata and starts background threads
Updates flow records
[C++]
Deletes exported records
[C++]
Starts as detached background thread
[C++]
Starts as detached background thread
[C++]
Raw Export Cache
[In-memory linked list datastructure]
Stores raw flow records
Aggregated Export Cache Manager
[Background Thread (C++)]
Discovers expired flow records
IPFIX Template Set Exporter
[Background Thread (C++)]
Exports templates in a regular interval
Updates flow records
[C++]
Transfers raw flow records
[C++]
Discovers expired records
[C++]
Deletes exported records
[C++]
Transfers expired aggregated flow records
[C++]
Aggregated Export Cache
[In-memory map dastructure]
Stores aggregated flow records
IPFIX Data Set Exporter
[C++ Function]
Exports flow records on demand
Exports IPFIX template sets
[IFPIX]
Exports IPFIX data sets
[IFPIX]
Monitoring System
Allows network operators to get detailed insight into current and history energy efficiency data of their network
Legend
person
system
container
external person
external system
external container

Figure 1.2.: IPFIX Extension Component Diagram

1.4.3.1. Aggregated Data Export

The centerpiece of the aggregated data export mechanism is the aggregated export cache manager. As already mentioned the aggregated export cache manager is executed as a detached background thread. The thread executes the function depicted in listing 1.20.

Line 3 Ensures that the function runs forever (until the BMv2 software switch is turned off).

Line 4 Causes the execution of the cache manager to stop for the specified amount of time. The constant definition to adjust the discovery interval can be found in the *ipfix.h* file.

Line 5 Acquires a lock on the cache index mutex.

Line 6 Declares a new set which will hold 32 bit unsigned integer values. The set will be used to store the keys of the empty flow record caches.

Line 8 Iterates over all caches present in the cache index.

Line 9 Declares a temporary flow record cache to store the expired records.

Line 10 Calls the function *DiscoverExpiredFlowRecords* and provides the current cache and the temporary flow record cache as arguments. The function will iterate over all records present in the cache and stores the expired records in the temporary cache previously allocated on line 9.

Line 11 Calls the function *ExportFlowRecords* to export the expired records found during the discovery now stored in the temporary cache.

Line 12 Calls the function *DeleteFlowRecords* to delete the expired and exported flow records from the cache. The flow records are deallocated which frees memory.

Line 13 Checks if the current cache is now empty.

Line 14 If the cache is empty the key of the cache is added to the empty cache keys set declared on line 6.

Line 17 Calls the function *RemoveEmptyCaches* and provides the empty cache keys set as argument. The empty caches are deallocated which frees memory.

Listing 1.20: Aggregated Export Cache Manager

```
 1 void ManageFlowRecordCache() {
 2   BMLOG_DEBUG("IPFIX EXPORT: Starting flow record cache manager");
 3   while (true) {
 4     sleep(IPFIX_CACHE_MANAGER_DISCOVERY_INTERVAL);
 5     std::lock_guard<std::mutex> guard(cache_index_mutex);
 6     std::set<uint32_t> empty_cache_keys;
 7     // Iterate over all keys and corresponding values
 8     for (auto i = cache_index.begin(); i != cache_index.end(); i++) {
 9       FlowRecordCache expired_records;
10       DiscoverExpiredFlowRecords(i->second, expired_records);
11       ExportFlowRecords(expired_records);
12       DeleteFlowRecords(i->second, expired_records);
13       if (i->second->empty()) {
14         empty_cache_keys.insert(i->first);
15       }
16     }
```

```
    RemoveEmptyCaches(empty_cache_keys);
  }
}
```

The explanation of the code snippet in listing 1.20 is taken from the file *cache.cpp*.

i Information

All functions present in listing 1.20 except for the function *ExportFlowRecords* are implemented in the file *cache.cpp*. The *ExportFlowRecords* function is implemented in the file *export.cpp*. Subsequent calls are made to the functions to generate the payload and to send the packet. Those functions are implemented in *export_utils.cpp* for your reference.

1.4.3.2. Raw Data Export

This section contains details specifically relevant for the export of raw data. As already mentioned in the general description of the export mechanism raw data is exported based on a per flow sampling rate. In the file ipfix.h the constant `IPFIX_RAW_EXPORT_SAMPLE_RATE` can be used to specify the sample rate. By default the value of the constant is set to 50 which means that every 50th packet of a flow will be exported.

The code snippet shown in listing 1.21 is used to decide whether a raw export is required or not and is taken from the file *cache.cpp*.

Line 2 Acquires a lock on the cache index mutex.

Line 3-6 Determine if the flow record exists.

Line 7-8 Determine the number of packets sent in the current flow since the last raw export. The condition is met if the number of packets since the last export is greater or equal the specified sample rate or if there was no raw export before.

Line 9-10 Check if the condition is fulfilled to attempt a raw export.

Line 11-12 Set the last raw export field of the current flow record to the value of the packet count field.

Listing 1.21: Raw Export Required Snippet

```
1  bool IsRawExportRequired(FlowRecordCache *cache, const bm::Data &flow_key) {
2    std::lock_guard<std::mutex> guard(cache_index_mutex);
3    auto i = cache->find(flow_key);
4    if (i == cache->end()) {
5      return false;
6    }
7    uint64_t last_export_delta = cache->at(flow_key).packet_delta_count -
8                                 cache->at(flow_key).last_raw_export;
9    if (last_export_delta >= IPFIX_RAW_EXPORT_SAMPLE_RATE ||
10       cache->at(flow_key).last_raw_export == 0) {
11     cache->at(flow_key).last_raw_export =
12         cache->at(flow_key).packet_delta_count;
13     return true;
14   }
15   return false;
16 }
```

If the function depicted in listing 1.21 returns true, the following lines are executed resulting in a transfer of the raw record to the exporter followed by the deletion of the raw record from the cache.

Listing 1.22: Raw Export Control Snippet

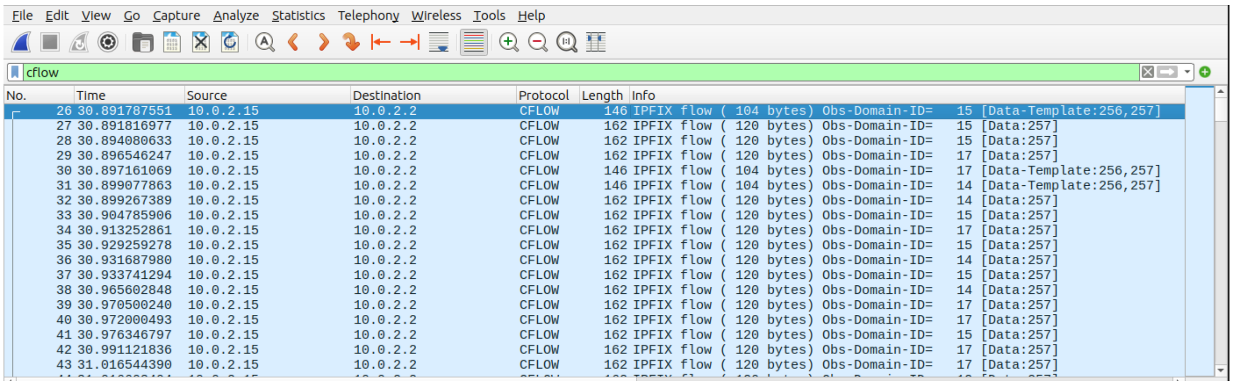
```
if (IsRawExportRequired(cache, flow_key)) {
  std::lock_guard<std::mutex> guard(raw_record_cache_mutex);
  RawRecord *record = GetRawRecord(raw_ipv6_header);
  InsertRawRecord(record);
  ExportRawRecords(raw_record_cache);
  DeleteRawRecords();
}
```

i Information

The *ExportRawRecords* function is implemented in the file *export.cpp*. Subsequent calls are made to the functions to generate the payload and to send the packet. Those functions are implemented in *export_utils.cpp* for your reference.

1.4.4. IPFIX Messages in Wireshark

IPFIX messages are sent via the network from the BMv2 targets to the monitoring system. On their way they can be captured using Wireshark which is an open source tool to analyze network traffic. The display filter to filter for IPFIX messages is called `cflow`. The reason for this strange name is that the IPFIX protocol evolved from the proprietary protocol called Cisco NetFlow through an IETF standardization process. Figure 1.3 contains an IPFIX packet list. One can see that the observation domain ID, which corresponds to the node ID of the exporting node, varies, which means that the exported packets originate from different BMv2 software switches.

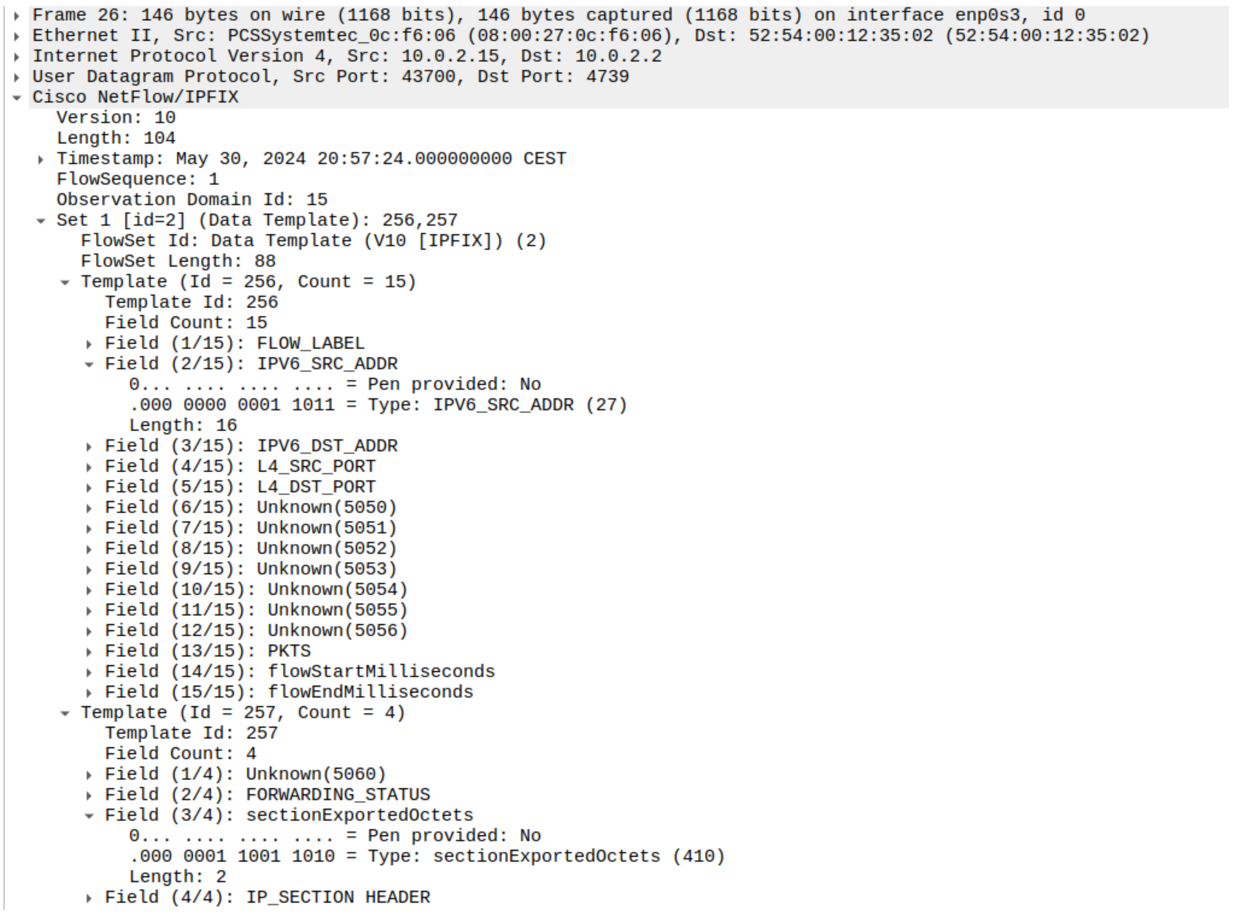

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
cflow

```
No.   Time         Source      Destination  Protocol  Length  Info
   26 30.891787551 10.0.2.15   10.0.2.2     CFLOW     146 IPFIX flow ( 104 bytes) Obs-Domain-ID=   15 [Data-Template:256,257]
   27 30.891816977 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   15 [Data:257]
   28 30.894080633 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   15 [Data:257]
   29 30.896546247 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   17 [Data:257]
   30 30.897161069 10.0.2.15   10.0.2.2     CFLOW     146 IPFIX flow ( 104 bytes) Obs-Domain-ID=   17 [Data-Template:256,257]
   31 30.899077863 10.0.2.15   10.0.2.2     CFLOW     146 IPFIX flow ( 104 bytes) Obs-Domain-ID=   14 [Data-Template:256,257]
   32 30.899267389 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   14 [Data:257]
   33 30.904785906 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   15 [Data:257]
   34 30.913252861 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   17 [Data:257]
   35 30.929259278 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   15 [Data:257]
   36 30.931687980 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   14 [Data:257]
   37 30.933741294 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   15 [Data:257]
   38 30.965602848 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   14 [Data:257]
   39 30.970500240 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   17 [Data:257]
   40 30.972000493 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   17 [Data:257]
   41 30.976346797 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   15 [Data:257]
   42 30.991121836 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   17 [Data:257]
   43 31.016544390 10.0.2.15   10.0.2.2     CFLOW     162 IPFIX flow ( 120 bytes) Obs-Domain-ID=   17 [Data:257]
```

Figure 1.3.: List of IPFIX Packets

1.4.4.1. Template Set Message

In IPFIX template sets are periodically exported to configure the collecting nodes. As specified in RFC 7011 a template set is a collection of one or more template records that have been grouped together in an IPFIX Message. In the template set shown in figure 1.4 two template records are exported. The template record with the ID 256 is the template for the aggregated export and the template record with ID 257 is for the raw export. The format and content of both templates was introduced in section 3.5.1 in the elaboration part.

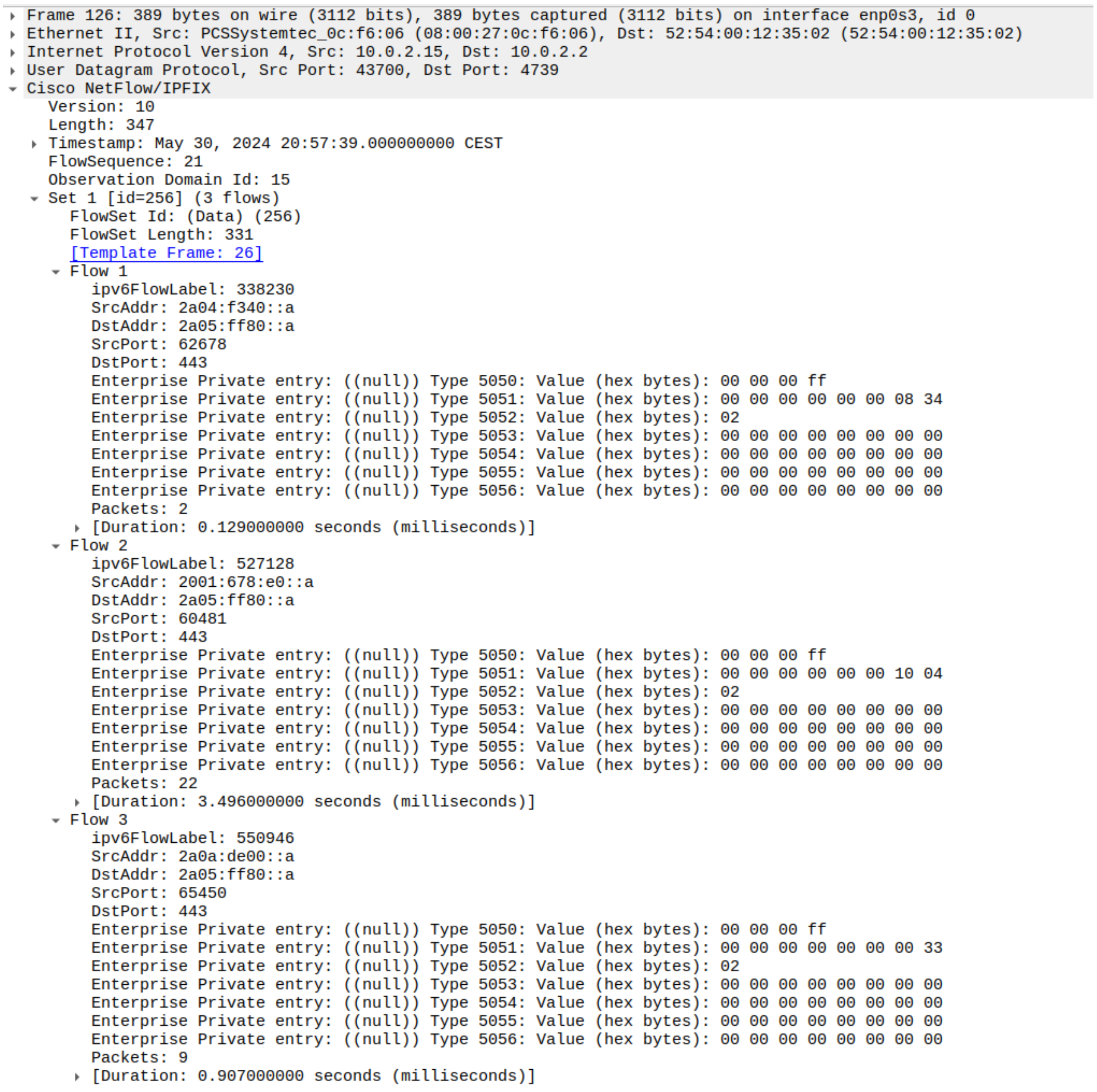


```
▸ Frame 26: 146 bytes on wire (1168 bits), 146 bytes captured (1168 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PCSSystemtec_0c:f6:06 (08:00:27:0c:f6:06), Dst: 52:54:00:12:35:02 (52:54:00:12:35:02)
▸ Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.2
▸ User Datagram Protocol, Src Port: 43700, Dst Port: 4739
▾ Cisco NetFlow/IPFIX
    Version: 10
    Length: 104
  ▸ Timestamp: May 30, 2024 20:57:24.000000000 CEST
    FlowSequence: 1
    Observation Domain Id: 15
  ▾ Set 1 [id=2] (Data Template): 256,257
      FlowSet Id: Data Template (V10 [IPFIX]) (2)
      FlowSet Length: 88
    ▾ Template (Id = 256, Count = 15)
        Template Id: 256
        Field Count: 15
      ▸ Field (1/15): FLOW_LABEL
      ▾ Field (2/15): IPV6_SRC_ADDR
          0... .... .... .... = Pen provided: No
          .000 0000 0001 1011 = Type: IPV6_SRC_ADDR (27)
          Length: 16
      ▸ Field (3/15): IPV6_DST_ADDR
      ▸ Field (4/15): L4_SRC_PORT
      ▸ Field (5/15): L4_DST_PORT
      ▸ Field (6/15): Unknown(5050)
      ▸ Field (7/15): Unknown(5051)
      ▸ Field (8/15): Unknown(5052)
      ▸ Field (9/15): Unknown(5053)
      ▸ Field (10/15): Unknown(5054)
      ▸ Field (11/15): Unknown(5055)
      ▸ Field (12/15): Unknown(5056)
      ▸ Field (13/15): PKTS
      ▸ Field (14/15): flowStartMilliseconds
      ▸ Field (15/15): flowEndMilliseconds
    ▾ Template (Id = 257, Count = 4)
        Template Id: 257
        Field Count: 4
      ▸ Field (1/4): Unknown(5060)
      ▸ Field (2/4): FORWARDING_STATUS
      ▾ Field (3/4): sectionExportedOctets
          0... .... .... .... = Pen provided: No
          .000 0001 1001 1010 = Type: sectionExportedOctets (410)
          Length: 2
      ▸ Field (4/4): IP_SECTION HEADER
```

Figure 1.4.: IPFIX Template Set Message

1.4.4.2. Data Set Message

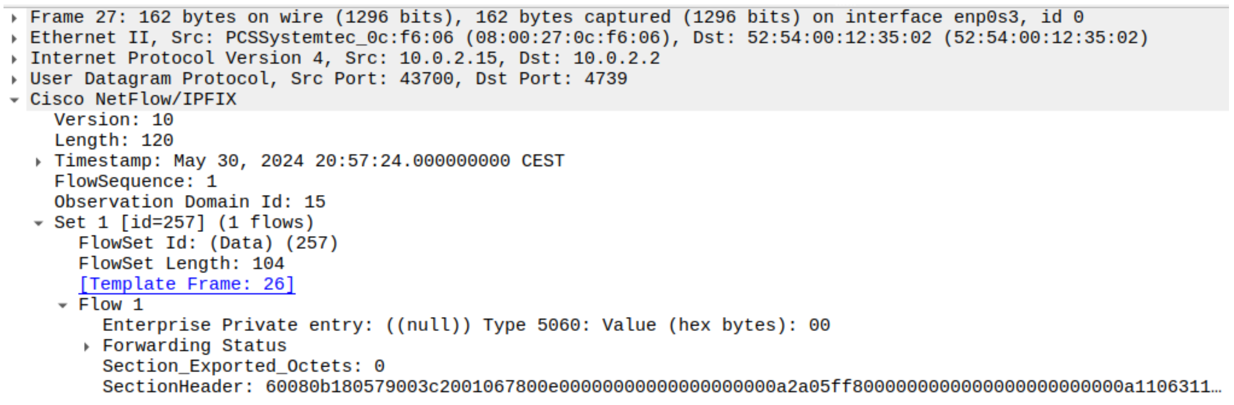
Figure 1.5 is an example for an aggregated data export based on the template with the ID 256. The IPFIX data set contains three data records which illustrates that the export of multiple records within one IPFIX message are supported by our implementation. For more details about the specific fields refer to section 3.5.1 in the elaboration part.



```
▸ Frame 126: 389 bytes on wire (3112 bits), 389 bytes captured (3112 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PCSSystemtec_0c:f6:06 (08:00:27:0c:f6:06), Dst: 52:54:00:12:35:02 (52:54:00:12:35:02)
▸ Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.2
▸ User Datagram Protocol, Src Port: 43700, Dst Port: 4739
▾ Cisco NetFlow/IPFIX
    Version: 10
    Length: 347
  ▸ Timestamp: May 30, 2024 20:57:39.000000000 CEST
    FlowSequence: 21
    Observation Domain Id: 15
  ▾ Set 1 [id=256] (3 flows)
      FlowSet Id: (Data) (256)
      FlowSet Length: 331
      [Template Frame: 26]
    ▾ Flow 1
        ipv6FlowLabel: 338230
        SrcAddr: 2a04:f340::a
        DstAddr: 2a05:ff80::a
        SrcPort: 62678
        DstPort: 443
        Enterprise Private entry: ((null)) Type 5050: Value (hex bytes): 00 00 00 ff
        Enterprise Private entry: ((null)) Type 5051: Value (hex bytes): 00 00 00 00 00 00 08 34
        Enterprise Private entry: ((null)) Type 5052: Value (hex bytes): 02
        Enterprise Private entry: ((null)) Type 5053: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5054: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5055: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5056: Value (hex bytes): 00 00 00 00 00 00 00 00
        Packets: 2
      ▸ [Duration: 0.129000000 seconds (milliseconds)]
    ▾ Flow 2
        ipv6FlowLabel: 527128
        SrcAddr: 2001:678:e0::a
        DstAddr: 2a05:ff80::a
        SrcPort: 60481
        DstPort: 443
        Enterprise Private entry: ((null)) Type 5050: Value (hex bytes): 00 00 00 ff
        Enterprise Private entry: ((null)) Type 5051: Value (hex bytes): 00 00 00 00 00 00 10 04
        Enterprise Private entry: ((null)) Type 5052: Value (hex bytes): 02
        Enterprise Private entry: ((null)) Type 5053: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5054: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5055: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5056: Value (hex bytes): 00 00 00 00 00 00 00 00
        Packets: 22
      ▸ [Duration: 3.496000000 seconds (milliseconds)]
    ▾ Flow 3
        ipv6FlowLabel: 550946
        SrcAddr: 2a0a:de00::a
        DstAddr: 2a05:ff80::a
        SrcPort: 65450
        DstPort: 443
        Enterprise Private entry: ((null)) Type 5050: Value (hex bytes): 00 00 00 ff
        Enterprise Private entry: ((null)) Type 5051: Value (hex bytes): 00 00 00 00 00 00 00 33
        Enterprise Private entry: ((null)) Type 5052: Value (hex bytes): 02
        Enterprise Private entry: ((null)) Type 5053: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5054: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5055: Value (hex bytes): 00 00 00 00 00 00 00 00
        Enterprise Private entry: ((null)) Type 5056: Value (hex bytes): 00 00 00 00 00 00 00 00
        Packets: 9
      ▸ [Duration: 0.907000000 seconds (milliseconds)]
```

Figure 1.5.: IPFIX Data Set with ID 256

Figure 1.6 is an example for a raw data export based on the template with the ID 257. It contains the fields as specified in draft-spiegel-ippm-ioam-rawexport-07. [14] The most important field is the *SectionHeader* field which contains the raw binary data of the complete IPv6 header including all extension headers. For more details about the specific fields refer to the draft and section 3.5.1 in the elaboration part.



```
▸ Frame 27: 162 bytes on wire (1296 bits), 162 bytes captured (1296 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PCSSystemtec_0c:f6:06 (08:00:27:0c:f6:06), Dst: 52:54:00:12:35:02 (52:54:00:12:35:02)
▸ Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.2
▸ User Datagram Protocol, Src Port: 43700, Dst Port: 4739
▾ Cisco NetFlow/IPFIX
    Version: 10
    Length: 120
  ▸ Timestamp: May 30, 2024 20:57:24.000000000 CEST
    FlowSequence: 1
    Observation Domain Id: 15
  ▾ Set 1 [id=257] (1 flows)
      FlowSet Id: (Data) (257)
      FlowSet Length: 104
      [Template Frame: 26]
    ▾ Flow 1
        Enterprise Private entry: ((null)) Type 5060: Value (hex bytes): 00
      ▸ Forwarding Status
        Section_Exported_Octets: 0
        SectionHeader: 60080b180579003c2001067800e0000000000000000000a2a05ff8000000000000000000000000a1106311…
```

Figure 1.6.: IPFIX Data Set with ID 257

2. Monitoring System

This chapter describes the most important information about the construction of the monitoring system. For detailed information about the implementation or configuration, please visit the individual repositories. The C4 container diagram of the monitoring system is shown in chapter 4 in the elaboration part and the design decision of the monitoring system are also recorded in the previously mentioned chapter in the elaboration part.

2.1. Getting Started

The monitoring system is deployed by using Docker Compose, so the prerequisite is that Docker must be installed on the host where it will be launched. To use the monitoring system clone the repository and change the working directory into the desired deployment folder, local or server.

local The local directory is used to deploy the monitoring system on the same computer as the network virtualization (Mininet) is running.

server The server directory is used to deploy the monitoring system on a dedicated server.

In the particular directory there is a Docker Compose file, you can run it with the following command:

Listing 2.1: Getting Started - Monitoring System

```
docker compose up
```

After the *docker compose up* command was entered the following three services are up and running:

Telegraf acts as IFPIX collector and listens on *udp://<IP-ADDRESS>:4739*

InfluxDB acts as time series database and is available on *http://<IP-ADDRESS>:8086*

Grafana acts as the monitoring dashboard and is available on *http://<IP-ADDRESS>:3000*

2.2. IPFIX Collector - Telegraf

As described in the elaboration chapter 4, Telegraf is used as IPFIX collector. The main advantage of Telegraf is that it is plugin based. That doesn't make it any easier, on the contrary, the configuration was quite complex because we had to evaluate our needs precisely and then look for the most suitable plugins to achieve the goal. But in the end a specific *telegraf.conf* file exists that suits for our specific use case.

The following four types of plugins exists in Telegraf: [15]

Input Plugins The input plugins are used to collect metrics from system, services or third-party APIs.

Processor Plugins The processor plugins transform, decorate and filter metrics.

Aggregator Plugins The aggregator plugins create aggregated metrics (mean, max, min, etc.)

Output Plugins The output plugins send metrics to various destinations.

Information

Further information about all the available plugins for Telegraf can be found in the official documentation.

The following subsections describe the plugins that are used in the Telegraf configuration to provide all the functionality required for Telegraf to act as an IPFIX collector. All code snippets shown in the following subsections are from the Telegraf's config file *telegraf.conf*.

2.2.1. Input Plugins

The input plugins are mainly used to collect metrics from systems and services. We used the following Telegraf input plugins for the IPFIX collector:

inputs.netflow The Netflow input plugin gathers metrics from Netflow v5, Netflow v9 and IPFIX collectors.

The configuration file snippet 2.2 shows that with the plugin *inputs.netflow* Telegraf is configured to listen for Netflow, IPFIX or sFlow packets and explicitly set to ipfix with the option *protocol = "ipfix"*. The service listening port is set to *udp://4739*. The option *inputs.netflow.tags* adds to each received metric the additional tag *export_type* with the value *bucket* which is in the beginning just a placeholder. This placeholder is later used to distinct if the collected metric is of type *aggregated_data_export* or *raw_data_export*.

Listing 2.2: Input plugins

```
# Netflow v5, Netflow v9 and IPFIX collector
[[inputs.netflow]]
    ## Address to listen for netflow,ipfix or sflow packets.
    service_address = "udp://:4739"

    ## Set the size of the operating system's receive buffer.
    ##   example: read_buffer_size = "64KiB"
    ## Uses the system's default if not set.
    read_buffer_size = "64KiB"

    ## Protocol version to use for decoding.
    ##   "ipfix"      -- IPFIX / Netflow v10 protocol (also works for Netflow v9)
    protocol = "ipfix"

[inputs.netflow.tags]
    export_type = "bucket"
```

2.2.2. Processor Plugins

The processor plugins are used to transform and filter the data within the IPFIX metrics. We used the following Telegraf processor plugins for the IPFIX collector:

processor.strings The strings converter plugin maps certain Go string functions onto tag and field values.

processor.converter The converter plugin is used to transform field types, change the type of tag or field and it can convert between fields and tags.

processor.regex The regex plugin transform tag and fields using a regular expression (regex) pattern.

processor.starlark Starlark (formerly known as Skylark) is a dialect of Python with limited functionality which is intended for use as configuration language.

The code snippet 2.3, shows how we used the *processor.strings.trim_left* plugin. The field *ip_header_packet_section* is selected and the two characters *0x* at the beginning of the string are removed.

Listing 2.3: Processor Plugin - Strings

```
[[processors.strings]]
    [[processors.strings.trim_left]]
      field = "ip_header_packet_section"
      cutset = "0x"
```

The config file snippet 2.4 shows how the field type of *type_5053, type_5054, type_5055* and *type_5056* is changed (from hexadecimal) to unsigned integer. In the next steps all the fields *flow_label, type_5053, type_5054, type_5055, type_5056, src, dst, src_port, dst_port* and *type_5060* are changed from type field to type tag. This type conversion is necessary because we need those fields as tags for the Flux query. Last but not least the field type of *type_5051* is changed (from hexadecimal) to integer. The specific fields from the IFPIX template are introduced and further described in section 3.5.1 in the elaboration part.

Listing 2.4: Processor Plugin - Converter

```
[[processors.converter]]
    [processors.converter.fields]
        unsigned = ["type_5053", "type_5054", "type_5055", "type_5056"]

# Conversion of specified IPFIX packet fields into tags
[[processors.converter]]
    [processors.converter.fields]
        tag = ["flow_label", "type_5050", "type_5052", "type_5053", "type_5054",
    "type_5055", "type_5056", "src", "dst", "src_port", "dst_port", "type_5060"]
        integer = ["type_5051"]
```

The regex processor plugin is used to transform tag and field values using regular expression (regex), further described in the GitHub README. We use this plugin to distinguish if the received metric is from the aggregated data export or the raw data export. In the code snippet 2.5 its shown how we check if its a aggregated or raw data export. We check whether the *type_5050* or the *type_5060* exists in the metric. The tag *type_5050* exists only in the aggregated data export so if this tag exists we replace the value of the previously created placeholder tag named *export_type* to aggregated_data_export. The same is done with the tag *type_5060* which only exists in the raw data export, so there the value of the tag *export_type* is set to raw_data_export. The output plugin that writes the metrics to the InfluxDB buckets will use the tag *export_type* to distinguish if the metric belongs to the bucket *aggregated_data_export* or to the bucket *raw_data_export*.

Listing 2.5: Processor Plugin - Regex

```
# Determine the export type of the metric and set the corresponding tag
[[processors.regex]]
```

```
    namepass = ["netflow"]

    [[processors.regex.tags]]
    key = "type_5050"
    pattern = '.*'
    result_key = "export_type"
    replacement = "aggregated_data_export"
    append = false

    [[processors.regex.tags]]
    key = "type_5060"
    pattern = '.*'
    result_key = "export_type"
    replacement = "raw_data_export"
    append = false
```

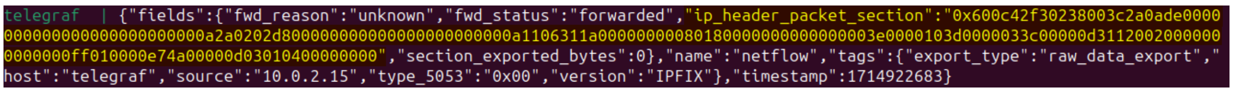
The Starlark processor plugin calls a Starlark function for each matched metric, the plugin is further described in the official GitHub README. This allows us to define a custom function to parse the data from the raw data export. Starlark is a dialect of Python but there are important differences to note: [16]

- Starlark has limited support for error handling. If an error occurs the script will immediately end and the metric will be dropped.
- Import other packages is not possible and the Python standard library is not available.
- Starlark has no access to the filesystem or sockets.

The code snippet in listing 2.6 shows the Starlark processing plugin with the custom function that is used to parse the data field *ip_header_packet_section* from the IPFIX raw data export. The IPFIX raw data export is further described in section 1.4.

The function **apply(metric)** gets the metric as input and checks first if the metric is not type of raw data export the function is aborted. The content of the field *ip_header_packet_section* is mapped to the variable *data* and the input metric is dropped. The relevant information from the raw data is then parsed into new tags and fields. After the parsing process is complete, the new metric is returned.

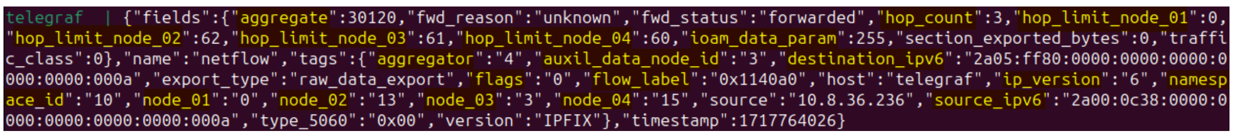
Figure 2.1 shows a received IPFIX raw data export from Telegraf before the raw data parser was implemented. As can be seen in the figure, the field *ip_header_packet_section* is a raw binary blob which is unreadable without the knowledge of the underlying structure.



```
telegraf  | {"fields":{"fwd_reason":"unknown","fwd_status":"forwarded","ip_header_packet_section":"0x600c42f30238003c2a0ade0000
000000000000000000a2a0202d8000000000000000000000000a1106311a000000000801800000000000000003e0000103d0000033c00000d3112002000000
0000000ff010000e74a00000d03010400000000","section_exported_bytes":0},"name":"netflow","tags":{"export_type":"raw_data_export","
host":"telegraf","source":"10.0.2.15","type_5053":"0x00","version":"IPFIX"},"timestamp":1714922683}
```

Figure 2.1.: Telegraf - Raw Data Export Before Parsing

Figure 2.2 below shows a received IFPIX raw data export from Telegraf where the raw data parser is implemented. The yellow marked values from the field *ip_header_packet_section* in the picture above are now parsed into multiple tags and fields.



```
telegraf  | {"fields":{"aggregate":30120,"fwd_reason":"unknown","fwd_status":"forwarded","hop_count":3,"hop_limit_node_01":0,
"hop_limit_node_02":62,"hop_limit_node_03":61,"hop_limit_node_04":60,"ioam_data_param":255,"section_exported_bytes":0,"traffi
c_class":0},"name":"netflow","tags":{"aggregator":"4","auxil_data_node_id":"3","destination_ipv6":"2a05:ff80:0000:0000:0000:0
000:0000:000a","export_type":"raw_data_export","flags":"0","flow_label":"0x1140a0","host":"telegraf","ip_version":"6","namesp
ace_id":"10","node_01":"0","node_02":"13","node_03":"3","node_04":"15","source":"10.8.36.236","source_ipv6":"2a00:0c38:0000:0
000:0000:0000:0000:000a","type_5060":"0x00","version":"IPFIX"},"timestamp":1717764026}
```

Figure 2.2.: Telegraf - Raw Data Export After Parsing

The function **reformat__ipv6__address(string)** takes a hex string as input and reformats the string to match the ipv6 address format.

Input 2001067800e00000000000000000000a

Output 2001:0678:00e0:0000:0000:0000:0000:000a

Listing 2.6: Processor Plugin - Starlark

```
# Parsing raw_data_export field ip_header_packet_section
[[processors.starlark]]
    source = '''
def apply(metric):
    if metric.tags["export_type"] == "raw_data_export":
        data = metric.fields.pop("ip_header_packet_section")

        metric.tags["ip_version"] = str(int(data[0:1], 16))
        metric.fields["traffic_class"] = int(data[1:3], 16)
        metric.tags["flow_label"] = "0x" + str(data[3:9])
        metric.tags["source_ipv6"] = reformat_ipv6_address(str(data[16:48]))
        metric.tags["destination_ipv6"] = reformat_ipv6_address(str(data[48:80]))
        metric.tags["node_01"] = str(int(data[110:116], 16))
        metric.tags["node_02"] = str(int(data[118:124], 16))
        metric.tags["node_03"] = str(int(data[126:132], 16))
        metric.tags["node_04"] = str(int(data[134:140], 16))
        metric.fields["hop_limit_node_01"] = int(data[108:110], 16)
        metric.fields["hop_limit_node_02"] = int(data[116:118], 16)
        metric.fields["hop_limit_node_03"] = int(data[124:126], 16)
        metric.fields["hop_limit_node_04"] = int(data[132:134], 16)
        metric.tags["namespace_id"] = str(int(data[148:152], 16))
        metric.tags["flags"] = str(int(data[152:153], 16))
        metric.fields["ioam_data_param"] = int(data[156:162], 16)
        metric.tags["aggregator"] = str(int(data[162:164], 16))
        metric.fields["aggregate"] = int(data[164:172], 16)
        metric.tags["auxil_data_node_id"] = str(int(data[172:178], 16))
        metric.fields["hop_count"] = int(data[178:180], 16)

    return metric

def reformat_ipv6_address(string):
    chunks = [string[i:i+4] for i in range(0, len(string), 4)]
    formatted_string = ':'.join(chunks)
    return formatted_string
'''
```

2.2.3. Output Plugins

The output plugins are used to send metrics to various destinations. We used the following Telegraf output plugins for the IPFIX collector:

outputs.influxdb__v2 The influxdb v2 output plugin writes metrics to the InfluxDB2.

outputs.file The file output plugin writes metrics to files.

The following code snippet shows the configuration of the output plugins. The *influxdb_v2* writes the metrics to the Influx database and distinguishes in which bucket the metrics belong (aggregated data export or raw data export) based on the previous set *export_type*. Additionally, the output is written to standard output in json format for debugging and development purposes.

For that it is necessary to specify the following parameters:

urls specifies the url (ip address and port) of the InfluxDB instance

organization specifies the organization which is used in the InfluxDB

token is the InfluxDB authorization token

bucket specifies the name of the bucket in which the metric is to be written

Listing 2.7: Output Plugins

```
# InfluxDB Bucket for the aggregated data export
[[outputs.influxdb_v2]]
urls = ["http://<INFLUX_HOST_IP>:8086"]
organization = "<ORGANIZATION>"
token = "<TOKEN>"
bucket = "aggregated_data_export"
[outputs.influxdb_v2.tagpass]
    export_type = ["aggregated_data_export"]

# InfluxDB Bucket for the raw data export
[[outputs.influxdb_v2]]
urls = ["http://<INFLUX_HOST_IP>:8086"]
organization = "<ORGANIZATION>"
token = "<TOKEN>"
bucket = "raw_data_export"
[outputs.influxdb_v2.tagpass]
    export_type = ["raw_data_export"]

[[outputs.file]]
    files = ["stdout"]
    rotation_interval = "1h"
    rotation_max_size = "20MB"
    rotation_max_archives = 3
    data_format = "json"
```

2.3. Time Series Database - InfluxDB

The InfluxDB does not need a lot of configuration. We use for the deployment as described in section 2.5.1.1 the latest Docker image of influxdb and configure the InfluxDB with predefined environment variables as described in section 2.5.2 during the initial startup process. In the following subsection it's described how the two additional buckets for the *aggregated_data_export* and the *raw_data_export* will be created.

2.3.1. Buckets

A script with the name *create_bucket.sh* is located in the folder *influx_scripts* to create the two necessary buckets. This script will be executed right after the initial startup of the influxdb service. For the additional bucket creation the influx cli is used to connect to the influx instance. A bucket with the name *aggregated_data_export* is created for the IPFIX Aggregated Export and another bucket with the name *raw_data_export* for the IPFIX Raw Export. For both of these buckets, the retention time is set to 10 days in order to avoid storing too much telemetry data and avoid running out of disk space. The *set -e* option aborts the execution of the script immediately if any command returns a non-zero exit status.

Listing 2.8: Create InfluxDB Buckets

```bash
#!/bin/bash
set -e

# Create a new bucket using influx CLI
influx bucket create --name aggregated_data_export --org OST --retention 10d
influx bucket create --name raw_data_export --org OST --retention 10d
```

2.3.2. Tags and Fields

In InfluxDB tags and fields are crucial components used to store and query data efficiently.

Tags Tags are key-value pairs that are indexed and intended for metadata, such as the flow label or source ip address. Tags allow efficient filtering and grouping of the time-series network efficiency data. Since the tags are indexed, queries involving tags are faster.

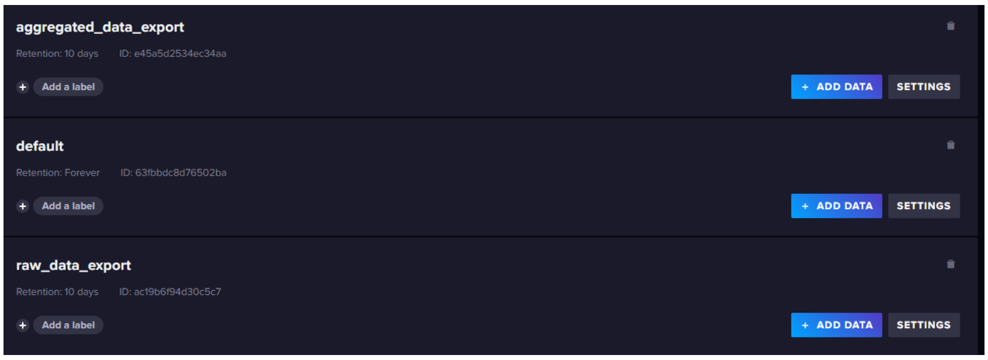
Fields Fields are also key-value pairs, but they are not indexed. They store the actual data values, such as the aggregate representing for example the flow efficiency indicator (FEI).

Information

In InfluxDB, tags must be of type string. Tags in InfluxDB are used for metadata and are indexed to provide efficient querying. Since tags are indexed and used for filtering and grouping queries, they are stored as strings to facilitate these operations.

2.3.2.1. InfluxDB Web-UI

The Web UI of the InfluxDB is available on http://<IP_ADDRESS>:8086. In the side navigation you can click on *Buckets* and the following buckets should be listed:



aggregated_data_export
Retention: 10 days   ID: e45a5d2534ec34aa

default
Retention: Forever   ID: 63fbbdc8d76502ba

raw_data_export
Retention: 10 days   ID: ac19b6f94d30c5c7

Figure 2.3.: InfluxDB Web UI - Buckets

2.3.2.2. InfluxDB Web-UI Query Builder

The query builder on the Web-UI of the InfluxDB is very helpful and easy to use for data analysis. We used the query builder quite often during the development process as data inspection tool.

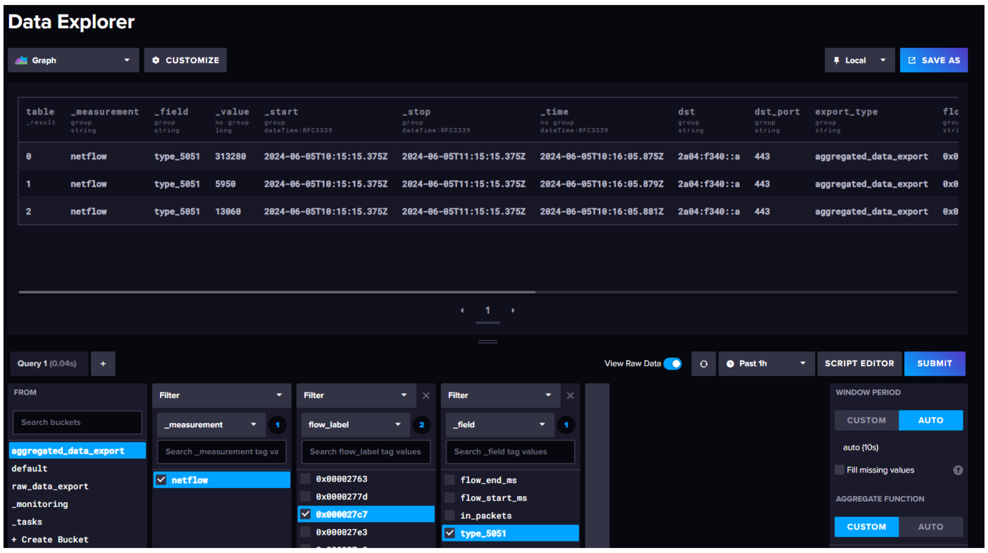


table _result	_measurement group string	_field group string	_value no group long	_start group dateTime:RFC3339	_stop group dateTime:RFC3339	_time no group dateTime:RFC3339	dst group string	dst_port group string	export_type group string	flo grou stri
0	netflow	type_5051	313280	2024-06-05T10:15:15.375Z	2024-06-05T11:15:15.375Z	2024-06-05T10:16:05.875Z	2a04:f340::a	443	aggregated_data_export	0x0
1	netflow	type_5051	5950	2024-06-05T10:15:15.375Z	2024-06-05T11:15:15.375Z	2024-06-05T10:16:05.879Z	2a04:f340::a	443	aggregated_data_export	0x0
2	netflow	type_5051	13060	2024-06-05T10:15:15.375Z	2024-06-05T11:15:15.375Z	2024-06-05T10:16:05.881Z	2a04:f340::a	443	aggregated_data_export	0x0

FROM	Filter	Filter	Filter
Search buckets	_measurement (1)	flow_label (2)	_field (1)
	Search _measurement tag va	Search flow_label tag values	Search _field tag values
aggregated_data_export	✓ netflow	0x00002763	flow_end_ms
default		0x0000277d	flow_start_ms
raw_data_export		✓ 0x000027c7	in_packets
_monitoring		0x000027e3	✓ type_5051
_tasks			
+ Create Bucket			

Figure 2.4.: InfluxDB Web UI - Query Builder

2.3.3. InfluxDB Web-UI Script Editor

The script editor is the pendant to the query builder, in the script editor the Flux query language is used. The script editor was also very helpful to write queries for the different Grafana dashboards because one gets instant feedback if the query works and the raw data is shown in the table above. So the development process of writing the queries for the different dashboards was very interactive with instant response of the Data Explorer in the InfluxDB Web-UI.

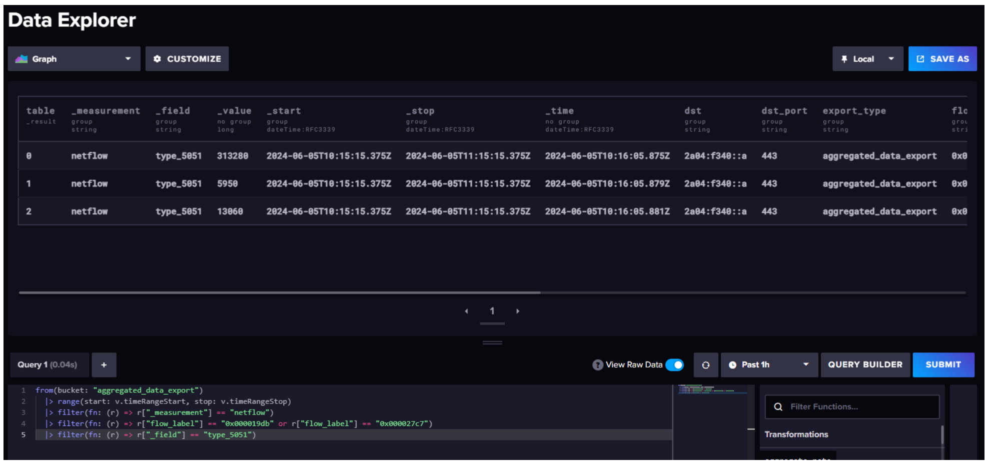


table _result	_measurement group string	_field group string	_value no group long	_start group dateTime:RFC3339	_stop group dateTime:RFC3339	_time no group dateTime:RFC3339	dst group string	dst_port group string	export_type group string	flo grou stri
0	netflow	type_5051	313280	2024-06-05T10:15:15.375Z	2024-06-05T11:15:15.375Z	2024-06-05T10:16:05.875Z	2a04:f340::a	443	aggregated_data_export	0x0
1	netflow	type_5051	5950	2024-06-05T10:15:15.375Z	2024-06-05T11:15:15.375Z	2024-06-05T10:16:05.879Z	2a04:f340::a	443	aggregated_data_export	0x0
2	netflow	type_5051	13060	2024-06-05T10:15:15.375Z	2024-06-05T11:15:15.375Z	2024-06-05T10:16:05.881Z	2a04:f340::a	443	aggregated_data_export	0x0

```
from(bucket: "aggregated_data_export")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "netflow")
  |> filter(fn: (r) => r["flow_label"] == "0x000019db" or r["flow_label"] == "0x000027c7")
  |> filter(fn: (r) => r["_field"] == "type_5051")
```

Figure 2.5.: InfluxDB Web UI - Script Editor

2.4. Dashboard - Grafana

We use for the deployment as described in section 2.5.1.3 the latest Docker image of grafana and configure Grafana with predefined environment variables as described in section 2.5.2 during the initial startup process. In the following subsection it's described how the provisioning of the datasources and the predefined dashboards work. It also describes two sample queries, how they are built, and what the end result as dashboard looks like.

2.4.1. Provisioning

The following two subsection describes the provisioning of Grafana's datasource and the dashboards.

Information

Further information and configuration options for the provisioning are described on the official Grafana documentation.

2.4.1.1. Datasource

A YAML file with the name *datasource.yaml* is located in the folder *grafana/provisioning/datasources*. The parent folder *grafana/provisioning* will be mapped to the grafana container's path */etc/grafana/provisioning*. The datasource config file below, shows that the *influxdb container* is set as data source. All the required parameters that are necessary for correct provisioning of the data source are shown in the following data source config file. The *deleteDatasources* option will delete the existing datasource with the name *influxdb* if it exists, before configuring the new one to make sure it always takes the correct *influxdb* as datasource.

Listing 2.9: Grafana - Provisioning Datasources Settings

```
deleteDatasources:
    - name: influxdb
      orgId: 1

datasources:
    - name: influxdb
      type: influxdb
      access: proxy
      orgId: 1
      url: http://<IP_ADDRESS>:8086
      jsonData:
        version: Flux
        organization: <ORGANIZATION>
        defaultBucket: default
        tlsSkipVerify: true
      secureJsonData:
        token: <TOKEN>
```

2.4.1.2. Dashboard Settings

A YAML file with the name *dashboard_settings.yaml* is located in the folder *grafana/provisioning/dashboards*. The parent folder, the same folder as before for the datasource config file *grafana/provisioning* will be mapped to the Grafana container's path */etc/grafana/provisioning*. As shown in the config file below the path */var/lib/grafana/dashboards* is specified, where the predefined Grafana dashboards can be saved to be loaded the next time Grafana starts.

Listing 2.10: Grafana - Provisioning Dashboard Settings

```
providers:
    - name: dashboards
        type: file
        updateIntervalSeconds: 5
        options:
            path: /var/lib/grafana/dashboards
            foldersFromFilesStructure: true
```

2.4.2. Flux Query Syntax Elements

Flux is a powerful and flexible data scripting and query language designed specifically for querying, analyzing, and acting on the Influx time-series database developed by InfluxData.

The following query functions were frequently used and further described in the official documentation

range() filters rows based on time bounds

filter() filters data based on conditions defined in a predicate function (fn)

group() regroups input data by modifying group key of input tables

map() iterates over and applies a function to the input rows

drop() removes specified columns from a table

keep() opposite of drop, it returns a stream of tables containing only the specified columns

aggregateWindow() down samples data by grouping data into fixed windows of time and applying an aggregate or selector function to each window

sort() orders rows in each input table based on values in specified columns

join() is used to join streams that have different schemas or the streams come from different data sources

union() is used to join data from the same data source and the same schema

2.4.3. Flux Query Examples

The following two subsections shows each a Flux query that was used to create the corresponding Grafana dashboard. The first query discovers the average flow efficiency index (FEI) over the last 5 minutes per end-to-end connection. The second query determines the probability of the most inefficient hop in the topology of the network simulation. In these two queries and all other developed queries, there is always a filter function to check that no error flag is set.

2.4.3.1. End to End Flow Efficiency Matrix - Query

The End to End Flow Efficiency Matrix or in short Flow Efficiency Heatmap shows per field the average flow efficiency indicator over the last 5 minutes for all flows in that time range from one host to another host. The following Flux query is used to query the relevant data from the InfluxDB to create the Flow Efficiency Heatmap, which is shown in figure 2.7. The following description refers to listing 2.11.

Line 1-3 The bucket, time range and the measurement type is specified.

Line 4 Only valid entries with no errors are selected (flag counters = 0).

Line 5 Only entires with the aggregator type SUM (0x01) are selected.

Line 6-7 Only the fields *type_5051* and *in_packets* are kept and the tables are grouped by *src* and *dst*.

Line 8-9 The *_value* is specified as new pivot and the result of the normalization (line 9) is written to it. The normalization process is necessary because not every flow contains the same amount of packets and if we don't do this step, the flows will not be comparable.

Line 10 The tables are aggregated over 5 minutes and the average will be calculated of the field *_value*

Line 11 Entries with values equal to 0 are removed

Line 12 Only the latest entry of the table is kept, so the latest average over the last 5 minutes will be displayed in the Grafana dashboard.

Line 13 All of the tables are grouped together so that a single series results from the query. (necessary for some Grafana dashboards)

Line 14-15 The corresponding host name is mapped to each ipv6 address which is done for every host present in the simulation network.

Listing 2.11: Flux Query - End to End Flow Efficiency Matrix

```
1  from(bucket: "aggregated_data_export")
2  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
3  |> filter(fn: (r) => r["_measurement"] == "netflow")
4  |> filter(fn: (r) => r["type_5053"] == "0" and r["type_5054"] == "0" and
      r["type_5055"] == "0" and r["type_5056"] == "0")
5  |> filter(fn: (r) => r["type_5052"] == "0x01")
6  |> filter(fn: (r) => r["_field"] == "type_5051" or r["_field"] == "in_packets")
7  |> group(columns: ["src", "dst"])
8  |> pivot(rowKey: ["_time"], columnKey: ["_field"], valueColumn: "_value")
9  |> map(fn: (r) => ({ r with _value: r.type_5051 / r.in_packets }))
10 |> aggregateWindow(every: 5m, fn: mean)
11 |> filter(fn: (r) => r["_value"] != 0)
12 |> last()
13 |> group()
14 |> map(fn: (r) => ({ r with src: if r.src == "2a04:f340::a" then "h01" else r.src }))
15 |> map(fn: (r) => ({ r with dst: if r.dst == "2a04:f340::a" then "h01" else r.dst }))
16 <-- details omitted -->
```

Figure 2.6 shows some of the raw data that resulted from the query above.

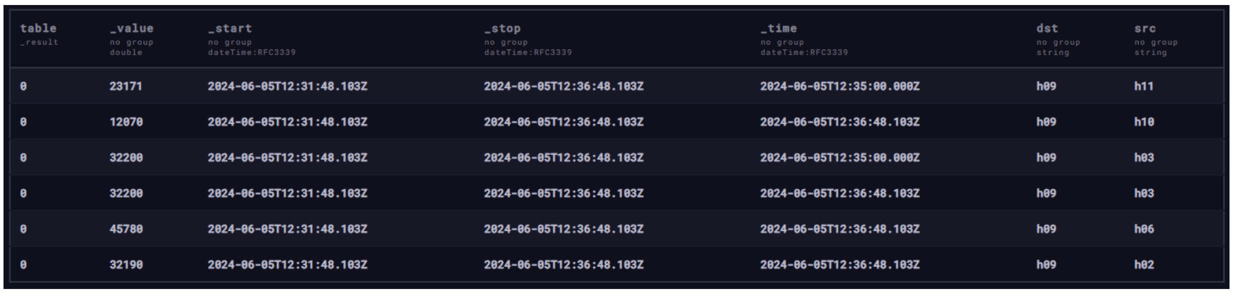


table	_value	_start	_stop	_time	dst	src
_result	no group double	no group dateTime:RFC3339	no group dateTime:RFC3339	no group dateTime:RFC3339	no group string	no group string
0	23171	2024-06-05T12:31:48.103Z	2024-06-05T12:36:48.103Z	2024-06-05T12:35:00.000Z	h09	h11
0	12070	2024-06-05T12:31:48.103Z	2024-06-05T12:36:48.103Z	2024-06-05T12:36:48.103Z	h09	h10
0	32200	2024-06-05T12:31:48.103Z	2024-06-05T12:36:48.103Z	2024-06-05T12:35:00.000Z	h09	h03
0	32200	2024-06-05T12:31:48.103Z	2024-06-05T12:36:48.103Z	2024-06-05T12:36:48.103Z	h09	h03
0	45780	2024-06-05T12:31:48.103Z	2024-06-05T12:36:48.103Z	2024-06-05T12:36:48.103Z	h09	h06
0	32190	2024-06-05T12:31:48.103Z	2024-06-05T12:36:48.103Z	2024-06-05T12:36:48.103Z	h09	h02

Figure 2.6.: Raw Data Output of the End to End Flow Efficiency Matrix Query

Figure 2.7 shows the Flow Efficiency Heatmap. That's the final result from the query above combined with the Grafana esnet-matrix-panel plugin, that is further described in official documentation.

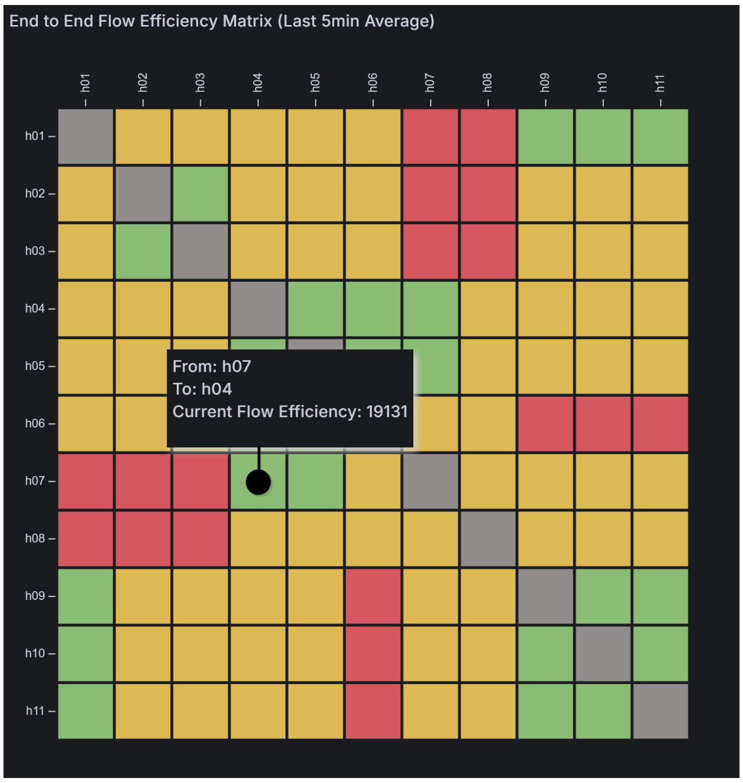


Figure 2.7.: Grafana - End to End Flow Efficiency Matrix (Last 5min Average)

2.4.3.2. Inefficient Hop Discovery (Relative) - Query

The following query identifies the most inefficient node in the simulation network topology. The node with the minimum or maximum hop traversal cost (HTC) is determined during MIN/MAX aggregation. The HTC is the result of accumulating the ingress link LEI, the HEI and the egress link LEI, and this accumulated value is stored in the aggregate field of the IPFIX export. In the simulation network topology, it is possible that multiple paths traverse a switch, but via different ingress and egress ports, resulting in a different HTC value because the LEI of the ingress and egress ports can be different. So we decided to use a query based on a normalized voting strategy to determine the worst or best node. The query is quite complex but necessary because we have to consider how many paths go through the different switches to perform a normalization, otherwise switches that occur in many paths would be weighted more heavily than switches that occur in very few paths, which would almost be neglected in the statistics and thus not detected. Therefore, this large query with subqueries is necessary to get the most accurate result. In the end, we have a statistic, shown in figure 2.9 that shows how likely it is that each node is the least efficient in the simulation topology. The following description refers to listing 2.12.

Line 3 A new stream is assigned to the variable *number_of_paths_via_s01* and the data source bucket is specified.

Line 4-6 The time range and the measurement type is specified and only valid entries are selected (flags = 0).

Line 7-8 Only the field *aggregate* is kept and and the tables are group by the columns *node_01, node_02, node_03, node_04*

Line 9 Only the first (newest) table entry is kept which results of a list of all possible paths (node combinations).

Line 10-11 Its checked if the path (grouped nodes identifies a path) goes via a particular switch.

line 12-13 The tables are grouped to on table and all values will be summarized (count()).

Line 14-15 The field *_value* is renamed and an additional tag is added to identify what this value corresponds to.

Line 19-34 All previously created substreams (*number_of_paths_via_sXX*) are united into a single stream.

Line 36-38 A new stream is assigned to the variable *main* and the data source bucket, the time range and the measurement type is specified.

Line 39-40 Only tables with the aggregator type MAX (0x04) are selected and only valid entries are selected (flags = 0).

Line 41-43 Only the field *aggregate* is kept and and the tables are grouped by the columns *node_01, node_02, node_03, node_04*. An additional field *path* is created based on the information of the node information.

Line 44 Only the specified columns are kept.

Line 45-46 The time is truncated to 1 second and a new field named *timestamp* and the value is set to the value of *_time* because the timestamp will be lost after using the aggregateWindow() function.

Line 47 The tables are grouped by the column *path*.

Line 48-50 Only the latest value per table will be kept, then the tables are grouped by the columns *auxil_data_node_id* and the values will be summarized (count()).

Line 51 The specified columns are dropped.

Line 52 Tables with values equal to 0 are removed.

Line 53-55 The field *_value* is renamed, the tables are grouped to one sorted table.

Line 57-62 The previously created stream *number_of_paths_per_switch* (Line 19) is joined on *auxil_data_node_id == switch_id* with the previously created stream *main* (Line 36).

Listing 2.12: Flux Query - Inefficient Hop Discovery (Relative)

```
import "join"

number_of_paths_via_s01 = from(bucket: "raw_data_export")
    |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
    |> filter(fn: (r) => r["_measurement"] == "netflow")
    |> filter(fn: (r) => r["flags"] == "0")
    |> filter(fn: (r) => r["_field"] == "aggregate")
    |> group(columns: ["node_01", "node_02", "node_03", "node_04"])
    |> limit(n: 1)
    |> map(fn: (r) => ({ r with is_via_switch: if r.node_01 == "1" or r.node_02 == "1"
    or r.node_03 == "1" or r.node_04 == "1" then true else false}))
    |> filter(fn: (r) => r["is_via_switch"] == true)
    |> group()
    |> count()
    |> rename(columns: {_value: "switch_total_path_count"})
    |> map(fn: (r) => ({ r with switch_id: "1"}))

<-- details omitted -->

number_of_paths_per_switch = union(
    tables: [
    number_of_paths_via_s01,
    number_of_paths_via_s02,
    number_of_paths_via_s03,
    number_of_paths_via_s04,
    number_of_paths_via_s11,
    number_of_paths_via_s12,
    number_of_paths_via_s13,
    number_of_paths_via_s14,
    number_of_paths_via_s15,
    number_of_paths_via_s16,
    number_of_paths_via_s17,
    number_of_paths_via_s18,
    ]
)

main = from(bucket: "raw_data_export")
    |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
    |> filter(fn: (r) => r["_measurement"] == "netflow")
    |> filter(fn: (r) => r["aggregator"] == "4")
    |> filter(fn: (r) => r["flags"] == "0")
    |> filter(fn: (r) => r["_field"] == "aggregate")
    |> group(columns: ["node_01", "node_02", "node_03", "node_04"])
    |> map(fn: (r) => ({ r with path: r.node_01 + "-" + r.node_02 + "-" + r.node_03 +
    "-" + r.node_04}))
    |> keep(columns: ["_time", "path", "_value", "auxil_data_node_id"])
    |> truncateTimeColumn(unit: 1s)
    |> map(fn: (r) => ({ r with timestamp: r._time }))
    |> group(columns: ["path"])
    |> aggregateWindow(every: inf, fn: last)
    |> group(columns: ["auxil_data_node_id"])
    |> aggregateWindow(every: inf, fn: count)
    |> drop(columns: ["_time", "_start", "_stop"])
    |> filter(fn: (r) => r["_value"] != 0 and r["auxil_data_node_id"] != "0")
    |> rename(columns: {_value: "switch_discovered_path_count"})
    |> group()
    |> sort(desc: true)

join.inner(
    left: main,
    right: number_of_paths_per_switch,
    on: (l, r) => l.auxil_data_node_id == r.switch_id,
    as: (l, r) => ({l with switch_total_path_count: r.switch_total_path_count}),
)
```

Figure 2.8 shows some of the raw data that resulted from the query above.

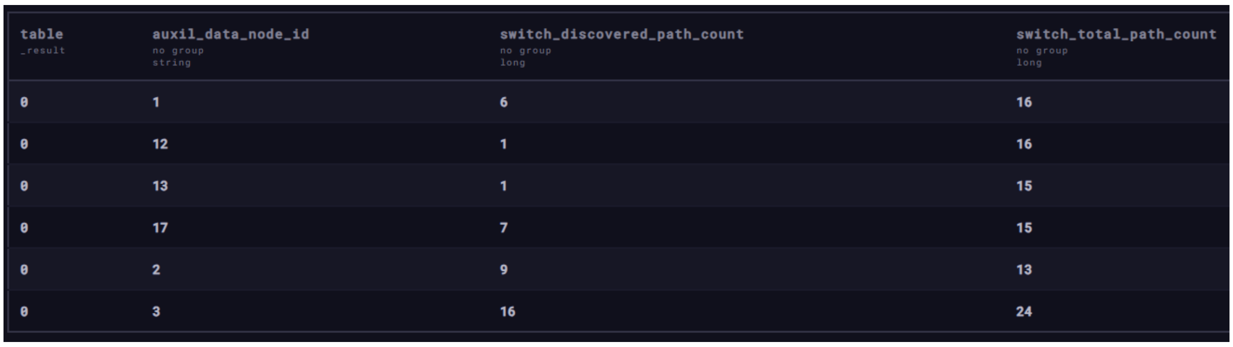


table _result	auxil_data_node_id no group string	switch_discovered_path_count no group long	switch_total_path_count no group long
0	1	6	16
0	12	1	16
0	13	1	15
0	17	7	15
0	2	9	13
0	3	16	24

Figure 2.8.: Raw Data Output of the Inefficient Hop Discovery (Relative) Query

Figure 2.9 shows the Grafana Dashboard - Inefficient Hop Discovery (Relative). This graph shows the percentage of paths for which a particular hop was determined to be the most inefficient. To calculate the percentage of paths, only the paths that traverse through the specific hop are considered as described in the query description in section 2.4.3.2.

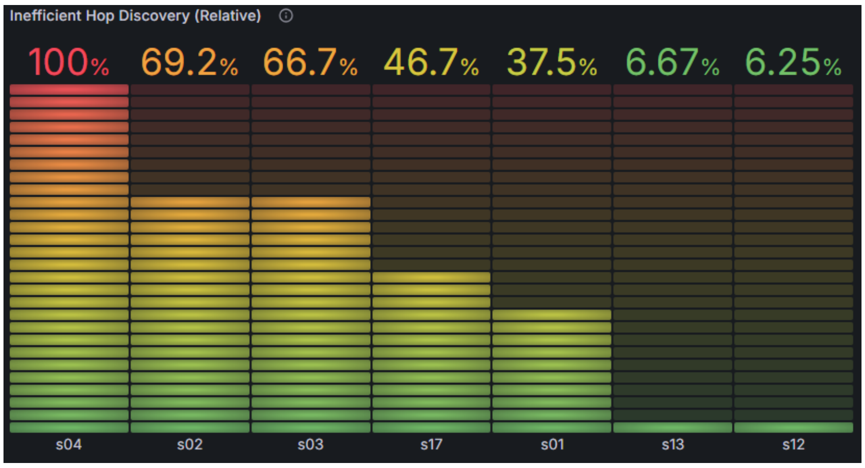


Figure 2.9.: Grafana - Inefficient Hop Discovery (Relative)

2.5. Docker

2.5.1. Docker Compose

Docker Compose is a tool to define and run multi-container applications and it simplifies the control of the application stacks and making it easy to manage services and volumes in a single YAML config file. [5] For the development and evaluation process of the bachelor thesis, we used two different deployment options

local deployment the local deployment was mainly used in the beginning of the bachelor thesis for local development and testing. Each team member had their own virtual machine with the development tools.

server deployment the server deployment is used for long-term monitoring and the execution of network simulations over several days. Additionally the server deployment is used for demonstration purposes and larger network topology can be tested over several days without interruption.

2.5.1.1. InfluxDB

The specification of the *influxdb* service includes the following information:

image Specifies what image is used, for the *influxdb* service the latest image of influxdb will be taken

container_name The container name is set to *influxdb*

ports The *host_port* is 8086 and this port is mapped to the *container_port* 8086

environment The environment tag contains the possible environment variables, those are specified in the *.env* file, further described in section 2.5.2 The different environment variables and the automated setup options are further described in the description of the influxdb image on DockerHub.

volumes The first volume tag entry maps the persistent storage of the influx database to the *influxdb_data* volume, further specified in section 2.5.1.4. The second volume tag entry maps the host folder *./influx_scripts* with bash scripts to the container path *docker-entrypoint-initdb.d*. All bash scripts that are located within this folder are executed by the startup of the *influxdb* container. We use this feature to create additional buckets in the InfluxDB, further described in section 2.3.1

restart The option *unless-stopped* specifies that the container will always restart except when the container was stopped (manually or otherwise)

Listing 2.13: Docker Compose - InfluxDB Service

```
influxdb:
    image: influxdb:latest
    container_name: influxdb
    ports:
        - "8086:8086"
    environment:
        - TZ=Europe/Zurich
        - DOCKER_INFLUXDB_INIT_MODE=setup
        - DOCKER_INFLUXDB_INIT_USERNAME=${INFLUXDB_USERNAME}
        - DOCKER_INFLUXDB_INIT_PASSWORD=${INFLUXDB_PASSWORD}
        - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=${INFLUXDB_TOKEN}
        - DOCKER_INFLUXDB_INIT_ORG=${INFLUXDB_ORG}
        - DOCKER_INFLUXDB_INIT_BUCKET=${INFLUXDB_BUCKET}
    volumes:
        - influxdb_data:/var/lib/influxdb
        - ./influx_scripts:/docker-entrypoint-initdb.d
    restart: unless-stopped
```

2.5.1.2. Telegraf

The specification of the *telegraf* service includes the following information:

image Specifies what image is used, for the *telegraf* service the latest image of telegraf will be taken.

container_name The container name is set to *telegraf*.

environment The environment tag contains the possible environment variables, those are specified in the *.env* file, further described in section 2.5.2.

volumes The *telegraf.conf* file will be mapped from the host system path *./telegraf/telegraf.conf* to the container path */etc/telegraf/telegraf.conf* with read-only mode.

ports The *host_port* is 4739 and this port is mapped to the *container_port* 4739, the */udp* specifies that the port should listen to UDP traffic, which is necessary because the IPFIX export uses the UDP protocol.

depends on This option controls the order of service startup, restart or shutdown. In our scenario Telegraf is started as soon as the InfluxDB container is available.

restart The option *unless-stopped* specifies that the container will always restart except when the container was stopped (manually or otherwise).

Listing 2.14: Docker Compose - Telegraf Service

```
telegraf:
    image: telegraf:latest
    container_name: telegraf
    environment:
        - DOCKER_INFLUXDB_INIT_ORG=${INFLUXDB_ORG}
        - DOCKER_INFLUXDB_INIT_BUCKET=${INFLUXDB_BUCKET}
        - DOCKER_INFLUXDB_INIT_ADMIN_ENABLE=${DOCKER_INFLUXDB_ADMIN_ENABLE:-true}
        - DOCKER_INFLUXDB_INIT_URL=${INFLUXDB_INIT_URL}
        - DOCKER_INFLUXDB_INIT_ADMIN_TOKEN=${INFLUXDB_TOKEN}
    volumes:
        - ./telegraf/telegraf.conf:/etc/telegraf/telegraf.conf:ro
    ports:
        - "4739:4739/udp"
    depends_on:
        - influxdb
    restart: unless-stopped
```

2.5.1.3. Grafana

The specification of the *grafana* service includes the following information:

image Specifies what image is used, for the *grafana* service the latest image of grafana-oss will be taken.

container_name The container name is set to *grafana*.

ports The *host_port* is 300 and this port is mapped to the *container_port* 3000.

environment The environment tag contains the possible environment variables, those are specified in the *.env* file, further described in section 2.5.2. Furthermore the environment variable *GF_INSTALL_PLUGINS* is used to install the additional Grafana plugin *esnet-matrix-panel* that is used to create the End to End Flow Efficiency Matrix, which is shown in figure 2.7.

volumes The first volume tag entry maps the persistent storage of grafana to the *grafana_data* volume, further specified in section 2.5.1.4. The second and third volume tag entry maps the two YAML provisioning configuration file into the container, further described in section 2.4.1.

restart The option *unless-stopped* specifies that the container will always restart except when the container was stopped (manually or otherwise).

depends on This option controls the order of service startup, restart or shutdown. In our scenario Grafana is started as soon as the InfluxDB container is available.

Listing 2.15: Docker Compose - Grafana Service

```
grafana:
    image: grafana/grafana-oss:latest
    container_name: grafana
    ports:
    - "3000:3000"
    environment:
        - GF_SECURITY_ADMIN_USER=${GRAFANA_USERNAME}
        - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_PASSWORD}
        - GF_INSTALL_PLUGINS=esnet-matrix-panel
    volumes:
    - grafana_data:/var/lib/grafana
    - ../global/grafana/dashboards:/var/lib/grafana/dashboards
    - ./grafana/provisioning:/etc/grafana/provisioning
    restart: unless-stopped
    depends_on:
        - influxdb
```

2.5.1.4. Volumes

For the *influxdb* and the *grafana* service, a separate volume will be created for the persistence of its data.

Listing 2.16: Docker Compose - Volumes

```
volumes:
    influxdb_data:
    grafana_data:
```

Information

Unless the Docker volumes are deleted, the above configuration of InfluxDB and Grafana is persistent. (Hint: With *docker volume ls* all the existing Docker volumes will be listed)

2.5.1.5. Delete Volumes

If the intention is to delete the data to start from scratch, the following commands must be executed:

Listing 2.17: Delete Docker Volumes

```
docker volume prune
docker volume rm influxdb_data
docker volume rm grafana_data
```

2.5.2. Environment Variables

The following *.env* file with the specified environment variables is defined in the same directory as the Docker Compose file. The environment variables are loaded and set the first time the three services (*influxdb, telegraf and grafana*) are started.

Listing 2.18: env file

```
INFLUXDB_USERNAME=<USERNAME>
INFLUXDB_PASSWORD=<PASSWORD>
INFLUXDB_ORG=<ORGANIZATION>
INFLUXDB_BUCKET=default
INFLUXDB_INIT_URL=http://<INFLUX_HOST_IP>:8086
INFLUXDB_TOKEN=<TOKEN>
GRAFANA_USERNAME=<USERNAME>
GRAFANA_PASSWORD=<PASSWORD>
```

2.5.3. Network

The Docker Compose file for the local deployment contains the following addition compared to the "standard" server deployment. A network named *efficientNetwork* will be defined in the Docker Compose file.

Listing 2.19: Docker - Network

```
networks:
    efficientNetwork:
      driver: bridge
      ipam:
        driver: default
        config:
          - subnet: <SUBNET_IP_RANGE>
            gateway: <GATEWAY_IP_ADDRESS>
```

All services *influxdb, telegraf* and *grafana* will be attached to this network and a static ip address will be assigned to each service.

Listing 2.20: Docker - Additional Network for each Service

```
networks:
    efficientNetwork:
      ipv4_address: <IP_ADDRESS>
```

3. Configuration Update System

As already specified in the design decisions in section 5 in the elaboration part, the Nornir network automation framework is used to implement the configuration update system. The configuration update system is written in Python.

It depends on the following libraries:

typer Typer is a library for building intuitive and easy-to-use CLI applications based on Python type hints, offering features like automatic help, completion, and minimal code duplication. It also includes a command line tool to run scripts, converting them into CLI applications automatically. Refer to: https://pypi.org/project/typer/

nornir Nornir is a pure Python automation framework intended to be used directly from Python. While most automation frameworks use their own Domain Specific Language (DSL) which you use to describe what you want to have done, Nornir lets you control everything from Python. Refer to: https://pypi.org/project/nornir/

nornir_rich Nonir rich is a plugin for Nornir which enhances the user experience by the addition of fancy terminal user interface (TUI) elements such as a progress bar and pretty print functions for results, failed hosts and the inventory. Refer to: https://pypi.org/project/nornir-rich/

The dependencies are listed inside the *requirements.txt* file.

3.1. Getting Started

To use the configuration update system clone the repository and change the working directory into the root of the git repository.

3.1.1. Installation

The first step is to setup the project by the creation of a virtual environment and the installation of the required dependencies. To do so follow the setup instructions in listing 3.1.

Listing 3.1: Setup Instructions

```
# Create a new virtual environment in the hidden folder .venv
python3 -m venv .venv

# Activate the virtual environment
source .venv/bin/activate

# Install dependencies
pip install -r requirements.txt
```

3.1.2. Configuration

The configuration of the Nornir BMv2 configuration update solution is the next pre-requisite to fulfill before the application can be run.

3.1.2.1. Defaults

Please specify the default values further described below, which apply to your circumstances in the *inventory/defaults.yaml* file.

p4_repo_path The absolute path to the repository containing the p4 source code and the JSON BMv2 runtime configuration files.

runtime_path The relative path starting from the *p4_repo_path* to the BMv2 runtime directory.

mininet_host The IP address or hostname of the host running the virtual network with the BMv2 software switches.

checksum_file The relative path to the file which will hold the checksums of the deployed configurations.

Information

The checksums of the deployed configuration files are used to check whether the same configurations previously pushed are pushed again. This avoids confusion of the developer in case the resource definition was updated but the execution of the configuration generator was forgotten.

```
The runtime configurations were not modified since the previous run at 2024-05-18
14:56:27.393337 on 127.0.0.1. Do you want to push the configurations anyway? [Y/n]:
```

3.1.2.2. Inventory

Please specify the connection parameters to the BMv2 software switches. To do so use the format shown in listing 3.2.

Information

The inventory file to be used can be specified in the file *config.yaml* by setting the option **host_file** as part of the *SimpleInventory* configuration. The path to the inventory file must be relative to the project root directory.

Listing 3.2: Example Nornir Inventory

```
---
s01:
  port: 50051
  data:
    device_id: 0
    runtime_file_name: "s01-runtime.json"
s02:
  port: 50052
  data:
    device_id: 1
    runtime_file_name: "s02-runtime.json"
<-- further specifications omitted -->
```

⚠ Warning

Make sure that the **device ID** and **port** are specified correctly. Otherwise the connection attempt will be rejected or the connection will be made to a wrong switch. In case the Mininet setup provided by p4lang is used the device IDs are specified ascending starting by 0 and the port numbers starting by 5051 in the order the switches are defined in the topology file.

3.2. Implementation Details

This chapter will go into some more detail about the components of the configuration update system and how they interact with each other. The explanations present in this chapter refer to figure 3.1.

3.2.1. Update Process

The configuration update process is an optional process. It is triggered by the network operator in case the person wants to modify the configuration of the BMv2 software switches while they are up and running.

⚠ Warning

It is very important to first change the network resource definition and then regenerate the configurations inside the **network virtualisation system** before the configuration update process is triggered.

1. The **network operator** triggers the configuration update executing the **task manager**.
2. The **task manager** reads the **configuration**.
3. The **task manager** reads the **inventory**.
4. The **task manager** calculates the SHA-256 signatures of the configuration files to provision and verifies if the files where changed since the last run. To do that the signatures from the previous run are loaded from the **configuration file signatures** and compared with the current signatures.
5. The **task manager** starts the **task** to update the BMv2 configuration for each BMv2 software switch specified in the **inventory**.
6. The **task** loads the pre-generated BMv2 runtime configuration for the specified host from the **network virtualization system**.
7. The **task** calls the **gRPC client application** and hands the information about the target BMv2 software switch and the corresponding configuration over.
8. The **gRPC client application** establishes a connection to the configuration endpoint of the BMv2 software switch inside the **network virtualization system**.
9. The **gRPC client application** uploads the updated configuration to the BMv2 software switch inside the **network virtualization system**.

3.2.2. Configuration Validation

The configuration validation is based on the SHA-256 signature algorithm. It ensures that the network operator is notified in case the BMv2 configuration files did not change. If the network operator tries to push the same configuration twice he will receive the message `The runtime configurations were not modified since the previous run at 2024-05-18 14:56:27.393337 on 127.0.0.1. Do you want to push the configurations anyway? [Y/n]:`

The signature file is formatted in json and follows the structure shown in listing 3.3 and an entry is identified with the tuple made of the the Mininet host and the target inventory file. In case an entry exists the signatures are loaded and compared and after the successful execution the signatures in that specific entry are overwritten. In case no entry was found a new entry is added after the successful execution of the configuration update task.

Listing 3.3: Signature File Structure

```
{
  "127.0.0.1": {
    "inventory/hosts_simulation.yaml": {
      "timestamp": "2024-05-18 14:56:27.393337",
      "checksums": {
        "/[...]/s01-runtime.json": "259db53[...]d1",
        "/[...]/s02-runtime.json": "d1eaf16[...]fd",
        <-- entries omitted -->
        "/[...]/s18-runtime.json": "770e01b[...]b1"
      }
    }
  },
  "ba-rb-rf-0.network.garden": {
    "inventory/hosts_simulation.yaml": {
      "timestamp": "2024-05-17 10:50:26.236140",
      "checksums": {
        "/[...]/s01-runtime.json": "259db53[...]d1",
        "/[...]/s02-runtime.json": "d1eaf16[...]fd",
        <-- entries omitted -->
        "/[...]/s18-runtime.json": "770e01b[...]b1"
      }
    }
  }
}
```

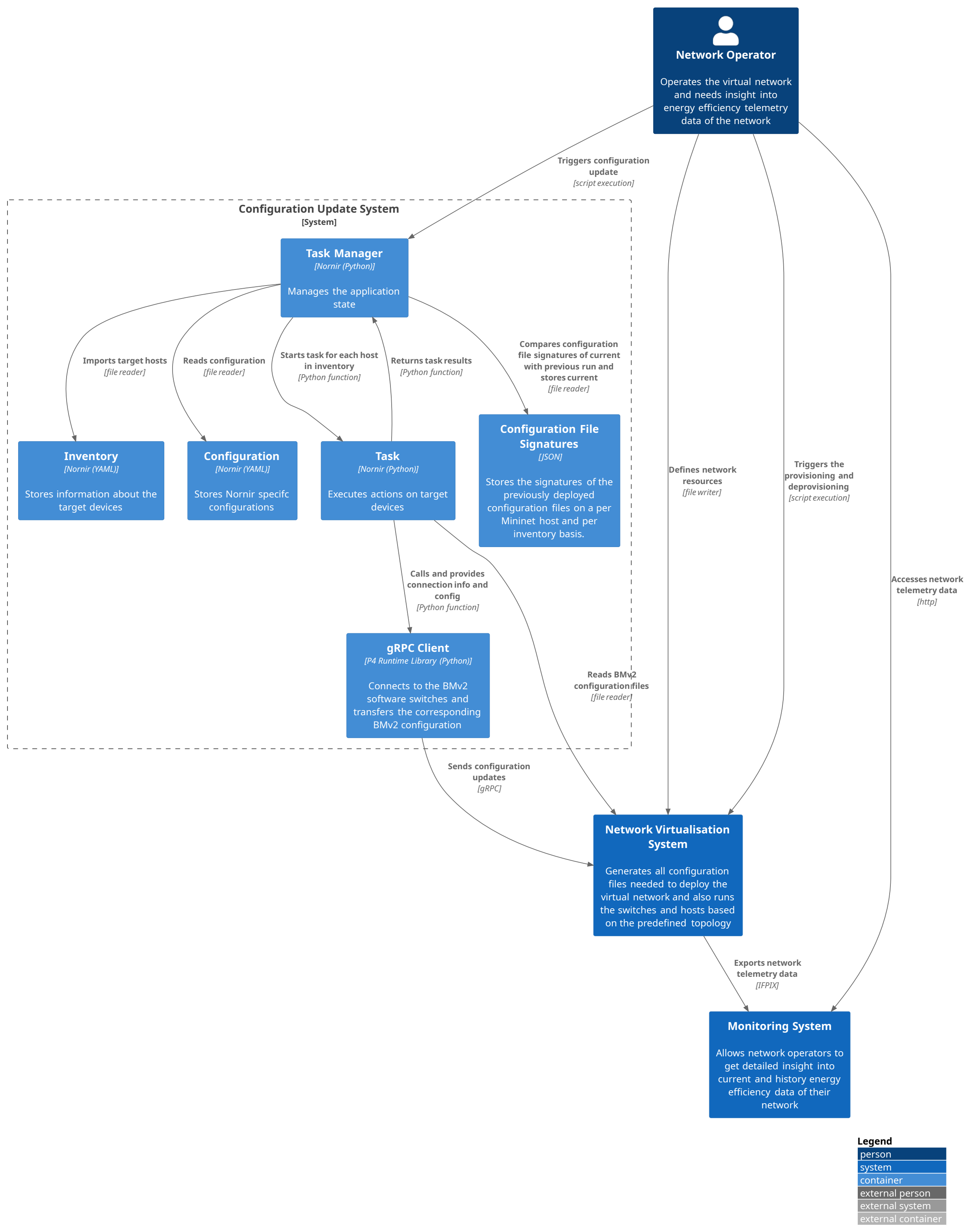

Network Operator
Operates the virtual network and needs insight into energy efficiency telemetry data of the network
Triggers configuration update
[script execution]
Configuration Update System
[System]
Task Manager
[Nornir (Python)]
Manages the application state
Imports target hosts
[file reader]
Reads configuration
[file reader]
Starts task for each host in inventory
[Python function]
Returns task results
[Python function]
Compares configuration file signatures of current with previous run and stores current
[file reader]
Inventory
[Nornir (YAML)]
Stores information about the target devices
Configuration
[Nornir (YAML)]
Stores Nornir specifc configurations
Task
[Nornir (Python)]
Executes actions on target devices
Configuration File Signatures
[JSON]
Stores the signatures of the previously deployed configuration files on a per Mininet host and per inventory basis.
Defines network resources
[file writer]
Triggers the provisioning and deprovisioning
[script execution]
Accesses network telemetry data
[http]
Calls and provides connection info and config
[Python function]
gRPC Client
[P4 Runtime Library (Python)]
Connects to the BMv2 software switches and transfers the corresponding BMv2 configuration
Reads BMv2 configuration files
[file reader]
Sends configuration updates
[gRPC]
Network Virtualisation System
Generates all configuration files needed to deploy the virtual network and also runs the switches and hosts based on the predefined topology
Exports network telemetry data
[IFPIX]
Monitoring System
Allows network operators to get detailed insight into current and history energy efficiency data of their network
Legend
person
system
container
external person
external system
external container

Figure 3.1.: Component Diagram Configuration Update System

4. Data Plane Optimizations

The P4 data plane is based on the implementation elaborated during the term paper in the previous semester. During the bachelor thesis we applied optimizations regarding the structure and the reduction of processing overhead. Furthermore we implemented proper error handling for the IOAM aggregation trace option to ensure that there is a possibility to ignore error prone data in the energy efficiency statistics of the network. Finally the data plane was extended with an additional processor which prepares the network telemetry data to be exported with IPFIX. As a key component the processor is responsible for the export and contains an extern function which passes the prepared data to the control plane.

Information

In order to be able to follow the explanations of the following sections it is advantageous that you have read the P4 related chapters of the term paper. In case you didn't, you should still get a high level idea of how the data plane operates.

4.1. P4 Introduction

Before going into details about the data plane implementation it is important to review some important concepts related to P4 and data plane programming in general.

It is the data plane's responsibility to forward data packets which arrive at a specific ingress port to specific egress port(s) and perform corresponding header field updates (e.g. decrement the TTL IP header field). The information about where to send a specific packet is stored in control plane tables which can be queried by the data plane. The data plane uses information inside the header field to query the control plane tables. The process of getting information from the control plane tables using queries is called match action. The result of a match action is always the execution of an action inside the data plane using the values retrieved by the specific match from the control plane. Actions are similar to functions in other programming languages with the limitation that only sequential code is allowed which means that neither calls to other actions nor loops can be present inside an action.

One can think of control plane tables as being the link between the data plane and the control plane. It is the way how the control plane exposes its information to the data plane.

The concept described above is part of the Protocol Independent Switch Architecture (PISA) depicted in figure 4.1. As shown, the PISA architecture is composed of a programmable parser which is responsible to map the header fields of an incoming packet to local struct-like data structures, an ingress pipeline, a buffer and an egress pipeline. Pipelines are units composed of one or more sequential match actions. Once the packet leaves the egress pipeline it enters the deparser (not shown on figure 4.1) which emits the internal header data structures back into the packet being sent out of the determined egress port.

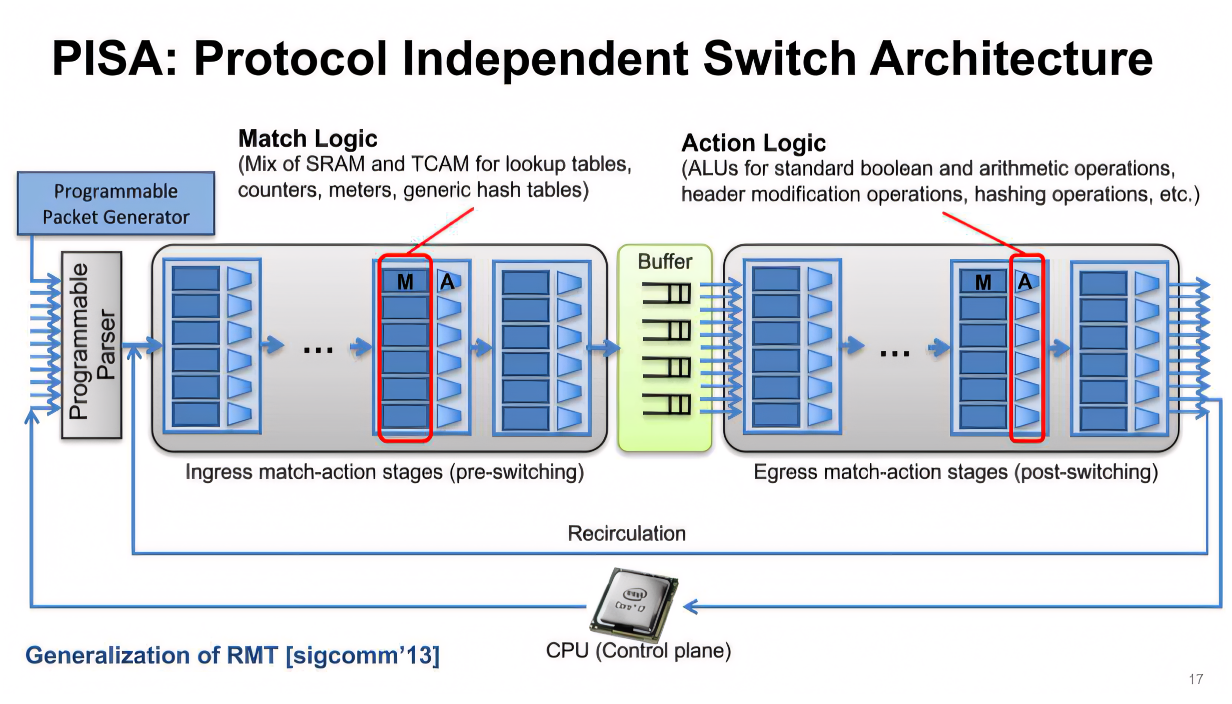

PISA: Protocol Independent Switch Architecture
Match Logic
(Mix of SRAM and TCAM for lookup tables, counters, meters, generic hash tables)
Action Logic
(ALUs for standard boolean and arithmetic operations, header modification operations, hashing operations, etc.)
Programmable Packet Generator
Programmable Parser
M
A
Buffer
Ingress match-action stages (pre-switching)
Egress match-action stages (post-switching)
Recirculation
CPU (Control plane)
Generalization of RMT [sigcomm'13]
17

Figure 4.1.: PISA Architecture [11]

In our P4 data plane implementation we put the IPv4 and IPv6 forwarding logic into the ingress pipeline. The logic to trace the path, determine and aggregate efficiency values and finally the IPFIX export are part of the egress pipeline further described in the next section.

4.2. The Modifications at a Glance

This section quickly summarizes the main modifications which were made to the data plane in the course of this bachelor thesis compared to the preceding term paper.

Extension Header Initialization The initialization process of the extension header which carries the network telemetry data inside the IOAM options has been modified to not only set the corresponding header fields but also save the data which remains constant as metadata so that it reusable by subsequent processors. This reduces the overall amount of table lookups and increases the overall performance of the data plane implementation.

HEI Processing The concept how the HEI is being processed was fundamentally changed in the bachelor thesis. In the term paper the HEI was computed in the data plane based on raw efficiency values retrieved from the control plane. The main processing load is now handled by the control plane and the data plane retrieves the value from a dedicated control plane table. More details are available in section 4.4.

IOAM Aggregation Option Error Handling The specification of the IOAM Aggregation Option in draft-cxx-ippm-ioamaggr-00 defines four error flags. [3] These flags are now being set as part of the data plane operation in case an error occurs. More details are available in section 4.5.

IPFIX Export The data plane has been extended with an extern function to trigger the export of network telemetry data using IPFIX. It is documented in section 4.6.

4.3. Egress Pipeline Structure

The egress pipeline elaborated as part of this bachelor thesis includes five processors with dedicated responsibilities. The egress pipeline definition inside the *main.p4* file is depicted in listing 4.1. As can be seen the egress pipeline's processors are only executed in case the IPv6 header is valid. The reason for that is, because the IPv6 Hop-by-Hop Options extension header is used to store IOAM network telemetry data which is obviously only available in IPv6. It would be possible to carry IOAM data with IPv4 inside the Options field defined in RFC 791 but this was defined to be out of scope for this thesis.

Listing 4.1: Data Plane Egress Pipeline

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {

    apply {
        if (hdr.ipv6.isValid()) {
            // Initialize IOAM protocol related header in IPv6 extension header
            process_ioam_init.apply(hdr, meta, standard_metadata);

            // IOAM Tracing
            process_ioam_tracing.apply(hdr, meta, standard_metadata);

            // Efficiency Indicator
            process_efficiency_indicator.apply(hdr, meta, standard_metadata);

            // IOAM Aggregation
            process_ioam_aggregation.apply(hdr, meta, standard_metadata);

            // IPFIX Export
            process_ipfix_export.apply(hdr, meta);
        }
    }
}
```

Before going into more detail about each individual processor the following elements need to be clarified referring to listing 4.1.

hdr Is a reference to the internal header data structures which where initialized by the programmable parser.

meta Is a reference to custom metadata used to store information which needs to be accessible by multiple processors.

standard_metadata Is a reference to metadata specific to the BMv2 v1 switch model.

4.3.1. Processor Responsibilities

The following description highlights the key functionalities and responsibilities of each individual processor. The processor implementations are located in *includes/<processor_name>.p4*.

ioam_init This is the first processor being called. It is only executed if the IPv6 Hop-by-Hop Options header is not yet initialized. This is typically the case on an ingress node. It has the following responsibilities:

- Initialize the IPv6 Hop-by-Hop Options extension header.

- Initialize the IOAM Pre-allocated Trace Option header carried by the Hop-by-Hop Options extension header.
- Initialize the IOAM Aggregation Trace Option header carried by the Hop-by-Hop Options extension header. As part of the initialization the aggregator to use is selected dynamically further described in section 4.4.2.1.
- Initialize the PadN Option to ensure that the IPv6 Hop-by-Hop Options extension header is aligned to a multiple of 8 octets as specified in RFC 2460.
- Initialize all IOAM related custom metadata fields which can then be used by subsequent processors. This reduces the number of control plane table lookups because the required information is already available.

ioam_tracing This processor is responsible to trace a node by adding the ID of the node to the pre-allocated node list and updating related header fields. In case there is no more space left in the pre-allocated node list the overflow bit is set instead.

efficiency_indicator In this processor the HEI and LEI are queried from the control plane. The resulting value is stored in the aggregate metadata field. More details about this processor is available in section 4.4.

ioam_aggregation This processor retrieves from the metadata and performs the corresponding aggregations. Results are stored in the IOAM Aggregation Option header fields. More details about this processor is available in section 4.4.

ipfix_export This processor extracts and reformats the relevant information. It then calls the extern function to pass the relevant data to the control plane which then takes care to export the network telemetry data using IPFIX. More details about this processor is available in section 4.6.

4.4. Efficiency Indicator Processing

The efficiency indicator processing overhead for the data plane has been reduced significantly with the chosen approach in the bachelor thesis. The following sections will compare the efficiency indicator processing implemented in the term paper with the implementation elaborated in the follow-up bachelor thesis.

4.4.1. Term Paper Implementation

In the approach chosen in the term paper the calculation of the HEI is in the responsibility of the data plane. Without going into mathematical details the HEI calculation process depicted in figure 4.2 is a five step approach repeated for every single packet further described below.

1. The values of the energy efficiency parameters are retrieved from the control plane. Those energy efficiency parameters might be the current power usage, bandwidth and amount of renewable energy available on the current hop. Each retrieval requires a lookup in the control plane.
2. The energy efficiency values might be out of an arbitrary range and in order to be able to map multiple values to a single efficiency indicator the values need to be normalized to a common range so that they are directly comparable.

3. Some of the values may behave inverse and need to be inverted inside the common range. This is the case for values which are better in regards to the efficiency if the value is higher. An example inverse value is the amount of renewable energy available where a higher number indicates that more energy is consumed from renewable sources which influences the efficiency of the hop in a positive way.

4. Some values may be more important than others and weights must be applied.

5. The steps above are repeated for each parameter value. Finally each result needs to be summed up to get the HEI value of the current hop.

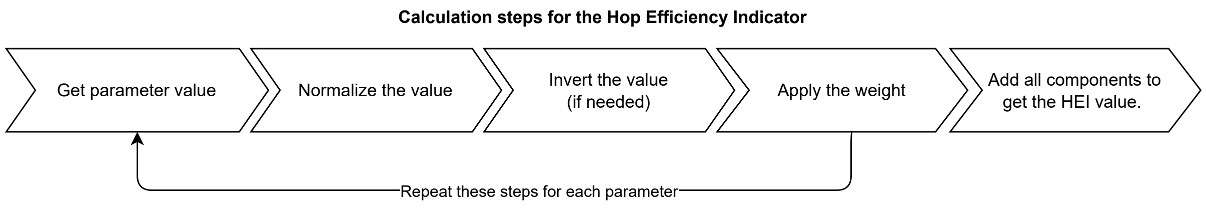

Calculation steps for the Hop Efficiency Indicator
Get parameter value
Normalize the value
Invert the value (if needed)
Apply the weight
Add all components to get the HEI value.
Repeat these steps for each parameter

Figure 4.2.: Term Paper HEI Calculation Process

The HEI value is then passed to the IOAM Aggregation Option implementation to update the aggregate field accordingly.

Information

The mathematical operations used to calculate the HEI value are limited to addition, subtraction and bit shifting to avoid significant performance losses during the HEI processing.

4.4.2. Bachelor Thesis Implementation

In the approach chosen in the bachelor thesis the calculation of the HEI is in the responsibility of the control plane. This has mainly two advantages.

- The efficiency indicator processing is decoupled from the forwarding pipeline. The value can be reprocessed by the control plane from time to time to consider the current environmental conditions. This increases the performance of the forwarding pipeline because the processing overhead for each individual data packet is reduced.
- Multiple efficiency indicators can be made available to the data plane simultaneously. This avoids the information loss caused by the combination of multiple component values to one indicator value in the solution proposed in the term paper. In a multi efficiency indicator scenario the ingress node would randomly select which indicator to use for this specific data packet.

Information

Regarding the second advantage, the dynamic selection of efficiency indicators is currently not implemented in the data plane, but the simultaneous collection of different indicator types within the same flow is supported by the IPFIX cache implementation in the control plane.

The apply block of the efficiency indicator processor in listing 4.2 has been simplified to three statements. The first one queries the current hop efficiency indicator value using the IOAM data param as key and adds the value to the aggregate metadata field. The second statement queries the efficiency of the ingress link using the number of the ingress port as key and adds the value to the aggregate metadata field. Finally the third statement does the same thing as the second one but of the egress interface. The resulting value is the so called Hop Traversal (HTC) Cost which is added to the IOAM Aggregation Option aggregate field in the ioam aggregation processor which is the next element of the egress pipeline.

Listing 4.2: Apply Block Efficiency Indicator Processor

```
// Efficiency Indicator Processing
get_hop_efficiency_indicator.apply();
get_ingress_link_efficiency_indicator.apply();
get_egress_link_efficiency_indicator.apply();
```

4.4.2.1. Aggregator Selection

The dynamic aggregator selection on the ingress node is implemented as part of the *ioam_init* processor. The reason why the indicator is selected randomly is to make the data in the statistics more meaningful, as information on the most and least efficient routers and path statistics are available at all times.

To decide which aggregator to use the last two bits of the IPv6 payload length are taken into consideration. The mapping from bit combination to the aggregator can be configured in the resource definition yaml file. In our simulation network the following mapping is deployed on all routers.

Listing 4.3: Aggregator Mapping

```
aggregators: # 1 = SUM / 2 = MIN / 4 = MAX
  - 1 # selected if last two bits of payload size are [00]
  - 2 # selected if last two bits of payload size are [01]
  - 1 # selected if last two bits of payload size are [10]
  - 4 # selected if last two bits of payload size are [11]
```

In the *ioam_init* processor the key to query the resulting control plane table defined in listing 4.3 is initialized with the action in listing 4.4.

Listing 4.4: Initialization IOAM Aggregator Selector

```
action init_ioam_aggregator_selector() {
    meta.ioamAggrMeta.aggregator_selector = (bit<2>) hdr.ipv6.payloadLen & 0b11;
}
```

4.5. IOAM Aggregation Option Error Handling

The IOAM Aggregation Option specification in draft-cxx-ippm-ioamaggr-00 specifies four header fields. [3]

Flag 1 Aggregator not supported

- Set on all nodes if the aggregator specified in the ioamAggr metadata is not implemented.

Flag 2 Unsupported IOAM data parameter

- Set on the ingress node if the specified data param in the configuration is not available in the lookup table.
- Set on the intermediary nodes if the specified data param in the ioam header is not available in the lookup table.

Flag 3 Unsupported Namespace

- Set on intermediary nodes if the namespace specified in the metadata differs from the namespace defined in the header.

Flag 4 Any other error

- Set on all nodes in case the Link Efficiency Indicator (LEI) is undefined for the ingress or egress interface.
- Set on all nodes in case of an overflow of the aggregator representing the Path Efficiency Indicator (PEI).

In the *ioam_init* and the *efficiency_indicator* processors, boolean fields in the custom metadata are set in case errors listed above occur.

The apply block of the *ioam_aggregation* processor in listing 4.5 has been adjusted to check for the specified error conditions. As can be seen the processor is only executed in case the IOAM Aggregation Option header is valid and the flags field is set to 0. Like this the IOAM Aggregation Option header fields are not touched anymore after an error has occurred.

Listing 4.5: Apply Block IOAM Aggregation Processor

```
if (hdr.ioam_a_ioam_aggregation.isValid() && hdr.ioam_a_ioam_aggregation.flags == 0 ) {
    if (hdr.ioam_a_ioam_aggregation.namespaceID != meta.ioamMeta.namespaceID) {
        set_flag(IOAM_FLAG_UNSUPPORTED_NAMESPACE);
    }
    if (meta.ioamAggrMeta.dataParamError == 1) {
        set_flag(IOAM_FLAG_UNSUPPORTED_DATA_PARAM);
    }
    if (meta.ioamAggrMeta.otherError == 1) {
        set_flag(IOAM_FLAG_OTHER_ERROR);
    }
    if (hdr.ioam_a_ioam_aggregation.flags == 0) {
        switch (hdr.ioam_a_ioam_aggregation.aggregator) {
            IOAM_AGGREGATOR_SUM: {ioam_aggr_sum();}
            IOAM_AGGREGATOR_MIN: {ioam_aggr_min();}
            IOAM_AGGREGATOR_MAX: {ioam_aggr_max();}
            default: {set_flag(IOAM_FLAG_UNSUPPORTED_AGGREGATOR);}
        }
    }
}
```

Additionally the action *set_flag* was defined shown in listing 4.6. It sets the flag according the given bitmask and updates the node id in the IOAM Aggregation Option header to indicate the node where the error has occurred.

Listing 4.6: Set Flag Action

```
action set_flag(ioamFlag_t flag) {
    hdr.ioam_a_ioam_aggregation.flags = hdr.ioam_a_ioam_aggregation.flags | flag;
    hdr.ioam_a_ioam_aggregation.auxilDataNodeID = meta.ioamMeta.nodeID;
}
```

4.6. IPFIX Export

One of the main goals in this bachelor thesis is the extension of the programmable switches with IPFIX export capabilities. The actual IPFIX implementation is done in the control plane and is documented in chapter 1. But somehow the data of interest, in this case the network telemetry data carried in the data packet in the IOAM Aggregation Option header, must be passed by the data plane to the IPFIX implementation in the control plane. We already covered the use case that the data plane retrieves data from the control plane by table lookups. But in this scenario data must be transferred in the opposite direction. For this use case extern functions come into play.

4.6.1. Extern Function

An extern function is used to expose control plane functionality to the data plane. Data available in the data plane (e.g. values of specific header fields) can be passed to the control plane as arguments to function parameters of the extern function.

In order to use an extern function in P4 it must be declared at the beginning of the program. In case the function name and number of arguments does not match the function definition in the control plane the BMv2 programmable switch throws an error on startup. Listing 4.7 contains the declaration of the extern function to call the IPFIX extension in the control plane. The declaration can be found in the file *includes/externs.p4*.

Listing 4.7: Extern Function Declaration

```
extern void ProcessEfficiencyIndicatorMetadata(
                                        in ioamNodeID_t nodeID,
                                        in flowKey_t flowKey,
                                        in flowLabel_t flowLabel,
                                        in ip6Addr_t srcIPv6,
                                        in ip6Addr_t dstIPv6,
                                        in bit<16> sourceTransportPort,
                                        in bit<16> destinationTransportPort,
                                        in bit<24> indicatorID,
                                        in ioamAggregate_t indicatorValue,
                                        in bit<8> indicatorAggregator,
                                        in bit<768> raw_ipv6_header
                                    );
```

return type The function has the return type of void which means that nothing is returned. This characteristic is advantageous because after the data plane called the extern function it can carry on directly and there is no need that it needs to wait for the termination of the function in order to retrieve the result.

parameters The parameters specified are the header fields of interest to identify the flow, end hosts and to export efficiency indicator data carried by the IOAM Aggregation Option header fields. The last parameter is used for the raw export only and transfers the whole IPv6 header including all extension headers to the control plane.

Information

The *in* keyword is part of the parameter type definition and defines the direction of the parameter. The direction *in* indicates that this parameter is an input that cannot be modified. [12] The direction *out* indicates that this parameter is an output whose value is undefined initially but can be modified. [12] The direction *out* for parameters is not allowed for extern functions.

4.6.2. Processor

The implementation of the *ipfix_export* processor is in the file *includes/ipfix_export.p4* The IPFIX export is triggered in case the IOAM Aggregation Option header is valid and the route type is set to 0 which means that it is a directly connected route. A route type of 0 indicates that the router is the egress node of the current path. Listing 4.8 shows the apply block of the IPFIX processor which calls the perform ipfix export action given the mentioned conditions.

Listing 4.8: IPFIX Export Processor Apply Block

```
apply {
    // Perform IPFIX export on last hop only
    if (hdr.ioam_a_ioam_aggregation.isValid() && meta.forwardingMeta.routeType == 0) {
        perform_ipfix_export();
    }
}
```

The perform IPFIX export action prepares the raw export binary blob with the use of the binary concatenation operator. Once the 768 bit binary blob is ready the extern function is called passing the corresponding header fields and the raw export binary blob as arguments. The two different IPFIX export mechanism for the aggregated data and raw data export are further described in detail in section 1.4.2. Listing 4.9 show the call of the extern function.

Listing 4.9: Call of Extern Function

```
// Pass all values to control plane by calling the extern function
ProcessEfficiencyIndicatorMetadata(
    meta.ioamMeta.nodeID,
    flowKey,
    hdr.ipv6.flowLabel,
    hdr.ipv6.srcAddr,
    hdr.ipv6.dstAddr,
    hdr.udp.srcPort,
    hdr.udp.dstPort,
    hdr.ioam_a_ioam_aggregation.dataParam,
    hdr.ioam_a_ioam_aggregation.aggregate,
    hdr.ioam_a_ioam_aggregation.aggregator,
    hdr.ioam_a_ioam_aggregation.flags,
    raw_full_ipv6_header);
```

5. Wireshark Dissector

As specified in the requirements specifications in US4 in section 2 in the inception phase a Wireshark dissector for the IOAM aggregation option shall be developed in the scope of this bachelor thesis.

A Wireshark dissector is a component or plugin used in Wireshark, to interpret and display the details of network protocol data. Wireshark captures packets traveling across a network and a dissector's role is to analyze the captured data according to a specific protocol's format.

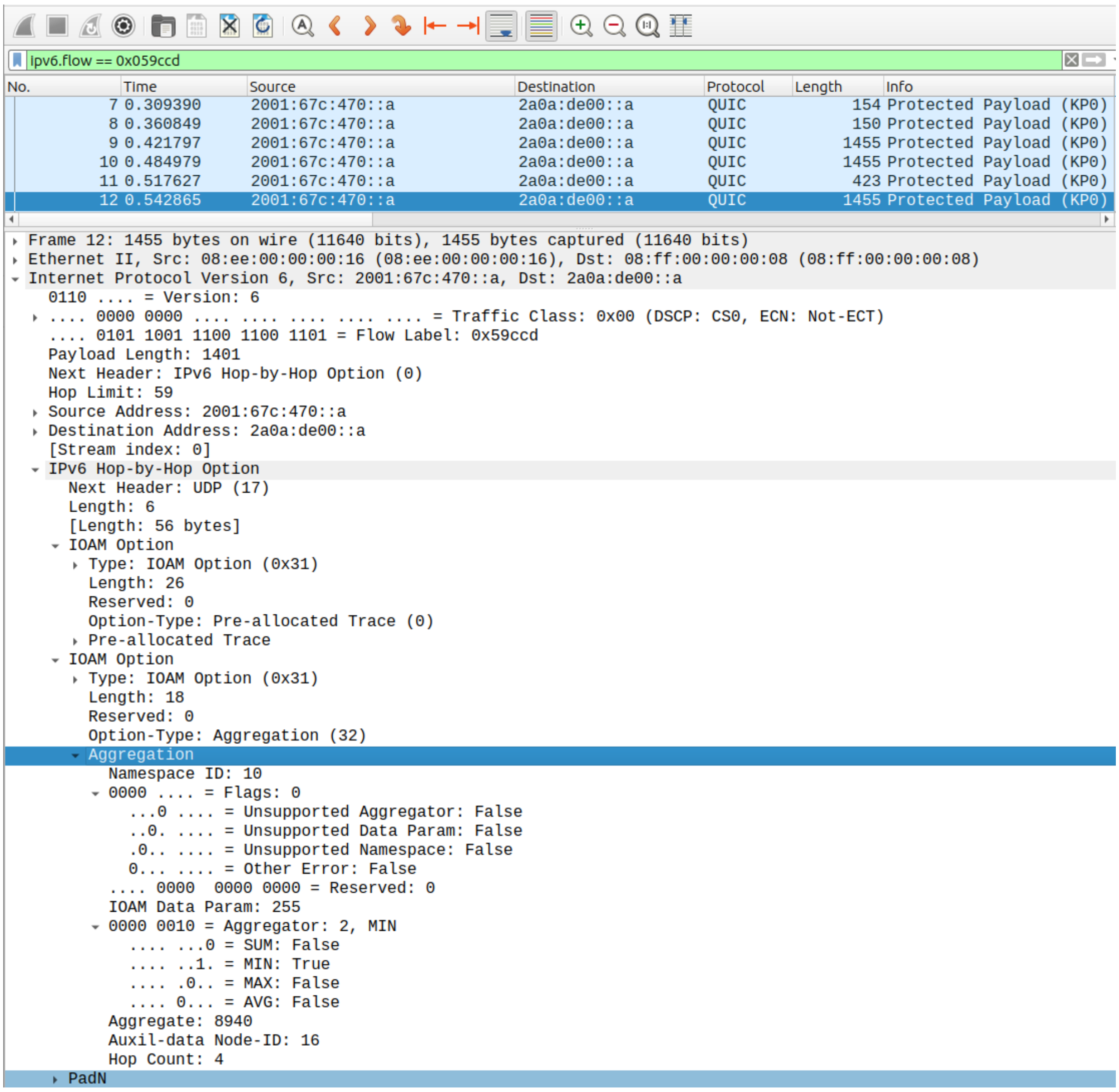
Capture Wireshark captures network packets in real time or from a saved file.

Identification The dissector identifies the protocol of each packet.

Dissection The dissector breaks down the packet into its constituent parts based on the protocol's structure.

Presentation It then presents this information in a human-readable format, showing fields and values, which helps users analyze the packet's contents and troubleshoot network issues.

Figure 5.1 is a screenshot of a packet dissected with a version of Wireshark which contains our IOAM aggregation option dissector. One can easily identify the header fields according to draft-cxx-ippm-ioamaggr-00. [3] Figure 5.2 shows exactly the same packet in a version of Wireshark which does not support the dissection of the IOAM aggregation option. It is much harder to identify the relevant information out of the raw byte-stream than it is in the dissected version.

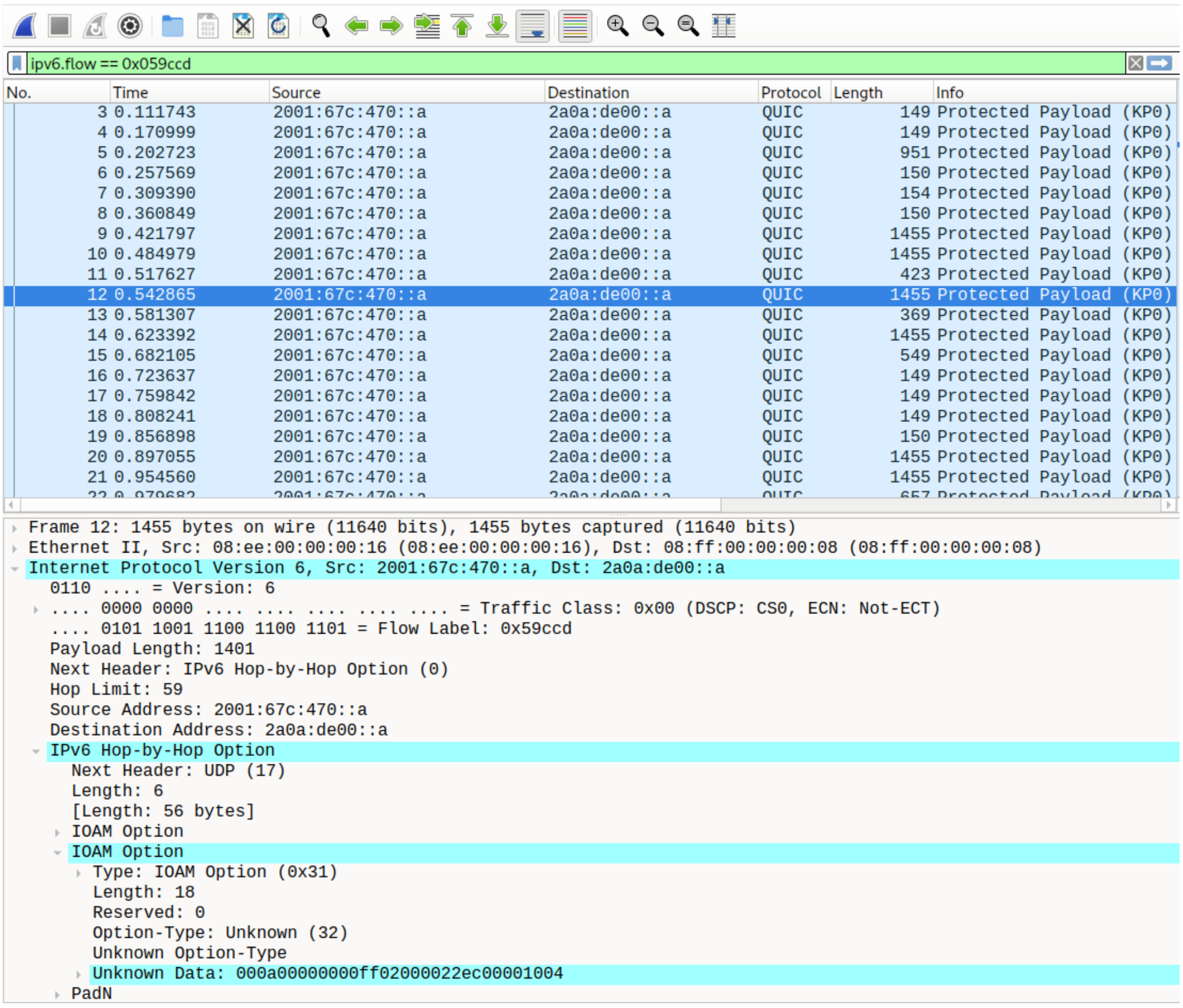


Ipv6.flow == 0x059ccd

```
No.         Time      Source              Destination      Protocol  Length  Info
         7 0.309390   2001:67c:470::a     2a0a:de00::a     QUIC       154 Protected Payload (KP0)
         8 0.360849   2001:67c:470::a     2a0a:de00::a     QUIC       150 Protected Payload (KP0)
         9 0.421797   2001:67c:470::a     2a0a:de00::a     QUIC      1455 Protected Payload (KP0)
        10 0.484979   2001:67c:470::a     2a0a:de00::a     QUIC      1455 Protected Payload (KP0)
        11 0.517627   2001:67c:470::a     2a0a:de00::a     QUIC       423 Protected Payload (KP0)
        12 0.542865   2001:67c:470::a     2a0a:de00::a     QUIC      1455 Protected Payload (KP0)
```

```
Frame 12: 1455 bytes on wire (11640 bits), 1455 bytes captured (11640 bits)
Ethernet II, Src: 08:ee:00:00:00:16 (08:ee:00:00:00:16), Dst: 08:ff:00:00:00:08 (08:ff:00:00:00:08)
Internet Protocol Version 6, Src: 2001:67c:470::a, Dst: 2a0a:de00::a
  0110 .... = Version: 6
  .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
  .... 0101 1001 1100 1100 1101 = Flow Label: 0x59ccd
  Payload Length: 1401
  Next Header: IPv6 Hop-by-Hop Option (0)
  Hop Limit: 59
  Source Address: 2001:67c:470::a
  Destination Address: 2a0a:de00::a
  [Stream index: 0]
  IPv6 Hop-by-Hop Option
    Next Header: UDP (17)
    Length: 6
    [Length: 56 bytes]
    IOAM Option
      Type: IOAM Option (0x31)
      Length: 26
      Reserved: 0
      Option-Type: Pre-allocated Trace (0)
      Pre-allocated Trace
    IOAM Option
      Type: IOAM Option (0x31)
      Length: 18
      Reserved: 0
      Option-Type: Aggregation (32)
      Aggregation
        Namespace ID: 10
        0000 .... = Flags: 0
          ...0 .... = Unsupported Aggregator: False
          ..0. .... = Unsupported Data Param: False
          .0.. .... = Unsupported Namespace: False
          0... .... = Other Error: False
        .... 0000  0000 0000 = Reserved: 0
        IOAM Data Param: 255
        0000 0010 = Aggregator: 2, MIN
          .... ...0 = SUM: False
          .... ..1. = MIN: True
          .... .0.. = MAX: False
          .... 0... = AVG: False
        Aggregate: 8940
        Auxil-data Node-ID: 16
        Hop Count: 4
    PadN
```

Figure 5.1.: Wireshark with IOAM Aggregation Option Dissector



```
No.   Time        Source             Destination     Protocol  Length  Info
   3 0.111743     2001:67c:470::a    2a0a:de00::a    QUIC      149 Protected Payload (KP0)
   4 0.170999     2001:67c:470::a    2a0a:de00::a    QUIC      149 Protected Payload (KP0)
   5 0.202723     2001:67c:470::a    2a0a:de00::a    QUIC      951 Protected Payload (KP0)
   6 0.257569     2001:67c:470::a    2a0a:de00::a    QUIC      150 Protected Payload (KP0)
   7 0.309390     2001:67c:470::a    2a0a:de00::a    QUIC      154 Protected Payload (KP0)
   8 0.360849     2001:67c:470::a    2a0a:de00::a    QUIC      150 Protected Payload (KP0)
   9 0.421797     2001:67c:470::a    2a0a:de00::a    QUIC     1455 Protected Payload (KP0)
  10 0.484979     2001:67c:470::a    2a0a:de00::a    QUIC     1455 Protected Payload (KP0)
  11 0.517627     2001:67c:470::a    2a0a:de00::a    QUIC      423 Protected Payload (KP0)
  12 0.542865     2001:67c:470::a    2a0a:de00::a    QUIC     1455 Protected Payload (KP0)
  13 0.581307     2001:67c:470::a    2a0a:de00::a    QUIC      369 Protected Payload (KP0)
  14 0.623392     2001:67c:470::a    2a0a:de00::a    QUIC     1455 Protected Payload (KP0)
  15 0.682105     2001:67c:470::a    2a0a:de00::a    QUIC      549 Protected Payload (KP0)
  16 0.723637     2001:67c:470::a    2a0a:de00::a    QUIC      149 Protected Payload (KP0)
  17 0.759842     2001:67c:470::a    2a0a:de00::a    QUIC      149 Protected Payload (KP0)
  18 0.808241     2001:67c:470::a    2a0a:de00::a    QUIC      149 Protected Payload (KP0)
  19 0.856898     2001:67c:470::a    2a0a:de00::a    QUIC      150 Protected Payload (KP0)
  20 0.897055     2001:67c:470::a    2a0a:de00::a    QUIC     1455 Protected Payload (KP0)
  21 0.954560     2001:67c:470::a    2a0a:de00::a    QUIC     1455 Protected Payload (KP0)
  22 0.979682     2001:67c:470::a    2a0a:de00::a    QUIC      657 Protected Payload (KP0)
```

```
> Frame 12: 1455 bytes on wire (11640 bits), 1455 bytes captured (11640 bits)
> Ethernet II, Src: 08:ee:00:00:00:16 (08:ee:00:00:00:16), Dst: 08:ff:00:00:00:08 (08:ff:00:00:00:08)
v Internet Protocol Version 6, Src: 2001:67c:470::a, Dst: 2a0a:de00::a
    0110 .... = Version: 6
  > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... 0101 1001 1100 1100 1101 = Flow Label: 0x59ccd
    Payload Length: 1401
    Next Header: IPv6 Hop-by-Hop Option (0)
    Hop Limit: 59
    Source Address: 2001:67c:470::a
    Destination Address: 2a0a:de00::a
  v IPv6 Hop-by-Hop Option
      Next Header: UDP (17)
      Length: 6
      [Length: 56 bytes]
    > IOAM Option
    v IOAM Option
      > Type: IOAM Option (0x31)
        Length: 18
        Reserved: 0
        Option-Type: Unknown (32)
        Unknown Option-Type
      > Unknown Data: 000a00000000ff02000022ec00001004
    > PadN
```

Figure 5.2.: Wireshark without IOAM Aggregation Option Dissector

5.1. Dissector as Plugin

Wireshark is designed with extensibility in mind. Plugins can be written in C and compiled to a shared object (so) on Linux or to a dynamically linked library (DLL) on Windows. Alternatively plugins can be written in Lua. The advantage of writing a plugin is that Wireshark itself does not need to be recompiled. Adding dissectors for custom proprietary protocols as plugins is a good choice because the custom extensions can be added to the program more easily. To profit from the lightweight extension possibilities using plugins we initially decided to follow the structure and implement the IOAM aggregation option dissector as a plugin. We managed to add a custom dissector for a simple example protocol which sets up on top of UDP, following the Wireshark documentation about the addition of a custom dissector. [1] Unfortunately the use case of adding a dissector for a protocol on top of UDP is different from the implementation of a dissector for the IOAM aggregation option because the header data is located in the Hop-by-Hop Options IPv6 extension header which is dissected by the built-in IPv6 dissector. We struggled at the point where we needed to register the dissector handle which is responsible to call our dissector for traffic which is associated with our protocol. The code snippet in listing 5.1 illustrates how the handler *foo_handle* is registered in the dissector table *udp.port* for traffic with the destination UDP port *FOO_PORT* which is a constant defined elsewhere.

Listing 5.1: Registering Dissector Handle for Example Protocol [1]

```c
void
proto_reg_handoff_foo(void)
{
    static dissector_handle_t foo_handle;

    foo_handle = create_dissector_handle(dissect_foo, proto_foo);
    dissector_add_uint("udp.port", FOO_PORT, foo_handle);
}
```

The registration of the IOAM aggregation option dissector would require a dissector table, comparable to *udp.port*, specifically for IOAM options. If that table would be available the IOAM aggregation option handler could be registered with the code snippet in listing 5.2

Listing 5.2: Registering Dissector Handle for IOAM Aggregation Option

```c
void
proto_reg_handoff_ioam_aggr(void)
{
    static dissector_handle_t ioam_aggr_handle;

    ioam_aggr_handle = create_dissector_handle(dissect_ioam_aggr, proto_ioam_aggr);
    dissector_add_uint("ioam.options", IOAM_AGGR_OPT, ioam_aggr_handle);
}
```

Unfortunately that didn't work and we couldn't figure out if there is an appropriate table somewhere. To avoid spending too much time on this we then decided to go for the implementation of a built-in dissector.

5.2. Built-in Dissector

To implement the built-in dissector for the IOAM aggregation option we followed the structure of the IOAM Pre-allocated Trace Option implementation which was added to Wireshark as a built-in dissector in November 2021 in merge request !4962. To do the implementation of the IOAM aggregation option dissector we decided to fork the Wireshark GitHub repository to a repository in our private namespace. Our changes to the Wireshark codebase are summarized in the pull request #1 in the mentioned GitHub repository.

In the following sections it is explained step by step how to extend the IPv6 dissector with an additional IOAM option type. The only file to update is *epan/dissectors/packet-ipv6.c*. Some important abbreviations to know when implementing a dissector for Wireshark are:

hf Header Field

ett Epan Tree Type

epan Enhanced Packet ANalyzer

tvb Testy, Virtual(-izable) Buffers

5.2.1. Add Option Type

The first set is to define a constant which defines the identification number of the IOAM option. The identification numbers are allocated by IANA. [8] As there is currently no allocation for the IOAM aggregation option we decided to use the number 32 which is currently unassigned.

Listing 5.3: Add IOAM Aggregation Option Type

```
#define IP6IOAM_AGGR                      32       /* Aggregation */

<-- lines omitted -->

static const value_string ipv6_ioam_opt_types[] = {
    { IP6IOAM_PRE_TRACE,  "Pre-allocated Trace" },
    { IP6IOAM_INC_TRACE,  "Incremental Trace"   },
    { IP6IOAM_POT,        "Proof of Transit"    },
    { IP6IOAM_E2E,        "Edge to Edge"        },
    { IP6IOAM_AGGR,       "Aggregation"         },
    { 0, NULL}
};
```

5.2.2. Add Header Fields

Next indices for all header fields are added as static global variables. The header fields defined conform to the definition in draft-cxx-ippm-ioamaggr-00. [3]

Listing 5.4: Add IOAM Aggregation Header Fields

```
static int hf_ipv6_opt_ioam_aggr_ns;
static int hf_ipv6_opt_ioam_aggr_flags;
static int hf_ipv6_opt_ioam_aggr_flag_1;
static int hf_ipv6_opt_ioam_aggr_flag_2;
static int hf_ipv6_opt_ioam_aggr_flag_3;
static int hf_ipv6_opt_ioam_aggr_flag_4;
static int hf_ipv6_opt_ioam_aggr_reserved;
static int hf_ipv6_opt_ioam_aggr_data_param;
static int hf_ipv6_opt_ioam_aggr_aggregator;
static int hf_ipv6_opt_ioam_aggr_aggregator_sum;
static int hf_ipv6_opt_ioam_aggr_aggregator_min;
static int hf_ipv6_opt_ioam_aggr_aggregator_max;
static int hf_ipv6_opt_ioam_aggr_aggregator_avg;
static int hf_ipv6_opt_ioam_aggr_aggregate;
static int hf_ipv6_opt_ioam_aggr_node_id;
static int hf_ipv6_opt_ioam_aggr_hop_count;
```

5.2.3. Add Subtree Fields

The last preparation step is the addition of the indices for the subtree. As visible in 5.1 both the *Flags* and the *Aggregator* have a subtree which can be expanded on demand. Indices to those two subtrees need to be defined beforehand similar to the header field indices definition.

Listing 5.5: Add IOAM Aggregation Subtree Fields

```
static gint ett_ipv6_opt_ioam_aggr_flags;
static gint ett_ipv6_opt_ioam_aggr_aggregators;
```

5.2.4. Add Sub-Dissector

The sub-dissector for the IOAM aggregation option is implemented in the function with the signature depicted in listing 5.6. Before going into any details about this specific dissector it is important to understand the concept behind dissecting and to know the role of the *tvbuff_t* data structure.

In the Wireshark documentation the importance of tvb data structure is described as follows: When dissecting a frame: The top-level dissector (packet.c) pushes the initial tvb (containing the complete frame) onto the stack (starts the chain) and then calls a sub-dissector which in turn calls the next sub-dissector and so on. Each sub-dissector may chain additional tvbs to the tvb handed to that dissector. [17] In other words the tvb contains the raw data of the captured frame and dissectors can read data from that frame given the current offset (index) and interpret the data accordingly.

5.2.4.1. Function Signature

The function *dissect_opt_ioam_aggr* takes the following arguments.

tvbuff_t *tvb Pointer to the buffer which contains the packet data. For more information refer to tvbuff_t in the Wireshark documentation.

gint offset Points to the data inside the tvb which has to be interpreted (dissected) next. Each time a field is dissected the offset gets increased by the amount of bytes read. At the end the updated offset is returned to the caller.

packet_info *pinfo Pointer to the packet metadata such as timestamps, the name of the protocol which is currently dissected and much more. For more information refer to packet_info in the Wireshark documentation.

proto_tree *opt_tree Pointer to the protocol tree element. Referring again to figure 5.1 this would be the tree called *Aggregation*.

struct opt_proto_item *opt_ti Pointer to an IPv6 dissector specific protocol item. It stores the type and length of the Hop-by-Hop Option extension header.

guint8 opt_len Stores the length of an option inside the Hop-by-Hop Option extension header.

The return value is an integer representing the offset in the tvb data structure after the dissection process.

Listing 5.6: Add IOAM Aggregation Dissector Logic

```c
static gint
dissect_opt_ioam_aggr(
  tvbuff_t *tvb,
  gint offset,
  packet_info *pinfo,
  proto_tree *opt_tree,
  struct opt_proto_item *opt_ti,
  guint8 opt_len
);
```

5.2.4.2. Add Item to Tree

Adding an item to a protocol tree is straight forward. Dissectors use proto_tree_add_* to add items to the protocol tree. In most cases proto_tree_add_item() is used.

The code snippet in listing 5.7 adds the IOAM aggregation option *Namespace-ID* header field. According to draft-cxx-ippm-ioamaggr-00 is a 16-bit identifier of an IOAM-Namespace. [3] The function is called providing the protocol tree, the target header field, the tvb buffer containing the frame data, the current offset in tvb, the length of the field in bytes and the encoding. Add the end the offset is increased by the number of bytes read from the tvb buffer.

Listing 5.7: Add IOAM Aggregation Add Tree Item

```
// Namespace
proto_tree_add_item(opt_tree, hf_ipv6_opt_ioam_aggr_ns, tvb, offset, 2,
    ENC_BIG_ENDIAN);
offset += 2;
```

This step is repeated for each generic header field. In case the field is not aligned to one octet, the function *proto_tree_add_bits_item()* is used instead. When adding a bits item one must take extra care to increment the offset correctly. Most likely multiple bit items are added one after an other and as soon as the offset is again aligned to an octet it should be increased accordingly.

5.2.4.3. Add Bitmask to Tree

To add a field which contains multiple flags and which should be expandable to a subtree a bitmask item should be added instead. In the case of the IOAM aggregation option this was done for the two fields flags and aggregator.

To add the aggregator field as a bitmask for example the first thing to do is to create a local null terminated array as the subtree index. It contains pointers the header fields in the order of occurrence.

Listing 5.8: Add IOAM Aggregation Flags Index

```
static int * const ioam_aggr_aggregators[] = {
    &hf_ipv6_opt_ioam_aggr_aggregator_sum,
    &hf_ipv6_opt_ioam_aggr_aggregator_min,
    &hf_ipv6_opt_ioam_aggr_aggregator_max,
    &hf_ipv6_opt_ioam_aggr_aggregator_avg,
    NULL
};
```

Next the *proto_tree_add_bits_item()* is called providing the protocol tree, the tvb buffer containing the frame data, the current offset in tvb, the target header field, the target field in the epan tree, the array initialized before and the encoding.

Listing 5.9: Add IOAM Aggregation Flags Index

```
// Aggregator
proto_tree_add_bitmask(
  opt_tree,
  tvb,
  offset,
  hf_ipv6_opt_ioam_aggr_aggregator,
  ett_ipv6_opt_ioam_aggr_aggregators,
  ioam_aggr_aggregators,
  ENC_NA);
offset += 1;
```

5.2.5. Register Dissector

The function for the sub-dissector is now implemented and needs to be called at a certain condition. As already mentioned the IOAM aggregation option dissector shall be called in case the IOAM option is equal to 32.

The registration of the sub-dissector function is straight forward and illustrated in listing 5.10. It is only the definition of a the new case *IP6IOAM_AGGR*, which is a constant an equal to 32, in the function *dissect_opt_ioam*.

Listing 5.10: Add IOAM Aggregation Register Dissector

```c
    switch (opt_type) {
    case IP6IOAM_PRE_TRACE:
    case IP6IOAM_INC_TRACE:
        offset = dissect_opt_ioam_trace(tvb, offset, pinfo, opt_type_tree, opt_ti,
    opt_len);
        break;
    case IP6IOAM_POT:
        break;
    case IP6IOAM_E2E:
        break;
    case IP6IOAM_AGGR:
        offset = dissect_opt_ioam_aggr(tvb, offset, pinfo, opt_type_tree, opt_ti,
    opt_len);
        break;
    }
```

5.2.6. Register Header Fields

As part of the IPv6 protocol registration, all header fields and the corresponding information needs to be declared. The header field information is of particular importance to the user interface. For example the user readable name and the filter name, the base, bitmasks and more is defined.

The registration of the *Namespace ID* header field is shown in listing 5.11.

Listing 5.11: Add IOAM Aggregation Register Field Namespace ID

```c
{ &hf_ipv6_opt_ioam_aggr_ns,
    { "Namespace ID", "ipv6.opt.ioam.aggr.ns",
        FT_UINT16, BASE_DEC, NULL, 0x0,
        NULL, HFILL }
},
```

The registration of the *Aggregator* header field is shown in listing 5.12.

Listing 5.12: Add IOAM Aggregation Register Field Aggregator

```c
{ &hf_ipv6_opt_ioam_aggr_aggregator,
    { "Aggregator", "ipv6.opt.ioam.aggr.aggregator",
        FT_UINT8, BASE_DEC, NULL, 0xFF,
        NULL, HFILL }
},
{ &hf_ipv6_opt_ioam_aggr_aggregator_sum,
    { "SUM", "ipv6.opt.ioam.aggr.aggregator.sum",
        FT_BOOLEAN, 8, NULL, 0x1,
        NULL, HFILL }
},
{ &hf_ipv6_opt_ioam_aggr_aggregator_min,
    { "MIN", "ipv6.opt.ioam.aggr.aggregator.min",
the \emph{proto{\_}tree{\_}add{\_}bits{\_}item()} is called providing the protocol
    tree, the tvb buffer containing the frame data, the current offset in tvb, the
    target header field, the target field in the epan tree, the array initialized
    before and the encoding.

        FT_BOOLEAN, 8, NULL, 0x2,
        NULL, HFILL }
},
{ &hf_ipv6_opt_ioam_aggr_aggregator_max,
    { "MAX", "ipv6.opt.ioam.aggr.aggregator.max",
        FT_BOOLEAN, 8, NULL, 0x4,
        NULL, HFILL }
},
{ &hf_ipv6_opt_ioam_aggr_aggregator_avg,
    { "AVG", "ipv6.opt.ioam.aggr.aggregator.avg",
        FT_BOOLEAN, 8, NULL, 0x8,
        NULL, HFILL }
},
```

Each header field element is associated to a header_field_info struct. The struct fields are described below in the order of occurrence.

name Is the full name of this field.

abbrev Is the filter name of this field.

type Is the field type, one of FT_ (from ftypes.h).

display Defines how to display a field. For example whether the field value shall be displayed as hexadecimal or decimal number. One of BASE_, or field bit-width if FT_BOOLEAN and non-zero bitmask.

strings In case the type of the field is is an *FT_PROTOCOL* or *BASE_PROTOCOL_INFO* then it points to the associated protocol_t structure

bitmask Marks bits of interest.

blurp Brief description of field.

HFILL Initializes all the *set by proto routines* fields in header field info.

5.2.7. Register Subtree Fields

Finally the two subtree fields used to display the flag's and the aggregator's bitmasks need to be registered. This is achieved by the addition of the lines 15 and 16 in listing 5.13.

Listing 5.13: Add IOAM Aggregation Register Subtree Fields

```
1  static gint *ett_ipv6[] = {
2      &ett_ipv6_proto,
3      &ett_ipv6_detail,
4      &ett_ipv6_detail_special_purpose,
5      &ett_ipv6_multicast_flags,
6      &ett_ipv6_traffic_class,
7      &ett_geoip_info,
8      &ett_ipv6_opt,
9      &ett_ipv6_opt_type,
10     &ett_ipv6_opt_rpl,
11     &ett_ipv6_opt_mpl,
12     &ett_ipv6_opt_dff_flags,
13     &ett_ipv6_opt_ioam_trace_flags,
14     &ett_ipv6_opt_ioam_trace_types,
15     &ett_ipv6_opt_ioam_aggr_flags,
16     &ett_ipv6_opt_ioam_aggr_aggregators,
17     &ett_ipv6_fragment,
18     &ett_ipv6_fragments
19 };
```

Part IV.

Transition

1. Demo

This demonstration provides an overview of the system's startup and configuration processes. It begins with an explanation of how to start up all system components. Once all systems are up and running, three scenarios are presented in which an efficiency or network parameter is changed. Each scenario includes a description of how the parameter update can be performed and a comparison of a dashboard state before and after the update is provided to illustrate the impact of the changes.

In the demonstration the network depicted in figure 1.1 is used as topology in the network virtualization system. The topology is further described in section 3.1 in the elaboration part.

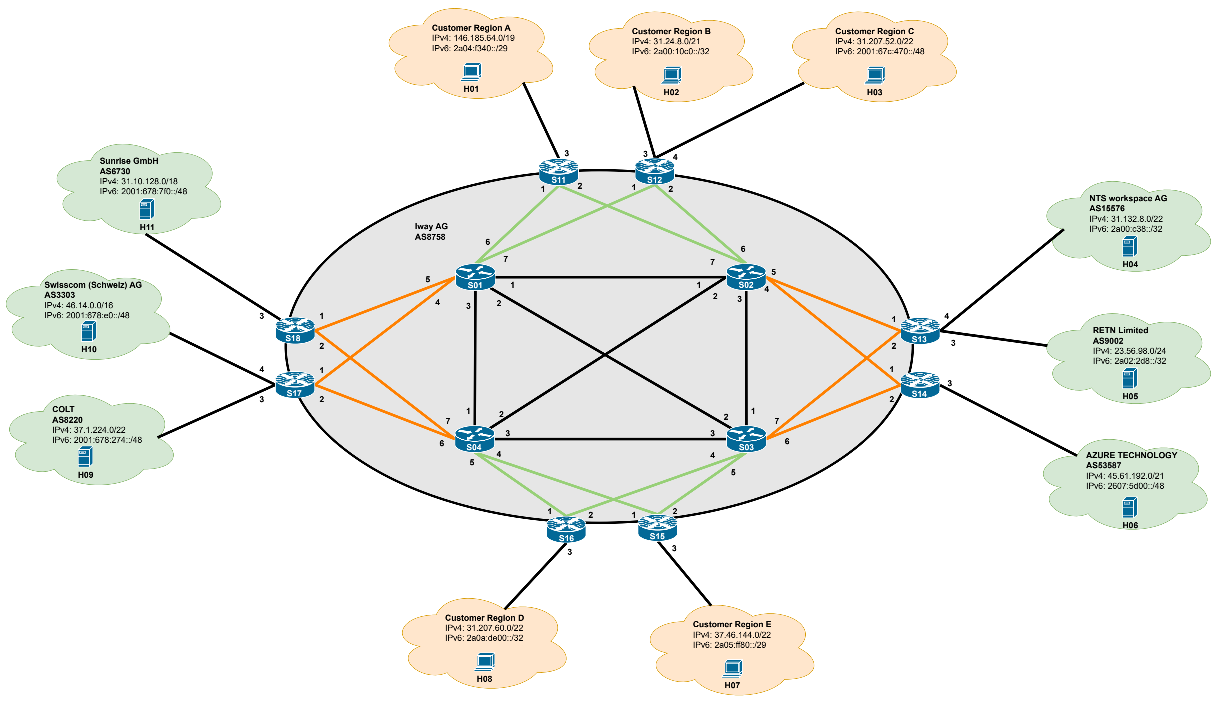


Figure 1.1.: Network Simulation Topology

1.1. Getting Started

In the following sections, it is explained how each service is started and used. After the network virtualization system (Mininet) is started, the traffic generator will automatically start sending packets without further interaction. For ease of use, dedicate a separate terminal instance to each service.

1.1.1. Network Virtualization System

To use the network virtualization system (Mininet) clone the repository *Efficiency Indicator P4* and change the working directory into the root of the git repository. With the following command the Mininet environment is started:

Listing 1.1: Startup - Network Virtualization System

```
make run
```

1.1.2. Monitoring System

To use the monitoring system clone the repository *Efficiency Indicator Monitoring* and change the working directory into the desired deployment folder, local or server.

local The local directory is used to deploy the monitoring system on the same computer as the network virtualization (Mininet) is running.

server The server directory is used to deploy the monitoring system on a dedicated server.

There is a Docker Compose file in each folder, and the following command is used to start the monitoring system deployment:

Listing 1.2: Startup - Monitoring System

```
docker compose up
```

After the startup command was executed the following three services are up and running:

Grafana http://<IP-ADDRESS>:3000

InfluxDB http://<IP-ADDRESS>:8086

Telegraf udp://<IP-ADDRESS>:4739

Information

The IP addresses of the three services are specified in the *telegraf.conf* file in the *Efficiency Indicator Monitoring* repository.

1.1.3. Configuration Update System

To use the configuration update system clone the repository and change the working directory into the root of the git repository.

1.1.3.1. Installation

The first step is to setup the project by the creation of a virtual environment and the installation of the required dependencies. To do so follow the setup instructions in listing 1.3

Listing 1.3: Setup Instructions

```
# Create a new virtual environment in the hidden folder .venv
python3 -m venv .venv

# Activate the virtual environment
source .venv/bin/activate

# Install dependencies
pip install -r requirements.txt
```

1.1.3.2. Configuration

Check if the correct *mininet_host* is specified in the configuration update system repository *efficiency-indicator-configuration-update/inventory/defaults.yaml*, it must be the ip address or the hostname of the system where the network virtualization system is running.

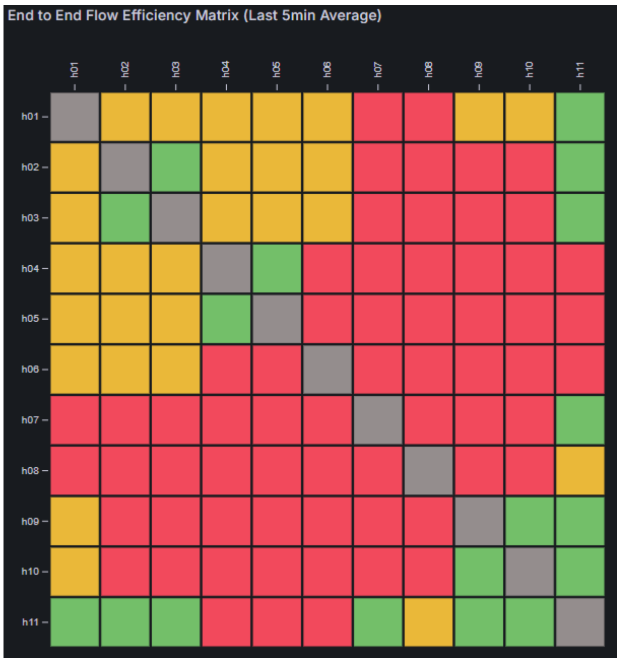
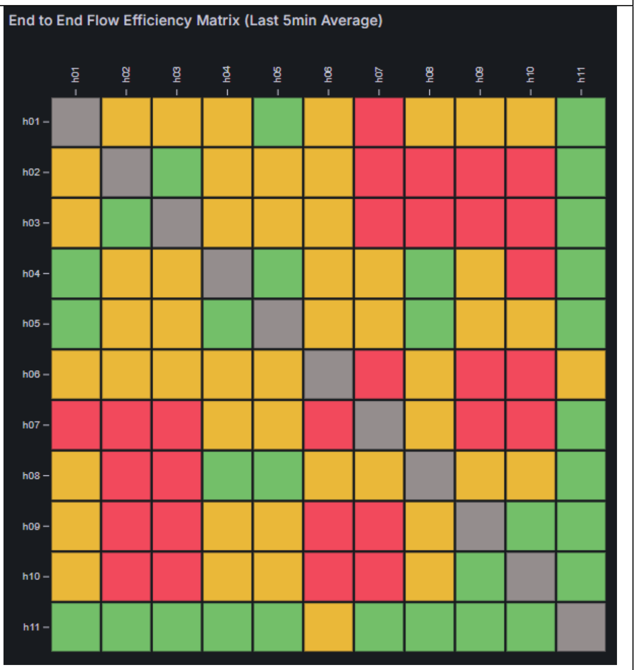
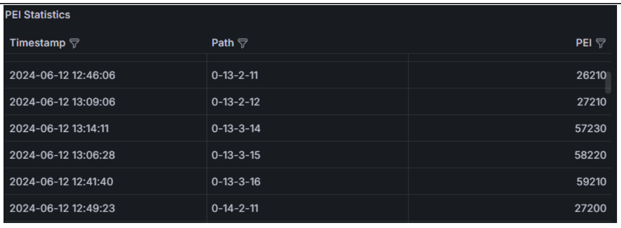
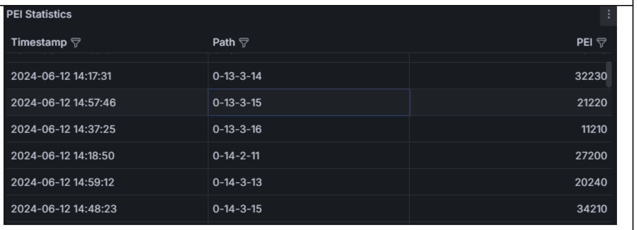
Listing 1.4: Setup Instructions

```
data: {
  p4_repo_path: "/home/boss/git/ba/efficiency-indicator-p4/",
  runtime_path: "dev-network/", # relative to p4_repo_path
  mininet_host: <IP-ADDRESS>,
  checksum_file: "tmp/checksums.json"
}
```

1.2. Update Scenarios

1.2.1. Change HEI on Switches s03, s13, s16

In the first scenario, we will change the HEI value of three switches in the simulation network topology to improve the flows that traverse the switches *s03*, *s13*, and *s16*. As can be seen in the Flow Efficiency Matrix (part of the Flow Statistics dashboard) in table 1.1, the flow efficiency e.g. from and to the hosts *h04* and *h05* are red, that means they are not that good. What do we expect, if we decrease the HEI on switch *s03* from 30000 to 5000, on *s13* from 13000 to 1000 and on *s16* from 16000 to 5000? We expect a significant improvement in flow efficiency e.g. from and to the hosts *h04* and *h05*. We also expect to see an improvement in the PEI statistics (part of the Path Statistics dashboard) for those paths that traverse via the switch *s03*.

Before HEI Update	After HEI Update
	
	

PEI Statistics (Before HEI Update)

Timestamp	Path	PEI
2024-06-12 12:46:06	0-13-2-11	26210
2024-06-12 13:09:06	0-13-2-12	27210
2024-06-12 13:14:11	0-13-3-14	57230
2024-06-12 13:06:28	0-13-3-15	58220
2024-06-12 12:41:40	0-13-3-16	59210
2024-06-12 12:49:23	0-14-2-11	27200

PEI Statistics (After HEI Update)

Timestamp	Path	PEI
2024-06-12 14:17:31	0-13-3-14	32230
2024-06-12 14:57:46	0-13-3-15	21220
2024-06-12 14:37:25	0-13-3-16	11210
2024-06-12 14:18:50	0-14-2-11	27200
2024-06-12 14:59:12	0-14-3-13	20240
2024-06-12 14:48:23	0-14-3-15	34210

Table 1.1.: HEI Update Comparison

The network flow efficiency statistic in figure 1.2 shows the average flow efficiency over the last 5 minutes. In the graph, you can see that the average flow efficiency (FEI) decreases after the changes of reducing the HEI on the switches.

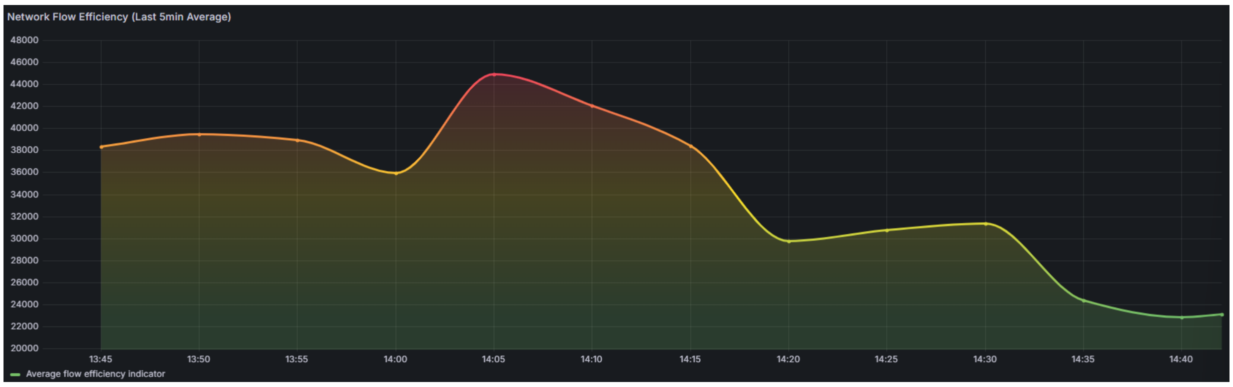
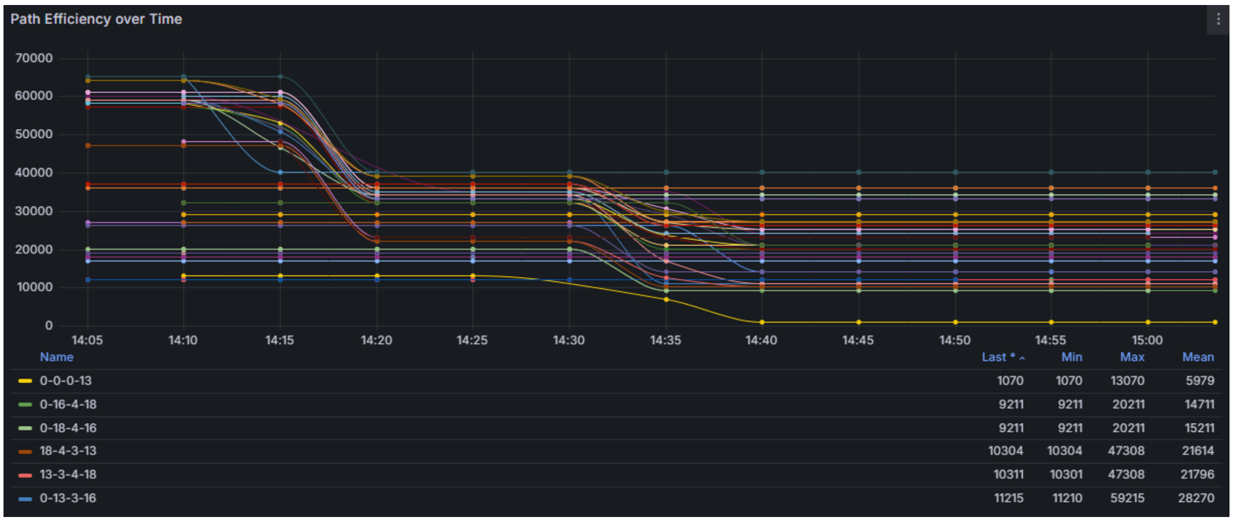


Figure 1.2.: Network Flow Efficiency (Last 5min Average)

As can be seen in figure 1.3 all paths of the simulation network topology are visualized in the time series statistic with its corresponding path efficiency (PEI). It is clearly visible that after reducing the HEI values on switches *s03*, *s13* and *s16*, the paths that traverse the previously mentioned switches shows a significantly better PEI (drop in the time series).



Name	Last *	Min	Max	Mean
🟨 0-0-0-13	1070	1070	13070	5979
🟩 0-16-4-18	9211	9211	20211	14711
🟩 0-18-4-16	9211	9211	20211	15211
🟧 18-4-3-13	10304	10304	47308	21614
🟥 13-3-4-18	10311	10301	47308	21796
🟦 0-13-3-16	11215	11210	59215	28270

Figure 1.3.: Path Efficiency over Time

To trigger the changes of the network efficiency in this demo case, the following steps were taken.

1. Change on switch *s03* the HEI value from 30000 to 5000 in the *resource.yaml* file

Listing 1.5: Change HEI value

```
s03:
  mac: 08:CC:00:00:00:03
  hei:
    - data_param: 255
      value: 5000
```

2. Change on switch *s13* the HEI value from 13000 to 1000 in the *resource.yaml* file

Listing 1.6: Change HEI value

```
s13:
  mac: 08:EE:00:00:00:13
  hei:
    - data_param: 255
      value: 1000
```

3. Change on switch *s16* the HEI value from 16000 to 5000 in the *resource.yaml* file

Listing 1.7: Change HEI value

```
s16:
  mac: 08:EE:00:00:00:16
  hei:
    - data_param: 255
      value: 5000
```

4. Open a terminal, change the working directory into the root of the *Efficiency Indicator P4* repository and enter the following command to generate the new config files

Listing 1.8: Generate New Configuration Files

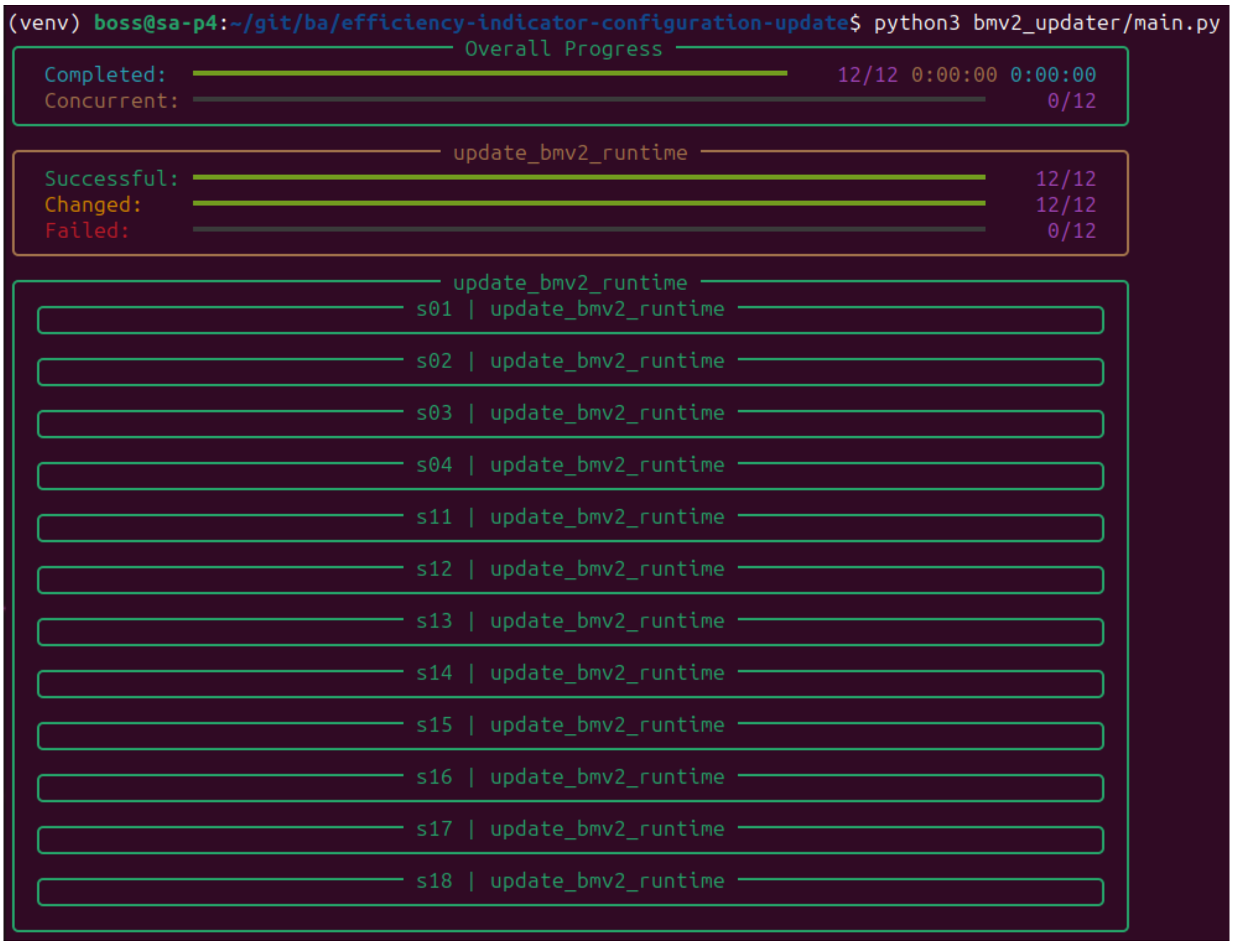
```
make config
```

5. Open a terminal, change the working directory into the root of the *Efficiency Indicator Configuration Update* repository and push the new config files to the BMv2 software switches

Listing 1.9: Config Update

```
python3 bmv2_updater/main.py
```

As can be seen in figure 1.4, the configuration is successfully updated on all switches. The configuration update system is further described in chapter 3 in the construction part.



```
(venv) boss@sa-p4:~/git/ba/efficiency-indicator-configuration-update$ python3 bmv2_updater/main.py
 Overall Progress
 Completed:                                    12/12 0:00:00 0:00:00
 Concurrent:                                                    0/12
 update_bmv2_runtime
 Successful:                                                   12/12
 Changed:                                                      12/12
 Failed:                                                        0/12
 update_bmv2_runtime
 s01 | update_bmv2_runtime
 s02 | update_bmv2_runtime
 s03 | update_bmv2_runtime
 s04 | update_bmv2_runtime
 s11 | update_bmv2_runtime
 s12 | update_bmv2_runtime
 s13 | update_bmv2_runtime
 s14 | update_bmv2_runtime
 s15 | update_bmv2_runtime
 s16 | update_bmv2_runtime
 s17 | update_bmv2_runtime
 s18 | update_bmv2_runtime
```

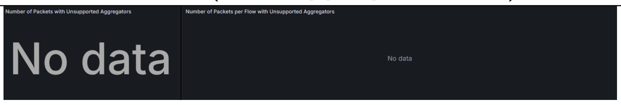
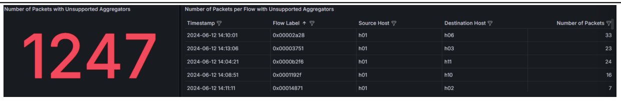
Figure 1.4.: Config Updater - Console Output

1.2.2. Add Invalid Aggregator on Switch s11

In this scenario, we will change an aggregator type of switch *s11* in the simulation network topology. Before we update the configuration we can see in the *Unsupported Aggregator* statistic in table 1.2, that no unsupported aggregator error occurred. What do we expect, if we change an aggregator on *s11* from type 1 (SUM) to type 3 (Unsupported Aggregator)? Certainly we expect, that errors will appear in the *Unsupported Aggregator* statistic.

Information

IOAM Aggregation Option error statistics are in the *Simulation Network Statistics* dashboard.

Before (s11, aggregator = 1)	**After (s11, aggregator = 3)**
	

Before (s11, aggregator = 1):

Number of Packets with Unsupported Aggregators

No data

Number of Packets per Flow with Unsupported Aggregators

No data

After (s11, aggregator = 3):

Number of Packets with Unsupported Aggregators

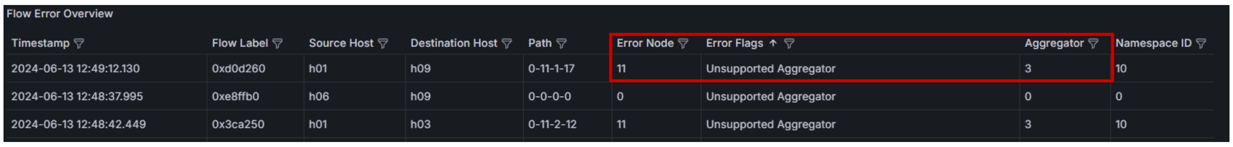
1247

Number of Packets per Flow with Unsupported Aggregators

Timestamp	Flow Label ↑	Source Host	Destination Host	Number of Packets
2024-06-12 14:10:01	0x00002a28	h01	h06	33
2024-06-12 14:13:06	0x00003751	h01	h03	23
2024-06-12 14:04:21	0x0000b2f6	h01	h11	24
2024-06-12 14:08:51	0x0001192f	h01	h10	16
2024-06-12 14:11:11	0x00014871	h01	h02	7

Table 1.2.: Aggregator Error Comparison

The Flow Error Overview statistic shown in figure 1.5 is based on the IPFIX raw data export, which is further described in section 3.5 in the elaboration part. The raw data export not only exports the path information of a specific flow, but also the error information, which means you can associate the error with a specific flow or even a specific node, making debugging easier and faster. As can be seen in figure 1.5, the unsupported aggregator set on switch *s11* is also detected by the raw data export, and it shows that an error of type *Unsupported Aggregator* has occurred at the *s11* error node.



Flow Error Overview

Timestamp	Flow Label	Source Host	Destination Host	Path	Error Node	Error Flags ↑	Aggregator	Namespace ID
2024-06-13 12:49:12.130	0xd0d260	h01	h09	0-11-1-17	11	Unsupported Aggregator	3	10
2024-06-13 12:48:37.995	0xe8ffb0	h06	h09	0-0-0-0	0	Unsupported Aggregator	0	0
2024-06-13 12:48:42.449	0x3ca250	h01	h03	0-11-2-12	11	Unsupported Aggregator	3	10

Figure 1.5.: Flow Error Overview

To trigger the occurrence of unsupported aggregator errors in this demo case, the following steps were taken.

1. Change on switch s11 the aggregator from 1 to 3 in the *resource.yaml* file

   Listing 1.10: Set Invalid Aggregator

```
s11:
  mac: 08:EE:00:00:00:11
  hei:
    - data_param: 255
      value: 11000
  ioam:
    namespace_id: 10
    node_id: 11
    aggregators: # 1 = SUM / 2 = MIN / 4 = MAX
      - 3 # selected if last two bits of payload size are [00]
      - 2 # selected if last two bits of payload size are [01]
      - 1 # selected if last two bits of payload size are [10]
      - 4 # selected if last two bits of payload size are [11]
```

2. Open a terminal, change the working directory into the root of the *Efficiency Indicator P4* repository.

   Listing 1.11: Generate New Configuration Files

```
make config
```

3. Open a terminal, change the working directory into the root of the *Efficiency Indicator Configuration Update* repository and push the new config files to the BMv2 software switches

   Listing 1.12: Config Update

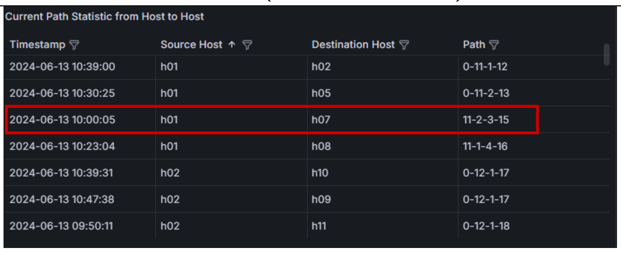
```
python3 bmv2_updater/main.py
```

If no errors occurred while updating the configuration, the console output from the configuration update system should look as shown in figure 1.4.

1.2.3. Change Path from h01 to h07

In this scenario, we will change the path between host *h01* to host *h07* in the simulation network topology. The path update is visible in table 1.3. The paths will be adjusted that the network traffic will traverse via the core switch *s04* instead of *s03*. What do we expect when we change the paths between these hosts? We expect the flow efficiency (FEI) between these hosts and the path efficiency (PEI) will be improved because the HEI value of *s04* is 26000 less than that of *s03*.

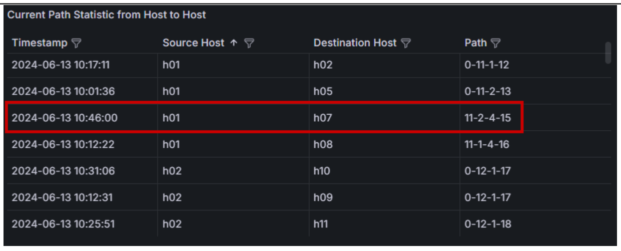
Before (Path via s03)



Current Path Statistic from Host to Host

Timestamp	Source Host ↑	Destination Host	Path
2024-06-13 10:39:00	h01	h02	0-11-1-12
2024-06-13 10:30:25	h01	h05	0-11-2-13
2024-06-13 10:00:05	h01	h07	11-2-3-15
2024-06-13 10:23:04	h01	h08	11-1-4-16
2024-06-13 10:39:31	h02	h10	0-12-1-17
2024-06-13 10:47:38	h02	h09	0-12-1-17
2024-06-13 09:50:11	h02	h11	0-12-1-18

After (Path via s04)

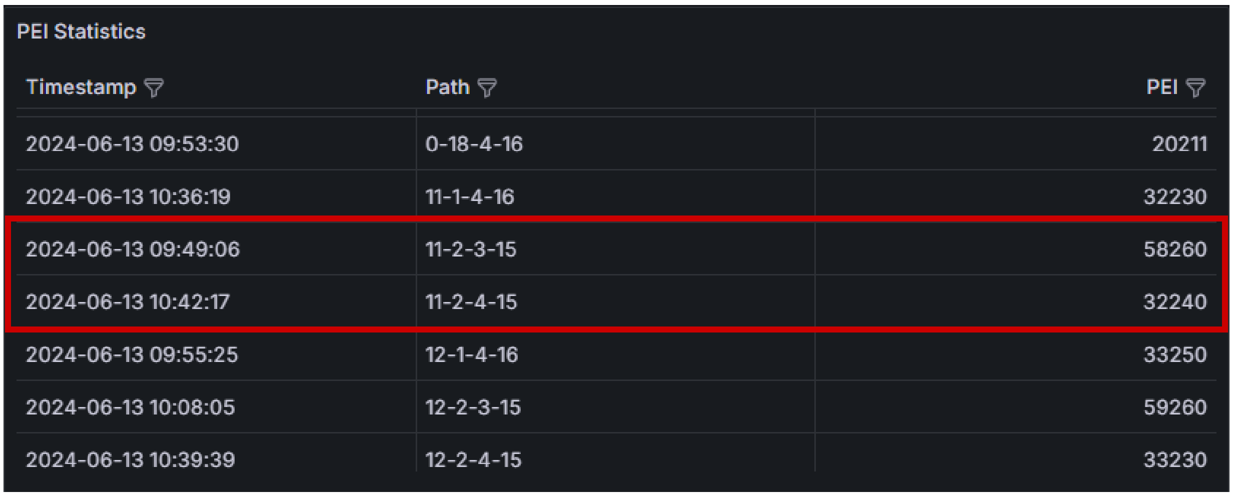


Current Path Statistic from Host to Host

Timestamp	Source Host ↑	Destination Host	Path
2024-06-13 10:17:11	h01	h02	0-11-1-12
2024-06-13 10:01:36	h01	h05	0-11-2-13
2024-06-13 10:46:00	h01	h07	11-2-4-15
2024-06-13 10:12:22	h01	h08	11-1-4-16
2024-06-13 10:31:06	h02	h10	0-12-1-17
2024-06-13 10:12:31	h02	h09	0-12-1-17
2024-06-13 10:25:51	h02	h11	0-12-1-18

Table 1.3.: Path Statistic Comparison

As can be seen in figure 1.6 the PEI of path *s11 - s02 - s03 - s15* is much higher than the PEI of path *s11 - s02 - s04 - s15*. So this optimization was good, we could almost halve PEI on the path from host *h01* to host *h07*.



PEI Statistics

Timestamp	Path	PEI
2024-06-13 09:53:30	0-18-4-16	20211
2024-06-13 10:36:19	11-1-4-16	32230
2024-06-13 09:49:06	11-2-3-15	58260
2024-06-13 10:42:17	11-2-4-15	32240
2024-06-13 09:55:25	12-1-4-16	33250
2024-06-13 10:08:05	12-2-3-15	59260
2024-06-13 10:39:39	12-2-4-15	33230

Figure 1.6.: PEI Statistics

The following steps were performed to get the similar result from above.

1. Change the path from *h01* to *h07* to go via *s11 - s02 - **s04**, s15* instead of *s11 - s02 - s03 - s15* in the *resource.yaml* file

Listing 1.13: Change Path

```
paths:
```

```yaml
# route from h01
- from: h01
  to: h07
  via: [s11, s02, s04, s15]
  return_route: true
```

2. Open a terminal, change the working directory into the root of the *Efficiency Indicator P4* repository.

Listing 1.14: Generate New Configuration Files

```
make config
```

3. Open a terminal, change the working directory into the root of the *Efficiency Indicator Configuration Update* repository and push the new config files to the BMv2 software switches

Listing 1.15: Config Update

```
python3 bmv2_updater/main.py
```

If no errors occurred while updating the configuration, the console output from the configuration update system should look as shown in figure 1.4.

1.2.4. Route Update to Avoid Switch s03

In the current topology of the simulation network, the core switch *s03* is the most inefficient switch with an HEI value of 30000. Now we want to improve the efficiency of the entire simulation network. How can we do this? We will change all paths that traverse the *s03* switch to alternative paths. To do this, we will change all affected paths in the *resource.yaml* file.

As can be seen in figure 1.7, the overall network path efficiency has improved significantly since path optimization was pushed to the BMv2 switches, after all paths are changed to alternative paths that were previously traversed via switch *s03*.

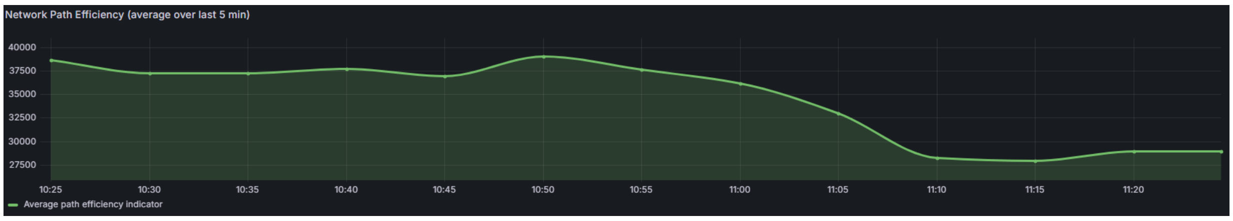


Figure 1.7.: Network Path Efficiency

The following steps were performed to get the similar result from above.

1. Change all affected paths to alternative paths in the *resource.yaml* file, see figure 1.1 to find alternative paths.

2. Open a terminal, change the working directory into the root of the *Efficiency Indicator P4* repository.

Listing 1.16: Generate New Configuration Files

```
make config
```

3. Open a terminal, change the working directory into the root of the *Efficiency Indicator Configuration Update* repository and push the new config files to the BMv2 software switches

   Listing 1.17: Config Update

   ```
   python3 bmv2_updater/main.py
   ```

   If no errors occurred while updating the configuration, the console output from the configuration update system should look as shown in figure 1.4.

2. Conclusion and Discussion

2.1. Main Purpose and Context

The primary aim of this study was to implement a fully functional use case for retrieving network efficiency data from a computer network within a simulated environment. This was executed as a proof of concept (PoC) to build on the green networking metrics developed in our preceding term paper. The study focused on implementing an IPFIX (IP Flow Information Export) exporter within the control plane of BMv2 software switches and modifying the data plane to accommodate these changes. A key aspect was the setup of a monitoring system designed to collect and visualize network telemetry data. This initiative was driven by the need to address sustainability concerns within computer networking by establishing a method to make energy efficiency data visible, thereby laying the groundwork for future reductions in the carbon footprint of global networking infrastructure.

2.2. Review of Findings

The study yielded several significant findings:

IPFIX Export and Control Plane Impact The IPFIX export process for network telemetry data, collected via the IOAM (In-situ Operations, Administration, and Maintenance) Aggregation Option, demonstrated an acceptable level of overhead in the control plane. Importantly, this overhead did not affect the data plane, ensuring that data forwarding performance remained unimpacted. This is a notable advantage as it means that the efficiency data can be gathered without compromising the speed and reliability of data traffic through the network.

Improved Efficiency Indicator Processing The approach chosen in this bachelor thesis improved the performance of efficiency indicator processing by shifting the responsibility for indicator calculations from the data plane to the control plane. Unlike the method proposed in the term paper, where the data plane recalculated the indicator value for every data packet, this study's approach involves periodic recomputation of indicator values by the control plane. This reduces the computational burden on the data plane and enhances overall processing efficiency, ensuring that network performance is not compromised while maintaining up-to-date efficiency metrics.

Aggregator Selection for Path Efficiency The study implemented a mechanism for random selection of aggregators at the ingress node. This approach provided a comprehensive view of network path efficiency as well as the identification of both the most and least efficient hops on paths. This method enhances the network's ability to pinpoint inefficiencies and optimize routing.

Multiple Efficiency Indicators Collection Unlike the approach suggested in the term paper, which aggregated multiple independent efficiency values into a single indicator, this study allowed for the collection of different efficiency indicators for different packets simultaneously. This prevents the loss of valuable information, as each efficiency metric can now be measured independently, offering a more granular and accurate view of network performance.

Visualization and Routing Insights The developed visualizations offered profound insights into network efficiency. The gathered information could be integrated into routing algorithms as custom weights, promoting the routing of traffic through the most efficient paths. This capability is crucial for optimizing network performance and energy usage, and ultimately contributes to more sustainable network operations.

2.3. Implications of the Study

The implications of this study are far-reaching, particularly for the optimization of network energy efficiency. The ability to dynamically incorporate efficiency indicators into data forwarding processes allows for the adaptation of network operations to reduce energy consumption. For example, network traffic can be routed through the most energy-efficient paths, while less efficient routes can remain deactivated until required, minimizing unnecessary energy use. This approach not only enhances the sustainability of network operations but also provides a foundation for further advancements in energy-efficient networking practices. The study's results underscore the potential for increased visibility into the energy efficiency of network paths, representing a pivotal step towards more sustainable networking infrastructure.

2.4. Limitations of the Study

Several limitations were identified in this study:

Simulated Environment Constraints The findings are currently applicable only within a simulated environment. The transition from a simulated setup to real-world application poses challenges that need to be addressed in future research.

Lack of Standardization The protocol extension used, the IOAM Aggregation Option, is not yet standardized and therefore is not available on current network devices. This limits the immediate applicability of the study's findings in practical network environments.

Data Availability and Device Constraints The efficiency information available on local devices is currently limited. There is a need for this data to be accessible through control plane table lookups during data forwarding, a capability not yet widely implemented.

Practical Design of Efficiency Indicators The study did not explore how the efficiency indicators need to be designed to be practically usable. This is an important aspect for the future application of these metrics in real-world network optimization.

Scalability and ISP Network Analysis The study did not investigate the scalability of ISP networks or whether each individual link within these networks is always necessary. Understanding these factors is crucial for the broader application of the study's findings.

2.5. Recommendations for Future Research

Design of Practical Efficiency Metrics Future research should focus on designing efficiency metrics that can be effectively used to reduce the carbon footprint of ISP networks. This includes understanding how these metrics can be integrated into existing network management practices.

Standardization Efforts Efforts should be made to push for the standardization of the IOAM Aggregation Option at the IETF. This will make the protocol extension usable on network devices in the foreseeable future, facilitating broader adoption of the study's proposed solutions.

Collaboration with Device Vendors Initiating collaborations with network device vendors is crucial for translating the concepts tested in a simulated environment to physical production switches. Such partnerships will help in understanding the practical challenges and requirements for implementing the proposed solutions in real-world networks.

2.6. Conclusion

This study has made significant contributions to the field of sustainable networking by demonstrating that it is feasible to retrieve detailed energy efficiency data from computer networks in a simulated environment. The findings highlight the potential for significant improvements in network efficiency without impacting performance, paving the way for more sustainable networking practices. Despite the limitations and the need for further research, the study establishes a strong foundation for future work aimed at reducing the carbon footprint of global networking infrastructure. The insights gained from this research mark an important step towards building a more energy-efficient and environmentally responsible network infrastructure.

Bibliography

[1] *9.2. Adding a basic dissector.* URL: https://www.wireshark.org/docs/wsdg_html_chunked/ChDissectAdd.html (visited on 06/02/2024).

[2] Paul Aitken, Benoît Claise, and Brian Trammell. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information.* Request for Comments RFC 7011. Num Pages: 76. Internet Engineering Task Force, Sept. 2013. DOI: 10.17487/RFC7011. URL: https://datatracker.ietf.org/doc/rfc7011 (visited on 03/23/2024).

[3] Alexander Clemm and Laurent Metzger. *Aggregation Trace Option for In-situ Operations, Administration, and Maintenance (IOAM).* Internet Draft draft-cxx-ippm-ioamaggr-00. Num Pages: 9. Internet Engineering Task Force, Oct. 2023. URL: https://datatracker.ietf.org/doc/draft-cxx-ippm-ioamaggr (visited on 03/23/2024).

[4] Alexander Clemm et al. *Green Networking Metrics.* Internet-Draft draft-cx-opsawg-green-metrics-02. Work in Progress. Internet Engineering Task Force, Mar. 2024. 26 pp. URL: https://datatracker.ietf.org/doc/draft-cx-opsawg-green-metrics/02/.

[5] *Docker Compose overview.* en. 100. URL: https://docs.docker.com/compose/ (visited on 06/03/2024).

[6] Kenneth Duda and Arista Networks. "Programmable Network Devices: One Vendor's Perspective". In: (May 2022). URL: https://opennetworking.org/wp-content/uploads/2022/05/Programmable-Network-Devices_-One-Companys-View-2022.pdf.

[7] *FURPS.* In: *Wikipedia.* Page Version ID: 1110147882. Sept. 13, 2022. URL: https://en.wikipedia.org/w/index.php?title=FURPS&oldid=1110147882 (visited on 11/01/2023).

[8] *In Situ OAM (IOAM).* URL: https://www.iana.org/assignments/ioam/ioam.xhtml (visited on 12/10/2023).

[9] *InfluxDB Buckets.* URL: https://docs.influxdata.com/influxdb/v2/admin/buckets/ (visited on 05/20/2024).

[10] *IP Flow Information Export (IPFIX) Entities.* URL: https://www.iana.org/assignments/ipfix/ipfix.xhtml (visited on 03/23/2024).

[11] *P4 architecture.* P4 Programming Language. Feb. 23, 2022. URL: https://forum.p4.org/t/p4-architecture/246 (visited on 06/05/2024).

[12] *P4~16~ Language Specification.* URL: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html (visited on 06/06/2024).

[13] *Risk Matrix.* URL: https://en.wikipedia.org/wiki/Risk_matrix (visited on 03/24/2024).

[14] Mickey Spiegel et al. *IOAM raw data export with IPFIX.* Internet Draft draft-spiegel-ippm-ioam-rawexport-07. Num Pages: 22. Internet Engineering Task Force, Feb. 2024. URL: https://datatracker.ietf.org/doc/draft-spiegel-ippm-ioam-rawexport-07 (visited on 03/23/2024).

[15] *telegraf/plugins/inputs/netflow/README.md at release-1.30 · influxdata/telegraf.* en. URL: https://github.com/influxdata/telegraf/blob/release-1.30/plugins/inputs/netflow/README.md (visited on 05/30/2024).

[16] *telegraf/plugins/processors/starlark/README.md at release-1.30 · influxdata/telegraf*. en. URL: https://github.com/influxdata/telegraf/blob/release-1.30/plugins/processors/starlark/README.md (visited on 06/05/2024).

[17] *Wireshark: Testy, Virtual(-izable) Buffers*. URL: https://www.wireshark.org/docs/wsar_html/group___tvbuff.html (visited on 06/02/2024).