Networked Systems
ETH Zürich — seit 2015

Delay Measurement, Path Tracing, and Telemetry Data Correlation in Segment Routed Networks

Master Thesis

Author: Leonardo Rodoni

Tutors: Tobias Bühler (ETH), Thomas Graf (Swisscom), Marco Tollini (Swisscom)

Supervisor: Prof. Dr. Laurent Vanbever (ETH)

October 2021 to March 2022

Acknowledgements

I'd like to express my deepest thanks to Thomas Graf and Marco Tollini at Swisscom and Tobias Bühler at ETH, whose continuous support and feedback have been extremely important for all aspects of this Thesis. I would also like to thank Prof. Laurent Vanbever at ETH, for enabling this great opportunity. I'm also grateful to Ahmed Abdelsalam at Cisco, for his help and insight. Finally, I would also like to thank Cristina Haldemann, for her continuous motivation and support, as well as all my family and friends.

Abstract

Delay measurement and forwarding path visibility are very important metrics to determine if a network is healthy and to measure its performance. This is especially true when considering ISP core networks built with Segment Routing or MPLS. Being able to collect accurate delays and paths for the various VPN connections through the network would enable many use-cases for visualization and also closed loop operation. We can use these metrics for example to verify correctness of the network configuration, or to quickly identify congestion or failures. Path Tracing is an in-band Telemetry protocol which provides records of forwarding paths and precise delay measurements, both end-to-end as well as hop-by-hop, for packets traversing a Segment Routed network. Thanks to Segment Routing, which allows steering of packets from the source, Path Tracing can be utilized to test both active paths as well as paths that aren't being currently used. In this Thesis we present and evaluate two possible designs of Path Tracing based visualization pipelines which provide real-time delay measurements and forwarding paths. Both pipelines are built with open-source software and were deployed in a virtual Linux environment. We put high importance on enabling correlation of the Path Tracing metrics with telemetry data from other protocols and thus the data exporting method is a key aspect in the discussion.

Contents

Chapter 1

Introduction

Thesis Motivation

A good network monitoring infrastructure is very important for Internet Service Providers (ISPs), since visibility on the network can help verifying that everything is running correctly or quickly detecting problems such as unexpected packet drops or congestion. On the other hand, developing such an infrastructure is not an easy task, especially because most networking protocols are not made to export metrics to analytics [38]. Traditional approaches rely on SNMP to export device-specific management data that helps reconstructing a local view on packet forwarding. This approach, however, lacks the possibility to construct an abstracted visualization of the global network differentiating the different flows. That's the reason why at IETF[1] there are many standardization efforts for network telemetry with the ultimate goal to provide real-time deeper visibility on network operation.

Network Telemetry refers to a newer network monitoring approach, which aims to collect data from devices at high speeds and on real-time. The main difference with respect to traditional protocols, such as SNMP, is that network devices are configured to proactively push data to the collector periodically. Such protocols allow for real-time network visibility, thus paving the way for greater network monitoring, anomaly detection and closed loop operation. Within this work we focus mostly on delay measurements and forwarding path visibility. Delay measurements are important in order to verify the quality of the connections in terms of latency, and to be able to detect congestion in the network at any times. The possibility to reconstruct the forwarding paths of packets instead can be extremely helpful in order to validate network configuration, i.e. to make sure that packets are forwarded through the network as expected, but also to proactively check backup paths or in general any forwarding path which isn't currently being used.

Thesis Scope

Path Tracing is an open-source in-band telemetry protocol developed by Cisco[2], recently submitted as an IETF draft[10]. The protocol provides records of forwarding paths and precise delay measurements, both end-to-end as well as hop-by-hop, for packets traversing a Segment Routed network. Path Tracing adds to various other existing IETF drafts proposing in-band telemetry protocols. In-band telemetry refers to a new approach which integrates network monitoring metrics into live traffic, by inserting metadata into packets as they are traversing the network. Traditionally, delay measurements and path reconstruction were only possible via other active probing protocols like

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

[2] `https://cisco.com`

ping, traceroute or TWAMP[3]. However, these approaches don't produce the real delays and paths of customer packets. In-band telemetry protocols open to the possibility of measuring real metrics of customer data packets as they're traversing the ISP's network.

In this Thesis we present and evaluate two possible designs of Path Tracing based visualization pipelines which provide real-time delay measurements and forwarding path visibility for packets traversing an SRv6 network. Path Tracing is currently only available within the VPP[3] platform, and for this reason, the network that the pipelines monitor is fully virtualized and running on a Linux server.

Within the Thesis, we also examine the correlation possibilities of Path Tracing with other Telemetry Protocols. Considering this and other aspects, we discuss possible improvements and extensions of the Path Tracing protocol with the ultimate goal to integrate it better along existing telemetry frameworks and thus in general granting more flexibility to network operators as to how they can configure it.

Thesis Structure

In Chapter 2 we present an outline on the relevant technologies and protocols utilized or mentioned throughout this Thesis. We start by giving an overview on the common network configurations and operations of ISPs, focusing mainly on Layer 3 VPNs and Segment Routing. We also discuss about industry and research current status in Network Telemetry, while also introducing the Path Tracing protocol. Chapter 3 focuses on presenting the design of the Linux virtual network environment, as well as the design of the two Path Tracing based visualization pipelines. In Chapter 4 we evaluate our pipelines, and we discuss advantages and shortcomings of each, while in Chapter 5 we present some possible improvements both on our work and on the Path Tracing project. Finally, Chapter 6 summarizes the main results and takeaways.

[3] `https://wiki.fd.io/view/VPP`

Chapter 2

Background and Related Work

This Chapter outlines the most relevant technologies and protocols which are important for this Thesis. In Section 2.1, we give an overview on common network configurations and operations of Internet Service Providers (ISPs), focusing mainly on Layer 3 VPNs and Segment Routing. In Section 2.2, we introduce the Vector Packet Processing (VPP) Platform. Then, in Section 2.3, the attention shifts towards Network Telemetry, with a discussion about the current usage in industry and the latest research. We present some of the more common telemetry approaches and protocols, as well as Swisscom's production deployment. We then introduce the new Path Tracing[1] protocol developed by Cisco and published at IETF.

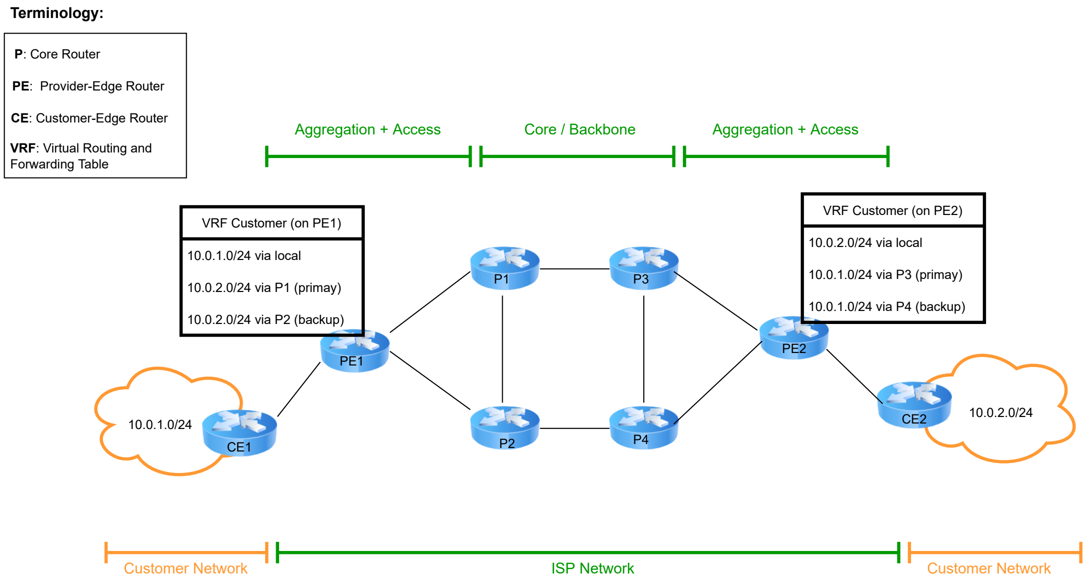
2.1 Internet Service Provider Networks

The main services offered by Internet Service Providers are Internet Access and Transit for both private and business customers. An ISP network has multiple layers, often referenced as core, aggregation and access layers [32]. The access layer represents the so called *last mile*, namely the physical connection of end customers with the ISP. The aggregation layer then gathers all the customers connections to higher speed links and connects them to the core layer. Finally, the core or backbone layer is responsible for providing fast, reliable and redundant data forwarding across the whole ISP network [30]. For this reason the backbone layer is composed of high performance routers, interconnected with high-speed and low-latency optical fiber links arranged in a highly resilient network topology [31].

Core routers need to forward packets as fast as possible, thus a configuration with BGP [19] as control plane protocol and IP as forwarding mechanism would cause too much of an overhead in the forwarding decision. The reason is that the BGP table is very large, with around 900k IPV4 and 100k IPV6 prefixes in 2021 [26], and continuing to grow. This, together with the need for network virtualization, is why most ISPs configure their core network to use MPLS [53] as forwarding mechanism. Multiprotocol Label Switching (MPLS) is a routing technique that uses labels to forward packets instead of IP network addresses [34]. This enables ISPs to build BGP-free backbones, removing the overhead of full routing table lookups from the core layer. Only the edge routers need to retain the full routing table, and use the BGP protocol to distribute the labels representing the forwarding for each prefix through the network. IP routing in the core is only used for internal communication, with the default routing table, so that the backbone routers are able to distribute prefixes and labels. Figure 2.1 presents a sketch of a simple ISP topology, also

[1] `https://github.com/path-tracing`

introducing some relevant terminology and details which will be discussed in the remainder of this section.



Terminology:

P: Core Router

PE: Provider-Edge Router

CE: Customer-Edge Router

VRF: Virtual Routing and Forwarding Table

VRF Customer (on PE1)

10.0.1.0/24 via local

10.0.2.0/24 via P1 (primay)

10.0.2.0/24 via P2 (backup)

VRF Customer (on PE2)

10.0.2.0/24 via local

10.0.1.0/24 via P3 (primay)

10.0.1.0/24 via P4 (backup)

Figure 2.1: Simplified Internet Service Provider Network topology, with a single business customer connecting to the ISP network at two different physical locations. Examples of private IP addresses and routing tables (VRFs) for the business customer are also provided.

Layer 3 VPNs

Businesses and enterprises often need to directly interconnect their various physical locations, and for this purpose a normal Internet connection does not provide enough quality of service (QoS) guarantees. Additionally, public IPv4 addresses are running out nowadays and thus the use of private IPs is preferred, also for security considerations. For these reasons, some the most common services offered by ISPs are Layer 3 Virtual Private Networks (VPNs). A Layer 3 VPN is a type of Virtual Private Network built and delivered on OSI Layer 3 technologies [33], in which the customer participates in IP routing with the ISP, sharing some of their prefixes over a BGP session. The provider then creates separate VRF Tables (Virtual Routing and Forwarding [51]) for each customer, to make sure that routing information is completely isolated from other customers and from the default routing table while it is being tunneled via the MPLS core network.

Figure 2.1 provides an example with VRFs for a business customer. A VRF table is unique for each PE-customer pair. The PE router again uses BPG to distribute customer prefixes, along with the respective labels, through the core towards the other PE routers. A customer-specific *route distinguisher* parameter (RD) is additionally appended to each prefix, to guarantee isolation and uniqueness of the prefixes inside the ISP network. Layer 3 VPNs enable enterprise customers to fully take advantage of the high speed and resiliency of the provider's MPLS core network and thus retain higher QoS guarantees for their interconnections.

2.1.1 Segment Routing

This Section focuses on Segment Routing [13], mainly on its newer implementation based on the IPv6 forwarding technology: SRv6 [12]. Segment Routing can be thought of as the latest evolution of the MPLS technology. It is a source-based routing technique, which aims at simplifying traffic engineering and network management [40]. Segment Routing divides a network path into *segments*, and to each segment it assigns a unique ID (SID) [39]. To form a forwarding path, segments are stacked together building a Segment List. Segment Routing can be deployed either on top of an MPLS network (MPLS-SR) or an IPv6 network (SRv6) [41]. In MPLS-SR, segments are encoded as MPLS labels, while in SRv6 each SID is represented by a full IPv6 Address. SIDs are then stacked to form the desired forwarding path and appended in the Segment Routing Header (SRH), a new type of IPv6 routing extension header which enables Segment Routing capabilities on the IPv6 forwarding plane. RFC 8754 [12] defines all fields and the encoding of IPv6 Segments in the Segment Routing Header.

In some cases, especially within traffic engineering, the SRH can become quite large, due to the high number of SIDs to be included. In some cases this represents an issue, as the ASICs at the hearth of networking devices sometimes struggle in parsing long headers. IETF Draft [7] presents a proposal to enable a compressed encoding of the SRv6 Segment IDs in the SRH to address this issue.

Advantages of SRv6

SRv6 offers multiple advantages when compared to MPLS and also MPLS-SR, mainly thanks to the IPv6 forwarding plane. MPLS labels are no longer required, nor are label distributing protocols. This greatly simplifies configuration and management, as the core network really only needs the ability to route IPv6 packets [42]. In fact, also when extended with a Segment Routing Header, the packets are still native IPv6. Thanks to this, not all routers in the core need to have SRv6 enabled if that's not required for traffic engineering purposes. Backbone routers only need to run an IGP protocol [29] to exchange an IPv6 address range as an IPv6 unicast prefix, which is used locally to assign the various SIDs. The possibility to use IPv6 Addresses instead of labels also removes the limitation on the label space. In fact, the 20-bit MPLS label field only allows to define slightly more than 1 million different labels, which for networks with high level of virtualization might be a big limitation.

Another important advantage of SRv6 over MPLS-SR is the built-in support of route summarization. This is a very useful feature which can greatly simplify inter-AS routing, for providers that have multiple networks for different services which need to be connected. With summarization, a single route that represents all network's routes can be advertized to other networks, thus greatly reducing the number of route exchange messages required and increasing scalability.

SRv6 Forwarding Explained

Figure 2.2 presents an example of a packet traversing an SRv6 network, specifically in the case of an IPv4 Layer 3 VPN, using the previously introduced simplified ISP network topology. In the example, a customer sends a plain IP packet directed to its other physical location through the ISP. Let's assume the packet is sent from the 10.0.1.0/24 network towards 10.0.2.0/24. If the packet matches policy conditions for a SRv6 path, router PE1 encapsulates it with an IPv6 Header and appends an SRH with the Segment List. The stacked SIDs represent the path the packet needs to be directed towards, as well as the behaviour of each traversed node when receiving it.

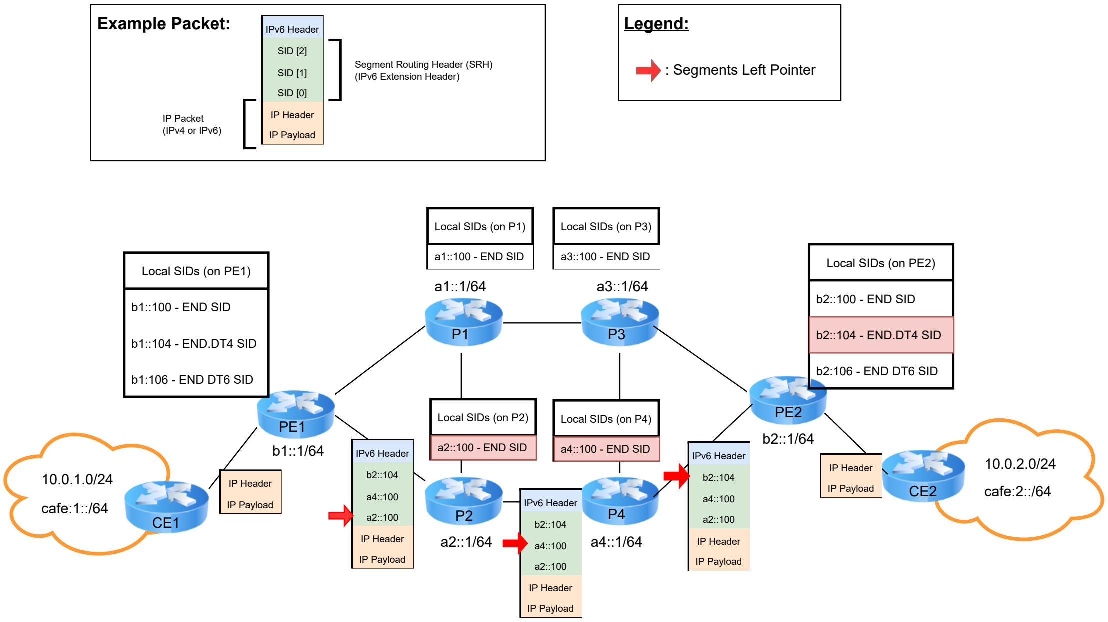


Figure 2.2: IP Packet forwarded through an ISP network using SRv6. The evolution of the IPv6 Segment Routing Header is presented along the various hops of the forwarding path. The Local SID table matches at each router are highlighted in red.

Each ISP router is assigned a */64* network to allocate Segment IDs, and each SID represents a locally defined instruction or function. The instructions could be as simple as just forwarding the packet and updating the SRH as well as any complex user-defined behaviour. RFC 8986 [11] specifies the possible functions that can be configured within SRv6. In order to setup a Layer 3 VPN, supporting both IPv4 and IPv6 customer packets, the following SID behaviours need to be used:

- **END:** Endpoint. The router inspects the SRH, updates the IPv6 destination address on the IPv6 header based on the SID stack and *segments left* pointer and updates the pointer, then forwards the packet based on the default IPv6 routing table.

- **END.DT4:** Endpoint with decapsulation and IPv4 lookup. The router decapsulates the packet by removing the IPv6 header and the SRH, then forwards the packet based on the specific IPv4 VRF table.

- **END.DT6:** Endpoint with decapsulation and IPv6 lookup. The router decapsulates the packet by removing the IPv6 header and the SRH, then forwards the packet based on the specific IPv6 VRF table.

When receiving an IPv6 packet, an SRv6 capable router checks if the IPv6 destination address matches with any of the SIDs inside the *Local SIDs* table, and if that's the case it will execute the respective function.

2.2 The VPP Platform

The Vector Packet Processing (VPP) platform is an open-source high-performance packet-processing stack part of the FD.io[2] project. The VPP framework can run on commodity hardware and provides out-of-the box production quality switch and router functionalities. It is a modular platform built on a packet processing graph, which allows for simple extensibility, since anyone is free to add new graph nodes or modify existing ones by adding a new plugin to the framework [47]. For these reasons, the VPP platform is often used as testing environment for newly developed protocols before they are rolled out to production routers.

VPP uses the vector processing technology, as opposed to the traditional scalar packet processing approach. Scalar packet processing refers to processing one packet at a time. This approach entails handling an interrupt and traversing the relative call stack individually for every single packet [47]. This results in poor performance, since the exact same long sequence of instructions needs to be performed for a large number of packets, often incurring in the same cache misses [50]. The vector packet processing approach aims to solve this problem, by processing multiple packets, i.e. a *vector* of packets, at the same time.

Rather that processing a single packet through the whole processing graph, VPP reads the largest available vector of packets from the network interface. The whole vector is then processed step by step through the graph. This means that before moving to the next node, all packets in the vector need to have finished going through the current node. This results in a high performance increase with respect to the scalar approach, which is proportional to the size of the vector. The reason is that when the first packet goes through a node, the processor loads all the useful data into the cache. The following packets of the vector will then be able to go through the node much faster, since most of the data required for the function calls is still saved into the cache [49].

2.3 Network Telemetry

Network Telemetry refers to both the telemetry data itself as well as the technologies used to produce, export, collect and consume this data. This also includes visualization platforms as well as automated processes or applications that might use the data to automate network configuration and management. IETF draft [23] provides an architectural framework for Network Telemetry, discussing key characteristics, main protocols and how they relate together, with the final purpose of setting a common ground for future reference and standardization.

Drawbacks of the Traditional Approach

Traditional network monitoring techniques such as SNMP [9] rely on pull mode for the interaction between the collector and network devices. The collector submits periodic query requests to each device, normally every second. The devices then parse the requests and send query responses with monitoring information back to the collector [43]. This approach is still used a lot in modern networks, however it is outdated as it can lead to long delays before the information arrives at the collector. This is especially problematic for on-change events: if for example an interface goes down and the polling interval is 10 minutes, in the worst case the outage detection can take up to 10 minutes.

Network Telemetry refers to a newer network monitoring approach, which aims to collect data from devices at high speeds and on real-time. The main difference with respect to traditional

[2] `https://fd.io/`

techniques is that telemetry uses push mode, meaning that devices proactively push data to the collector periodically. Network Telemetry protocols only require an initial subscription exchange between collector and devices, specifying operational parameters such as the collection period, then the devices start generating telemetry data without any further interaction required.

Another drawback of traditional network monitoring approaches is that they only provide device-level information. As an example, SNMP exports counter values for each connected interface. This is helpful to determine the amount of traffic going through the device at a specific time, however it cannot be used to reconstruct a real-time overall view of the network. In fact, other than not providing real-time data, SNMP also lacks a lot of information that would be required in order to perform network-wide correlation at the collector level.

Advantages of Network Telemetry

Network operators often need to troubleshoot connectivity issues by inspecting device-specific data which is a complicated task, especially on large networks [38]. Network Telemetry protocols introduce the possibility to correlate information from multiple layers (forwarding plane, control plane and device statistics) at the data collection level. Thanks to correlation we can finally reconstruct an overall view of the network, allowing operators to visualize network metrics such as flow information or control-plane changes in real-time. Other than greatly improving network monitoring and troubleshooting, this paves the way for greater network automation. With real-time full-network visibility, it would possible to design a closed loop system which reacts on failures by directly triggering device configuration changes and thus dramatically reducing incident response time!

The following Sections introduce some of the Network Telemetry protocols and provide insight on how correlation can be performed in real-time and at scale. Within this discussion we also briefly present Swisscom's current Telemetry deployment. Finally, we introduce the new Path Tracing [10] protocol developed by Cisco and recently published at IETF, which constitutes the basis for the visualization pipeline developed in this Thesis.

2.3.1 Out-of-band Telemetry

Out-of-band Telemetry refers to all protocols that use dedicated traffic (i.e., independent from the customers network traffic) to send telemetry data to the collector [37]. Examples of such protocols are IPFIX [1], BMP [20] and YANG Push [8], the latter is also known as streaming telemetry. All these protocols are based on the push approach and require a subscription or peering with the collector, before they start sending telemetry data. IPFIX exports forwarding-plane metrics while BMP (or BGP Monitoring Protocol) exports BGP sessions and peering information, thus providing the control-plane perspective. YANG push is an IETF standardized subscription based mechanism which enables export of device specific information.

IPFIX

IP Flow Information Export (IPFIX) is an IETF protocol specified by RFC 7011 [1], created as a universal standard for exporting flow information from networking devices. At the device level the protocol works by sampling packets and caching information locally for every observed flow, which is usually defined by the standard 5-tuple (source and destination IP/port and protocol). This procedure is also referenced as flow aggregation, and is described by RFC 7015 [25]. An aggregated flow represents a set of packets from multiple original flows sharing some set of common properties. Sampling is not mandatory, as one could also consider every packet, however is recommended in most cases due to cache size limitations. The local cache is then flushed at regular intervals, while the flow information is sent to the collector. The main metrics exported by IPFIX other than the flow's 5-tuple are usually packet and byte count as well as the time interval in which the flow packets have been sampled. The protocol is also extensible, thus any metrics coming from any existing or newly developed protocols might be added as well. The exported metrics are specified in a *template*, which is also periodically sent to the collector along with the actual IPFIX data. All currently defined metrics are documented by IANA [28].

2.3.2 Swisscom Environment and Technologies

Swisscom has been developing a Network Visualization Big Data Pipeline based on open-source technologies since 2015 [38]. The final goal is fully automating network configuration thanks to trend and anomaly detection using the real-time telemetry data provided by the pipeline. The idea is that the network will be able to react upon failures, congestion or other issues by automatically triggering control plane or device configuration changes. This concept is also known as closed loop operation, and would drastically simplify the job of network operators as well as reduce network downtime. Figure 2.3 presents a highly simplified sketch on how Swisscom's pipeline is configured.

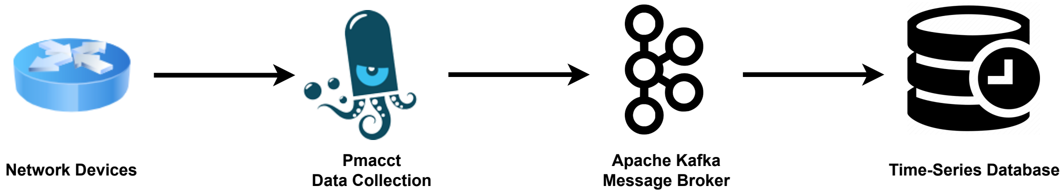


Figure 2.3: Swisscom's Network Visualization Big Data Pipeline. The diagram describes a simplified view of how telemetry data is processed, starting from network devices to being stored in a Time-Series Database.

First, network devices push telemetry packets towards the pmacct collector. Pmacct[3] is an open-source small-set of multi-purpose passive network monitoring tools. Among other tasks it can collect, classify, aggregate and export forwarding plane data via IPFIX; collect and correlate control-plane data via BGP and BMP; and collect infrastructure data via YANG Push [36]. At Swisscom, pmacct is deployed as collector for BMP and IPFIX data, and also aggregates those dimensions together by correlating flow information to Layer 3 VPN information.

Correlation between BMP and IPFIX dimensions is performed by pmacct via the Route Distinguisher (RD) parameter. In newer versions of IPFIX this is directly possible since the RD field is also exported: IANA entity 90, mplsVpnRouteDistinguisher [28]. With older version that don't support RD export, correlation is done using a static Interface-Name to RD mapping that can be constructed via SNMP queries to the network devices.

The aggregated metrics are then produced into the Kafka[4] Message Broker. Apache Kafka is an open-source distributed event-streaming platform used within high-performance data pipelines and streaming analytics. It is very widely deployed and supported by most existing databases backends. Finally, the data is moved from Kafka to a Time-Series Database, with the help of Kafka Connect[5].

[3] `https://github.com/pmacct/pmacct`

[4] `https://kafka.apache.org/`

[5] `https://docs.confluent.io/platform/current/connect/index.html`

2.3.3 In-band Telemetry

Different from previously introduced out-of-band protocols, in-band network telemetry integrates network monitoring metrics into live traffic, by inserting metadata into packets as they are traversing the network [24]. This is usually achieved by encapsulating network packets with an additional header, where the telemetry data is written to by the traversed network devices. Packets are then decapsulated before exiting the telemetry domain, while the metadata is exported and sent to a collector.

IOAM

In-situ Operation, Administration and Maintenance (IOAM) refers to an IETF standardization effort for in-band telemetry protocols. IOAM records operational and telemetry information in the packet while this traverses a path in the network. IETF draft [6] discusses IOAM fields and data types, while IETF draft [5] provides a framework for IOAM deployment along with current best practices. The latter also presents a nice reference on IOAM packet encapsulations within various transport protocols such as IPv6 [4], SR [14] and SRv6 [2]. Some examples of metadata fields collected are traversed node and interface IDs as well as sequence numbers and timestamps for each traversed node. This metadata can be used to precisely determine network paths and compute delays at the data visualization layer. The specifications are flexible, and also any other custom generic data might be added to the packet [37]. This approach enables a whole new level of network monitoring and visualization capabilities which are not possible only using out-of-band protocols.

IOAM is expected to be deployed on a limited network portion, called IOAM domain. Devices who append IOAM metadata to packets entering the domain are called encapsulating nodes, wherease devices which remove it are called decapsulating nodes. Other nodes within the domain that are IOAM aware and might read, write or process IOAM data are called transit nodes [6].

Active and Passive IOAM

Passive IOAM refers to inserting IOAM data into live customer network traffic. This opens the possibility for network operators to very accurately measure real-time delay and visualize forwarding paths of customer packets across the ISP underlay network. Without IOAM these kind of measurements are only possible via other active probing approaches like ping, traceroute or TWAMP [3]. However, these approaches don't produce the real delays and paths of customer packets.

A limitation of a purely passive telemetry system is that it only allows to visualize network performance when traffic is being generated by customers. That is not the case at nighttime, which is also when network operators perform maintenance tasks. Being able to evaluate network performance on real-time during maintenance windows is essential in order to quickly determine if something went wrong and thus minimizing disruptions. This is the reason why allowing to actively trigger the generation of IOAM telemetry packets is also important. Active IOAM refers to generating probe packets containing IOAM data and forwarding it through the network along custom paths, actually simulating customer's traffic.

Combining active with passive IOAM enables the creation of a visualization pipeline capable of covering most network monitoring use cases in an ISP network.

Exporting Metrics: Passport vs Postcard mode

As stated in IETF draft [6], IOAM supports different *option-types*, or categories, representing different use-cases. We are mainly interested in the Trace option-types, as they record traversed node data in the packet, such as node IDs, Interface IDs and timestamps, thus enabling path tracing and delay visualization capabilities. IOAM Trace is inherently conceived to work with passport-based exporting approaches. Passport mode consists of analysing, eventually aggregating and exporting telemetry data only at the decapsulation node, when the packets exit the IOAM domain. This approach has the advantage that the collector already receives full-path information, thus making it very easy to reconstruct forwarding paths and respective hop-by-hop delays. Unfortunately, the passport mode also has some limiting factors. A first disadvantage is that IOAM's hop-by-hop mechanism generates a lot of telemetry data, also since it is lacking filtering and compression [24]. This results in telemetry packets becoming too big, proportionally to the forwarding path length, and thus limiting customers packet size in terms of MTU. Another problem arises at the database level, since full-path information is very expensive to correlate. Even though IOAM information already contains the forwarding path it is not entirely self-contained. When building a visualization pipeline, operators might need to correlate path tracing information with other network dimensions, such as control-plane or device metrics. This is a very CPU expensive task to be performed with full-network table lookups, and it might severely limit the rate at which IOAM enriched packet can be generated in the network. Finally, this approach also doesn't take into consideration an eventual packet drop, which would cause the loss of IOAM data along with the packet itself [24].

To overcome this issues, additional IOAM option-types implementing postcard-based approaches have been proposed. Postcard mode refers to exporting IOAM telemetry data also at every transit node and not exclusively at the decapsulation node. IETF draft [21] defines the Direct Export (DEX) option, which uses an instruction header to trigger nodes to locally aggregate, process and export IOAM data instead of appending it to the packet. IETF draft [22] proposes an alternative approach to DEX that doesn't require an extra instruction header, but only uses a *marking* bit in existing headers to trigger IOAM data export in transit nodes.

Implementations and Proposals

Existing In-band Telemetry protocols include a P4 implementation [35], as well as a VPP implementation [48], and a recently developed linux kernel implementation [16] of IOAM Trace, the latter both based on the IPv6 forwarding plane. The following Section presents a new open-source protocol designed by Cisco which provides in-band telemetry capabilities on the SRv6 data-plane.

2.3.4 Path-Tracing

Path Tracing [10] is an in-band telemetry protocol which provides records of the packet's forwarding path as well as end-to-end delay, per-hop delay and load on each transit node interface. Path tracing stores mid-nodes data by compressing it in a 40 bytes IPv6 Hop-by-hop (HBH) header. Overall it allows to trace up to 14 hops (source, sink and 12 mid nodes). Thanks to compression, it has a lower MTU overhead compared to IOAM Trace. IETF draft [10] defines the path tracing specifications for the SRv6 data-plane, although the technology will be applicable to MPLS-SR as well. An implementation of SRv6 path tracing is available for the VPP platform as a plugin[6], currently only supporting active probing. In the future the possibility to encapsulate customer packets (passive in-band telemetry) might be added as well.

[6] https://github.com/path-tracing/vpp

Figure 2.4 provides a graphical representation of a simple Path Tracing domain as well as the structure of probe packets carrying path tracing data.

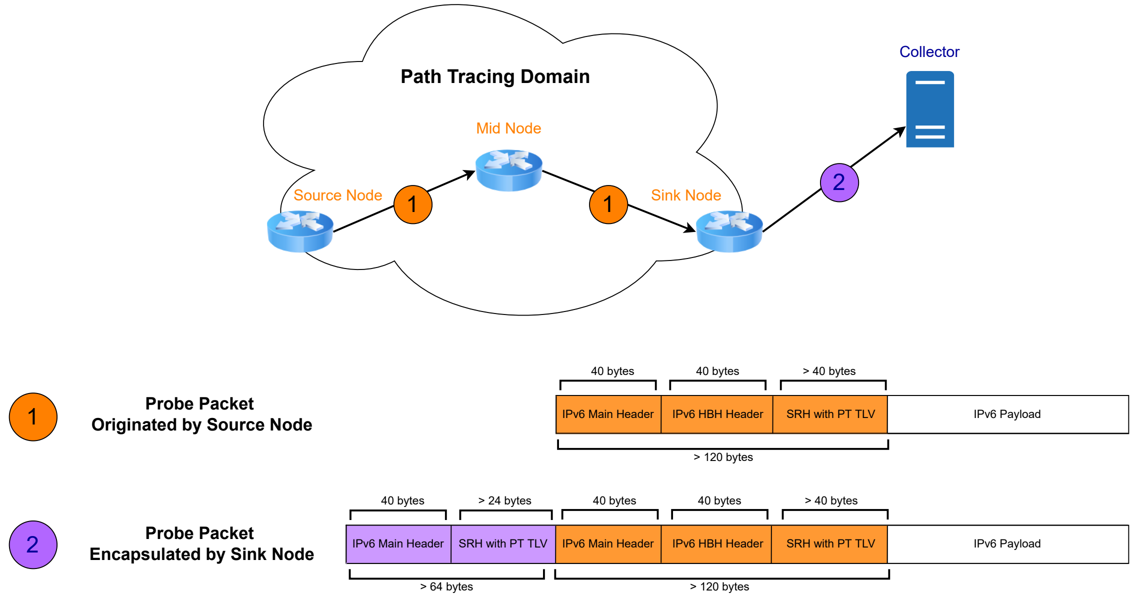


Figure 2.4: The upper part depicts a simple Path Tracing domain with the source, a single mid-node and the sink. The Data-Collector which collects probes from the sink node is also depicted. In the lower part, the structure of probing packets as they're traversing the domain is presented.

Protocol Overview

The source node originates IPv6 probe packets with an IPv6 hop-by-hop option header and a Segment Routing Header. The HBH Header is modified by mid nodes, which write compressed path tracing information to it. The SRH contains the required information for SRv6 steering as well as a TLV field with path tracing information from the source node itself. TLV (type-length-value) is an encoding scheme used to introduce optional informational elements in a certain protocol [44].

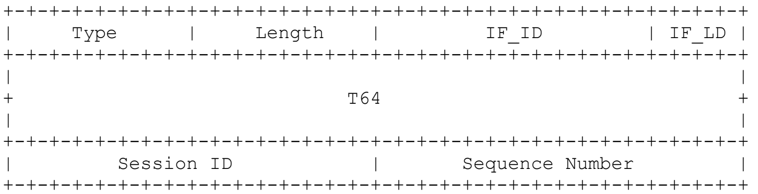
Once the packets reach the sink node, they are once again encapsulated with a new IPv6 Header and a SRH, the latter containing a TLV with path tracing information from the sink node. This behaviour is triggered by a new SRv6 Instruction: **End.B6.TEF** (i.e., *"Endpoint Behavior bound to an SRv6 Policy with Timestamp, Encapsulation and Forward"*). The reason for this additional encapsulation is to redirect probe packets towards the collector, where they are analysed to gather all the collected data into JSON files. Chapter 3 provides more details on all the metrics collected and how they could be visualized within a telemetry pipeline.

The SRH generated by the source contains at least one SID (the End.B6.TEF) required by the sink node to trigger probe packets encapsulation. Therefore its size starts from 40 bytes depending on the SID list length. The SRH pushed by the sink node doesn't need to contain any SIDs, since it's only required in order to store path-tracing information from the sink node before redirecting the packet to the collector. Therefore its size starts from 24 bytes.

SRH Path Tracing TLV

The structure of the new SRH TLV defined for path tracing is depicted in Figure 2.5. For path and delay computation purposes we are interested in the following fields:

- **IF_ID:** 12 bit Interface ID (outgoing or incoming depending if the TLV is generated by source or sink node)
- **T64:** 64 bit PTP Timestamp (reference RFC: [18])



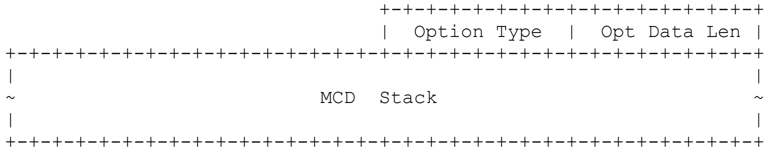
```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |    Length     |         IF_ID         | IF_LD |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                              T64                              +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Session ID            |        Sequence Number        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.5: SRH Path Tracing TLV Format. Source: [10].

IPV6 Hop-by-hop Path Tracing Option

The format of the IPv6 path tracing option is presented in Figure 2.6. Every path tracing enabled mid node along the forwarding path pushes the following information to the MCD (Midpoint Compressed Data) stack:

- **MCD.OIF:** 12 bit interface ID associated with the egress physical port of the router
- **MCD.OIL:** 4 bit outgoing interface load
- **MCD.TTS:** 8 bit truncated timestamp



```
                                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                |  Option Type  |  Opt Data Len |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                          MCD  Stack                           ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.6: IPv6 Hop-by-Hop Path Tracing Option Format. Source: [10].

Currently, the VPP implementation supports a 36 bytes MCD stack (pre-allocated) with capability of storing data for up to 12 mid nodes, hence 3 bytes per node. In future versions this might be customizable by the user, for example by reducing the maximum amount of mid nodes in order to store more information or increase resolution of the hop-by-hop collected data.

The MCD.TTS represents a portion of the full 64 bit PTP timestamp, with different resolutions supported depending on a chosen *template*. The reconstruction of the truncated timestamp as well as the different template structures are explained in detail in Chapter 3. Examples of Wireshark[7] dissected probing packets are available in Appendix A.

Summarizing

To summarize, Figure 2.7 presents a graphical overview of the in-band telemetry technologies introduced in this Chapter, classified by data export approach, and gives an idea on how Path Tracing fits in the overall picture. It is worth mentioning that Path Tracing is still under active development, and in the future it will probably also support the postcard mode. Thanks to the flexible protocol specification with two separate additional headers, the hop-by-hop IPv6 option header could be simply omitted when running path-tracing in postcard mode, triggering data collection, aggregation and export on transit nodes instead.

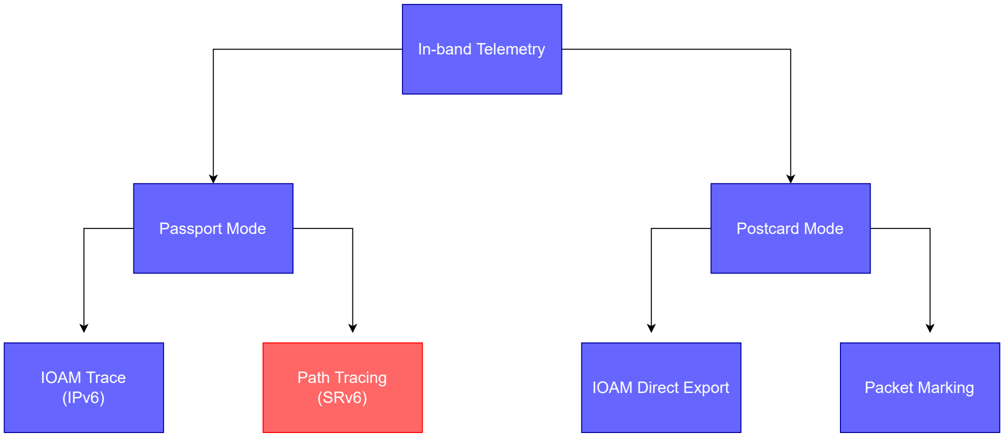


Figure 2.7: In-band telemetry technologies, ordered by telemetry data export methodology. Highlighted in red is the Path Tracing protocol described in this Section.

[7] https://www.wireshark.org/

Chapter 3

Design

This Chapter describes in detail the design and configuration of the whole Path Tracing [10] testing and visualization environment we set up on an Ubuntu Linux server. The goal is to develop a pipeline which enables us to reconstruct paths of packets and visualize relevant metrics, most importantly full-path and hop-by-hop delays in the network. Instructions as well as the full code and binaries required to reproduce the pipeline are available at `https://github.com/rodonile/path-tracing`. Further details on the hardware configuration as well as software versions for all the technologies that were deployed can be found in the main *README* file as well.

In Section 3.1, we first describe the design of the virtual lab environment, which simulates a real ISP network with an SRv6 backbone and Path Tracing capabilities. Then, in Section 3.2, we explain the setup of the pipeline which was deployed to analyze and visualize Path Tracing metrics. Finally, in Section 3.3, we present an alternative approach for visualization, in which we use the IPFIX protocol and the postcard export approach.

3.1 Virtual Lab Environment

This Section provides detailed descriptions, explanations and graphical representations about the setup of the network environment that has been deployed for this Thesis. The environment is fully virtual, since as of today, only a VPP implementation of the Path Tracing protocol is available.

3.1.1 Network Topology

Figure 3.1 shows the reference ISP network which we have deployed for our tests. Our main goal when designing a suitable network topology was to emulate a real ISP network. For this reason we chose a highly redundant core with four P-routers connected in a full mesh. We added four PE-routers, simulating different physical locations, each connected with two P-routers to provide resiliency. To each link we added virtual delays, according to the following pattern: 1ms or 2ms to core links and 3ms to P-PE links. We also introduced three different business customers, which make use of the ISP network to interconnect their various physical locations with a Layer 3 VPN. In order to distinuguish customers' packets we have assigned the *traffic class* (tc) parameter to be consistent with the customer ID. Another important aspect we took into consideration is that the network should have multiple possible paths interconnecting the various customers, which is clearly satisfied in the chosen topology. This enables us to have multiple ECMP paths as well as the possibility to deploy a wide variety of SRv6 steered paths.

Equal-cost multi-path routing (ECMP) is a routing strategy based on per-hop decisions made independently at each router, typically with hash computations. In VPP the hash algorithm

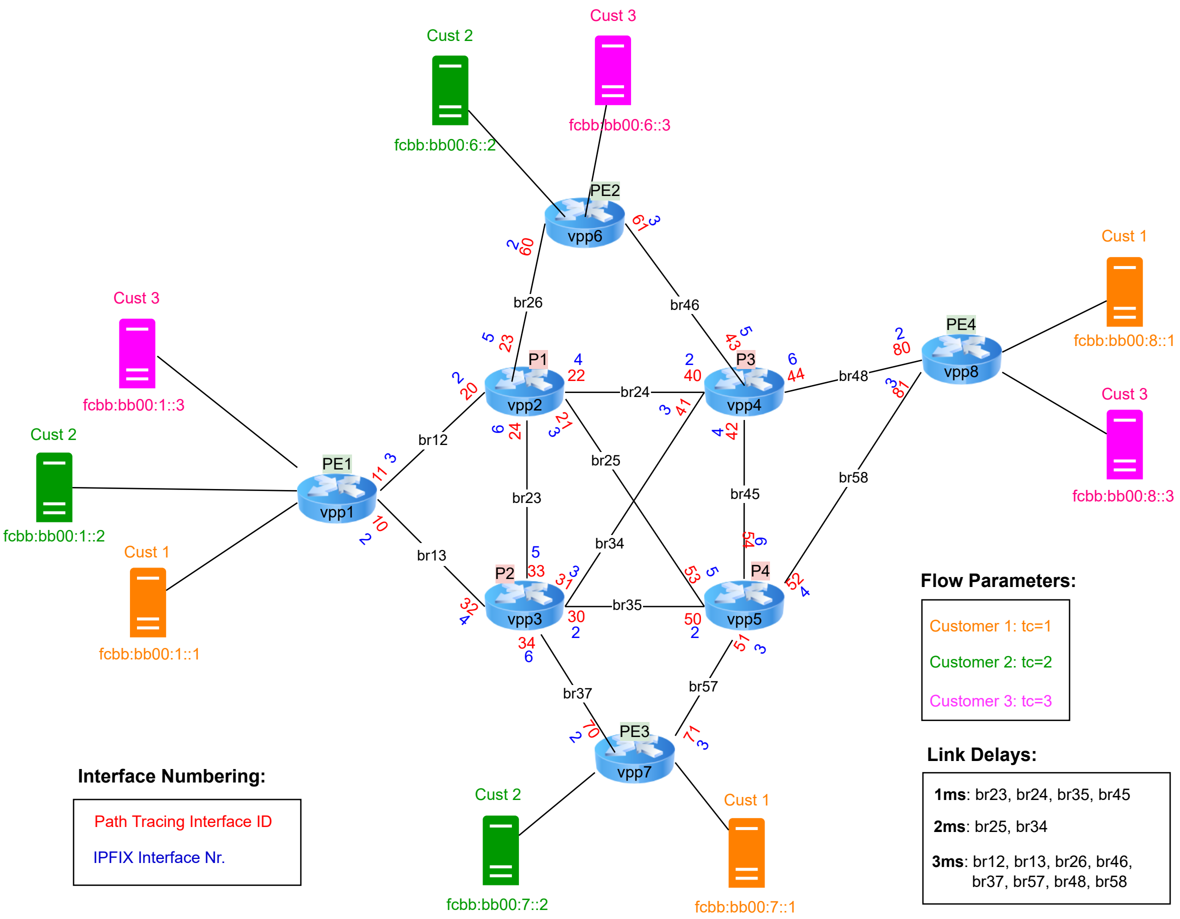


Figure 3.1: Reference ISP-like Network Topology Graph. Multiple nomenclature and parameters are highlighted in the Graph, as it is also meant to be a reference for the interpretation of the visualizations in Chapter 4. Taking the vpp1-vpp2 link as an example, the graph depicts the link name (br12), and interface numberings at the link's endpoints: red for the Path Tracing IDs and blue for the IPFIX IDs. Exact link delay values are available in the bottom right's legend. For each router instead, we both highlight the node name (such as vpp8) and the router type (P or PE router).

takes the following parameters into account: source and destination IP, source and destination port, protocol, and flow-label. With ECMP, packet forwarding towards a single destination can occur over multiple paths, provided that these paths have the same routing priority (hence equal-cost) [52]. For the remainder of this Thesis we will reference to such paths as *ECMP paths*. Still taking the network from Figure 3.1 as reference, let's consider packets originating from customer 1's vpp1 location (source IP: *fcbb:bb00:1::1*) destined at customer 1's vpp8 location (destination IP: *fcbb:bb00:8::1*). Since we haven't assigned any particular weights to the links within the IGP routing protocol, all links have the same routing priority. For this reason, the packets have four different ECMP paths to get from source to destination: *vpp1-vpp2-vpp4-vpp8, vpp1-vpp2-vpp5-vpp8, vpp1-vpp3-vpp4-vpp8*, and *vpp1-vpp3-vpp5-vpp8*. Traffic from different flows will be load balanced across these paths.

3.1.2 Network and Testing Environment on Linux

In order to build the virtual network based on our reference in Figure 3.1, we have written a bash script[1] that takes care of all the necessary Linux and VPP configuration. The script is based on another simpler script[2], kindly provided by the Path Tracing developers at CISCO, which sets up a VPP network with SRv6 and Path Tracing configurations.

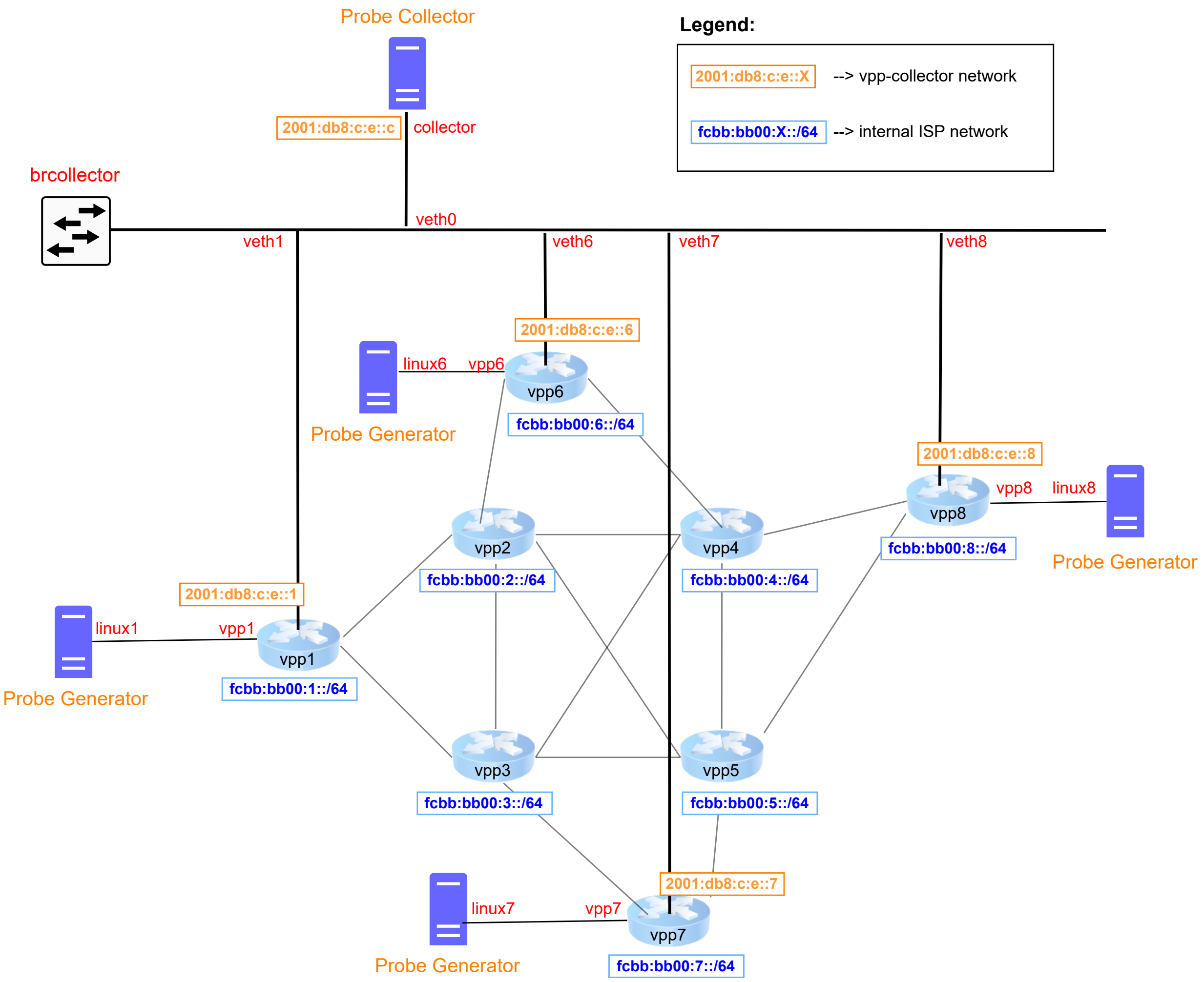


Figure 3.2: Virtual Network Environment Configuration on the Linux Server. The graph highlights the connections between VPP nodes and probe generators processes, which are implemented with Linux virtual interfaces. The connection between Path Tracing sink nodes and the probe collector process is carried out with a Linux bridge. IPv6 addresses for both the internal VPP network as well as the connections with the probe collector (VPP-collector network) are also depicted.

Figure 3.2 shows the actual network topology that we configured on the Linux server, and gives more insight on how the virtual environment is actually built. The setup script starts and

[1] https://github.com/rodonile/path-tracing/blob/main/pipeline/setup-network.sh

[2] https://github.com/path-tracing/scripts/blob/0ec4d7ce8ef007a8cec2402c092f336f1656d0e1/vpp-lab-setup/setup-testbed.sh

configures the VPP instances, Linux interfaces and bridges. It also configures routing entries and SRv6 parameters to enable routing and forwarding between the VPP instances. The script also starts probe generation processes to send packets throughout the network. Finally, it launches a *tmux* window providing an overview of the running processes as well as the possibility to stop or restart the probe generation sessions. In the remainder of this Section, some important aspect of the script are described in more detail. For further information on the virtual network configuration refer to the script itself, which is extensively commented.

Network Topology Mapping File

While bringing up the network, the script also generates a static topology mapping file (*network_mapping.json*) in the working directory, which can be used to reconstruct the network topology graph. For the interface as well as the linux bridges numbering schemes refer to Figure 3.1. The mapping file includes information for every interface on the network. Here is a section of the file with the information for interface 10 on vpp1:

```
    "10":                           # Interface ID (used by Path Tracing)
    {
      "node_id": "vpp1",
      "interface_name": "tap10",
      "interface_idx": 2,           # Interface IDx (used by IPFIX)
      "linux_bridge": "br13",
      "connected_interface": 32     # ID of the connected Interface
    }
```

This mapping file will be used by the visualization pipeline to perform full network topology correlation on the Path Tracing and IPFIX telemetry packets.

SRv6 and Path Tracing configuration

The following snippet provides an example on how we configured SRv6 and Path Tracing parameters on vpp1, which is both a source and a sink node:

```
# Enable Path Tracing on Interfaces 10 and 11
pt iface add iface tap10 id 10 tts-template 2
pt iface add iface tap11 id 11 tts-template 2

# Set host interface vpp1 as Path Tracing probe generator source
pt probe-inject-iface add iface host-vpp1

# Define END.SID (for SRv6 steering)
sr localsid address fcbb:bb00:1::100 behavior end

# Set SRv6 source address for encapsulation (for probes redirected to collector)
set sr encaps source addr 2001:db8:c:e::1

# Define behaviour for END.B6.TEF SID
sr policy add bsid fcbb:bb00:1:f0ef:: next 2001:db8:c:e::c encap tef
```

Thanks to the last policy configuration, if vpp1 receives an IPv6 packet with destination matching the address *fcbb:bb00:1:f0ef::*, it will encapsulate the packet with an additional IPv6 header and an SRH and redirect it to the collector according to Path Tracing protocol specifications.

Probing Sessions

As can be seen in Figure 3.2, the probe generator process is not embedded into the VPP node. Instead it needs to be run as a separate set of binary processes, also developed and made available to us by CISCO. In future IOS releases, they will be also integrated into the router OS. Currently there are three binaries: *ptprobegen* (the probe generator), *ptprobegen-client* (a client to interact with probe generators) and *probe-collector* (the collector process that receive the Path Tracing probe packets and gathers the relevant metrics). The setup script takes care of starting the processes and initiates probing sessions such that packets are generated in the network simulating customer traffic as displayed in Figure 3.1. The following snippet presents an example on how we are generating probes to simulate customer 1's traffic between its locations on vpp1 and vpp8:

```
./ptprobegen-client --fls=720 --fle=1080 --ppf=100 --pps=10 --tc=1
   --src-addr=fcbb:bb00:1::1
   --tef-sid=fcbb:bb00:8:f0ef::
   --segment-list=’fcbb:bb00:2::100,fcbb:bb00:2::100,fcbb:bb00:5::100,fcbb:bb00:8::100’
   --ptprobegen=127.0.0.1:50001
```

The *ptprobegen-client* binary issues a request to *ptprobegen* (the probe generator process for vpp1 listening at 127.0.0.1:50001) specifying the session parameters. In addition to source address and END.B6.TEF SID, other flow parameters that we can configure are flow label start/end (fls/fle), number of packets per flow (ppf), number of packets per second (pps) and traffic class (tc). Additionally in this case we are also configuring the following SRv6 steered path for the probing packets to traverse: vpp1 - vpp2 - vpp5 - vpp8.

Conclusion

After the virtual network is online, the setup script also launches some python programs that we have written, whose purpose is to analyze the telemetry information coming from the probe collector and from IPFIX. As these programs are part of the visualization pipeline, their functionalities are explained in the following sections.

3.2 Main Visualization Pipeline

This Section describes the design, as well as installation and configuration of our main Telemetry Pipeline for visualizing delays and paths of packets in our virtual network environment using the Path Tracing protocol. All software and technologies used as well as our pipeline are free and open-source and thus reproducible by anyone. The design and chosen technologies were largely inspired by Swisscom's Network Telemetry Environment, which was briefly introduced in Chapter 2. A graphical overview of the pipeline and its main processes is depicted in Figure 3.3.

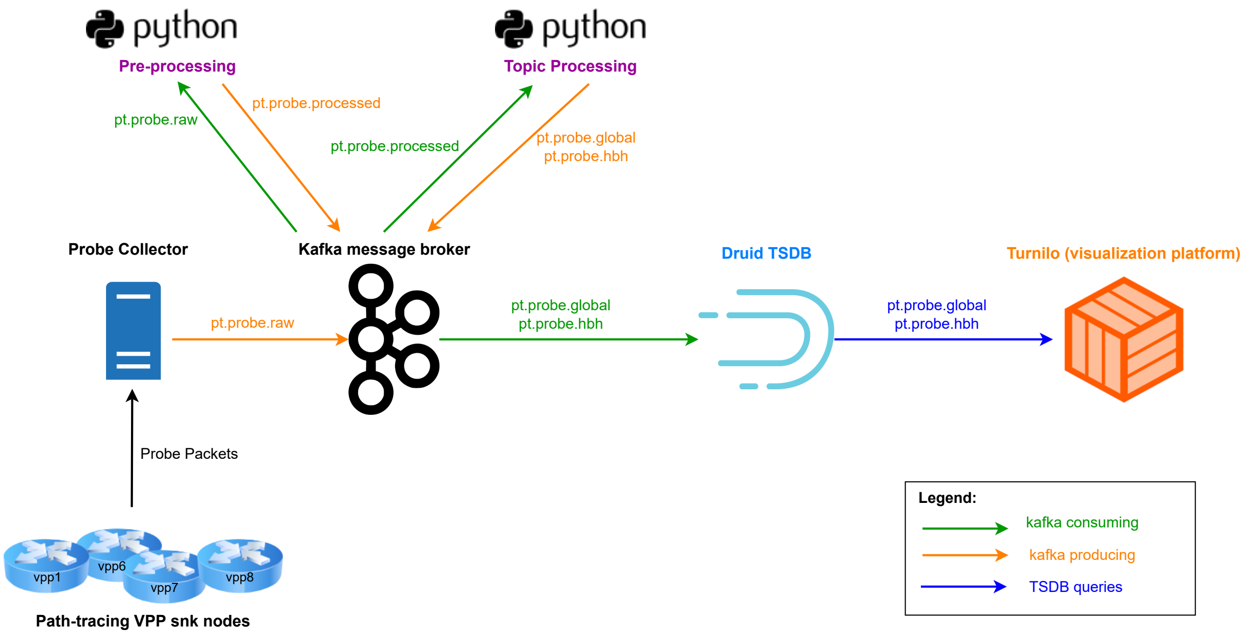


Figure 3.3: Path Tracing Main Visualization Pipeline. The diagram shows all processes and technologies used within the pipeline, while highlighting the overall pipeline's operation and evolution of the telemetry data.

The Probe Collector process receives all probe packets from the sink nodes, as expected by the Path Tracing protocol specification, which in our case are the PE routers vpp1, vpp6, vpp7, and vpp8. When receiving a probe packet, the collector gathers all the relevant metrics from the various headers and writes them into a JSON object, which is then produced into a Kafka Broker in the **pt.probe.raw** topic. Subsequently, the JSON object is consumed, processed and re-injected into Kafka by a set of python processes, whose scope is to decode and enrich the raw metrics so that they can be better visualized. These processing steps are explained in greater detail throughout the remainder of this section.

Afterwards, the processed JSON objects are ingested into a Druid instance. Apache Druid[3] is an open-source real-time analytics database specifically designed for use cases such as real-time ingestion and fast query performance [27]. Druid is also the TSDB of choice in the Swisscom Telemetry environment.

[3] https://druid.apache.org/

Finally, as visualization backend we opted to use the open-source Turnilo[4] software. Turnilo is an open-source data exploration and visualization web application for Apache Druid. It is a fork of Pivot[5], which is currently available under commercial license only [45].

3.2.1 Installation and Configuration

Since this pipeline is based on a virtual network and is conceived for testing purposes we have opted to deploy the required software within Docker containers, wherever possible, as this is the quickest and easiest way to guarantee compatibility on multiple platforms. The Druid instance and the Kafka Broker were both deployed as containers. A *docker-compose* file and instructions on how to deploy it are available in the github repository for this Thesis. Turnilo instead was installed directly on the server since a Docker image wasn't provided by the project maintainers. All the configuration files required to get the software working with our pipeline are also available on the repository.

3.2.2 Pre-Processing

The python preprocessing script, as depicted in Figure 3.3, consumes the raw metrics from the **pt.probe.raw** topic, processes them and produces the resulting JSON object in the **pt.probe.processed** topic. Listing 3.1 provides an example JSON message as it is produced by the probe collector.

```
{
"src_node": {
    "addr": "/Lu7AAABAAAAAAAAAAAAAQ==",
    "t64": 7049743939458283446,
    "out_interface_id": 11
},
"snk_node": {
    "addr": "/Lu7AAAIAAAAAAAAAAAAAQ==",
    "t64": 7049743940258383836,
    "in_interface_id": 81,
    "tef_sid": "IAENuAAMAA4AAAAAAAAADA=="
},
"mcd_stack": "A+ApArAuAWAyAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA",
"path_info": [
    {
        "node_id": "",
        "t64": 0,
        "out_interface_id": 0,
        "out_interface_load": 0,
        "out_interface_name": ""
    },
    ...
]
}
```

Listing 3.1: pt.probe.raw JSON message.

[4] https://github.com/allegro/turnilo

[5] https://imply.io/imply-pivot/

For length reasons, JSON snippets as well as all other listings in this section have been shortened to include only the most relevant metrics in order to explain the processing operations. Examples of full JSON messages can be found in Appendix B.

The pt.probe.raw JSON messages, as can be seen in Listing 3.1, already contain fields for all the metrics that are collected along the network. However, many of those fields aren't populated yet or set to "0", while others contain *base64* encoded strings. The pre-processing script handles the decoding of these fields, and the population of the missing ones. Listing 3.2 shows an example of a JSON message produced in the pt.probe.processed topic.

```
{
    "src_node": {
        "node_id": "vpp1",
        "addr": "fcbb:bb00:1::1",
        "t64": 7057538678431652836,
        "out_interface_id": 11
    },
    "snk_node": {
        "node_id": "vpp8",
        "addr": "fcbb:bb00:8::1",
        "t64": 7057538678452034330,
        "in_interface_id": 81,
        "tef_sid": "2001:db8:c:e::c"
    },
    "midpoint_count": 3,
    "path_info": [
        {
            "node_id": "vpp6",
            "t64": 7057538678441902080,
            "out_interface_id": 62
        },
        ...
    ]
}
```

Listing 3.2: pt.probe.processed JSON message.

In order to decode hop-by-hop information, the mcd_stack needs to be read in chunks of 3 bytes. For a reference on how the stack is constructed refer to Chapter 2. The HBH information is highly compressed, only including shortened interface IDs, interface load and truncated timestamps. To populate full node and interface IDs, the script correlates with the statically generated network topology mapping file (*network_mapping.json*), previously introduced in Section 3.1.2.

Timestamp Reconstruction and Rollover Correction

The pre-processing script handles reconstruction of the 8-bit truncated timestamp into a full 64-bit PTP timestamp (T64). The 64-bit PTP timestamp (or PTP truncated timestamp format, defined in [18]) is composed of a 32-bit part representing the integer portion, and a 32-bit part specifying the fractional portion of seconds (in nanoseconds) since the epoch. The timestamp format is depicted in Figure 3.4.

Path Tracing currently supports 4 different templates, whose details are shown in Table 3.1, in order to accommodate for different resolutions. It's the network operator's decision how to configure each link's template, depending on the type of link and its average delay.

```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Seconds                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Nanoseconds                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.4: 64-bit PTP Timestamp Format. Source: [18].

	PTP bits	**Rollover (ms)**	**Precision (ms)**	**Use-case**
Template 0	Nanoseconds [12:19]	1.04448	0.004096	Directly connected (DC) links
Template 1	Nanoseconds [14:21]	4.17792	0.016384	Directly connected (DC) links
Template 2	Nanoseconds [18:25]	66.84672	0.262144	WAN links
Template 3	Nanoseconds [19:26]	133.69344	0.524288	Intercontinental links

Table 3.1: Path Tracing supported TTS Templates. For each Template the table describes which PTP bits are taken into account and the suggested use-case relative to the link type. Rollover intervals as well as measurement's precision are also stated. Source: CISCO.

Timestamp reconstruction is performed in a backwards manner, taking as reference the next node's timestamp along the path. Following template specifications, the respective PTP bits (in the nanoseconds portion) of the next hop's timestamp are substituted with the truncated bits, while the remaining LSB bits are set to 0. The result is the reconstructed T64 timestamp for the current node.

It is important to note that a rollover check always needs to be performed on the reconstructed timestamps. Basically, we need to make sure that the reconstructed timestamp is not bigger than the reference next node's timestamp. If that's the case, a rollover happened and the timestamp value needs to be corrected. Refer to the script itself for further insight on the calculations.

Considerations

The reason why we have to perform this pre-processing steps is because the Path Tracing project is still under active development. In the final release, there will be a probe analyzer process (that is currently in development) which will perform these tasks and produce the already decoded metrics into Kafka. Hence, this pre-processing step will not be required in order to build a visualization pipeline for Path Tracing in the future.

3.2.3 Topic Processing

The Topic Processing script, like illustrated in Figure 3.3, consumes from pt.probe.processed and produces JSON messages in two new topics: **pt.probe.global** and **pt.probe.hbh**. The purpose of this script is to enrich the metrics and prepare them for Druid ingestion and visualization.

```
{
    "path_info": {
        "nodes_path": "vpp1 --> vpp2 --> vpp5 --> vpp6 --> vpp8",
        "delay": 20.32512
    },
    "sid_list_full": "fcbb:bb00:2::100 - fcbb:bb00:5::100 - fcbb:bb00:6::100 -
        fcbb:bb00:8::100 - fcbb:bb00:8:f0ef::"
}
```

Listing 3.3: pt.probe.global JSON message.

Listing 3.3 provides a snippet of a JSON message produced in the pt.probe.global topic. This information is meant to be used for visualization of full-path dimensions. Within this message we generate strings with aggregated full-path and SID list, as well as an already computed full-path delay.

The metrics produced into the pt.probe.hbh topic are instead aimed for visualization of per-link dimensions. To produce messages in this topic we use a technique called *explosion*. This results in generating multiple messages from a single JSON object consumed from pt.probe.processed, one for each link along the path.

```
{
    "link_info": {
        "src_node_id": "vpp6",
        "dst_node_id": "vpp8",
        "src_interface_id": 62,
        "dst_interface_id": 81,
        "link_id": "br68",
        "src_t64": 7057539335198867456,
        "dst_t64": 7057539335209112773,
        "delay": 10.245376
    }
}
```

Listing 3.4: pt.probe.hbh JSON message.

Listing 3.4 provides a snippet of a JSON message produced in the pt.probe.hbh topic. In each new message, we append a *link_info* field containing link identification as well as timestamps and computed delay.

3.3 IPFIX Integration and Postcard-based Pipeline

This Section describes the design of an alternative visualization pipeline, which makes use of the IPFIX protocol to export metrics to a collector. Since we use IPFIX we do not receive already correlated data from the sink nodes only, but aggregated local metrics from each VPP node in the network (postcard-based export) instead. A graphical overview of the pipeline and its main processes is given by Figure 3.5.

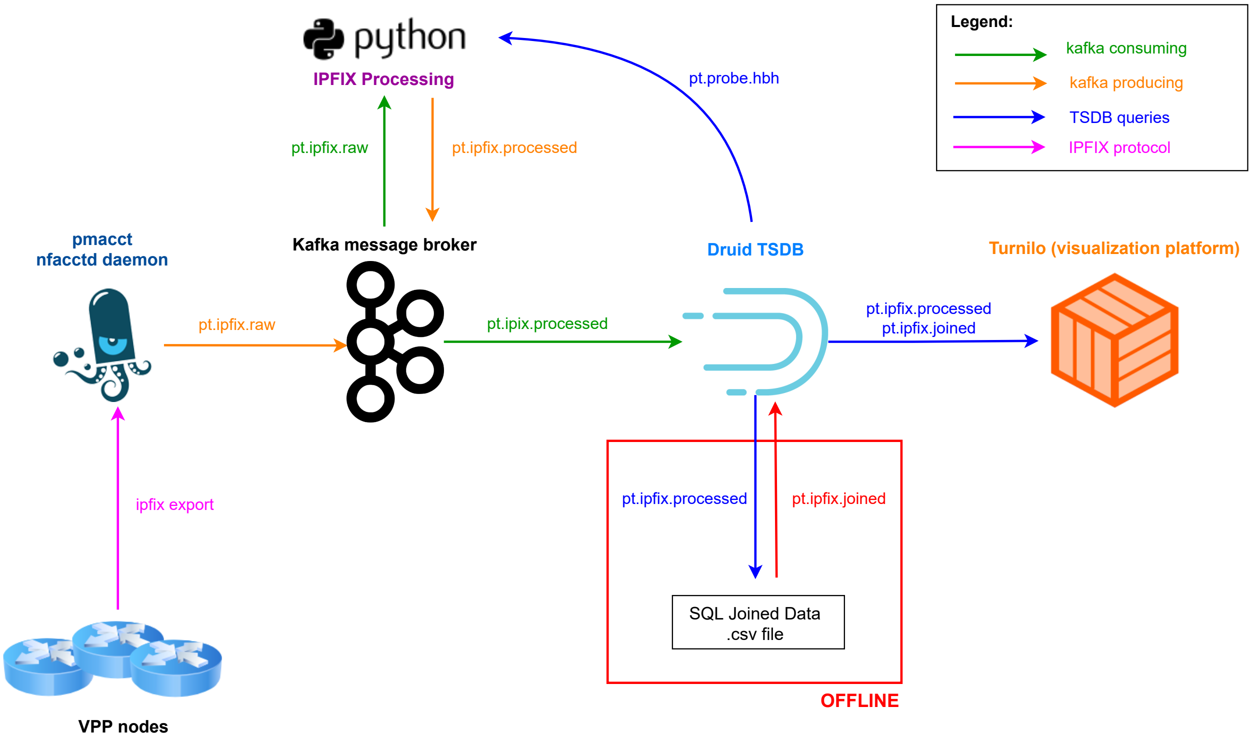


Figure 3.5: Path Tracing Postcard-based Pipeline with IFPIX. The diagram shows all processes and technologies used within the pipeline, highlighting the evolution of the IPFIX data. As will be explained in the remainder of this Section, some functionalities of the pipeline are not automated, hence need to be performed manually and offline.

We designed this alternative pipeline mostly as a *concept demonstration* to give an idea on what is possible with Path Tracing and the postcard export approach. IPFIX packets are sent by all VPP nodes to a pmacct collector daemon, which aggregates them once again and periodically produces them in the **pt.ipfix.raw** Kafka topic.

Since Path Tracing currently only supports the passport mode, in order to run this alternative pipeline we require the main pipeline to be running as well. In fact, we use the same infrastructure for storage and visualization. The reason is that, as will be explained in the remainder of this section, we correlate with Path Tracing metrics to enrich the IPFIX messages with link delay measurements.

3.3.1 IPFIX Processing

As shown in Figure 3.5, the IPFIX Processing script consumes from the pt.ipfix.raw topic and produces delay enriched metrics into the **pt.ipfix.processed** topic. Concretely, the script queries the pt.probe.hbh datasource on Druid over the IPFIX sampling interval to retrieve average, minimum and maximum delay measurements for the relevant link. To map IPFIX indexes with Path Tracing Interface IDs as well as to recover full node and interface IDs we correlate with the static mapping file (*network_mapping.json*) introduced in Section 3.1.2.

Listing 3.5 shows a snippet of a JSON message produced by the script. The metrics on the upper block are the raw IPFIX metrics, and all the additional metrics added by the script are in the lower block. Full JSON messages for both the pt.ipfix.raw and pt.ipfix.processed topics are available in Appendix B. As can be seen in Listing 3.5, we now have an IPFIX message with delay information, similar to a message that could be originating from routers supporting Path Tracing in postcard mode.

```
{
    "peer_ip_src": "192.168.0.4",
    "iface_in": 3,
    "iface_out": 6,
    "ip_src": "fcbb:bb00:1::1",
    "ip_dst": "fcbb:bb00:8:f0ef::",
    "timestamp_export": "1645613144.000000",
    "packets": 64,
    "bytes": 7680,

    "peer_id": "vpp4",
    "link_in": "br34",
    "link_in_connected_node": "vpp3",
    "iface_in_avg_delay": 2.0519147610619424,
    "iface_in_max_delay": 2.359296,
    "iface_in_min_delay": 1.835008,
    "link_out": "br48",
    "link_out_connected_node": "vpp8",
    "iface_out_avg_delay": 3.128008517110266,
    "iface_out_max_delay": 3.41376,
    "iface_out_min_delay": 2.777088
}
```

Listing 3.5: pt.ipfix.processed JSON message.

In reality, from a postcard implementation of Path Tracing, we would receive delay measurements from source to the peer node, and not only for the previous link. We also wouldn't receive delay information relative to the outgoing node, because that would be provided by the next hop. We had to introduce this information to overcome some limitations of IPFIX in VPP, which comes from its missing SRv6 support. For instance, when a packet is encapsulated (which in our network happens at the sink node), IPFIX doesn't record it. This limitations as well as other issues we had with IPFIX will be discussed more in detail in Chapter 4.

3.3.2 Offline Database Queries and Joins

With the postcard approach we don't have a message which already contains the packet's forwarding paths. This information needs to be reconstructed by correlating the different IPFIX packets at the database level. This is possible for example via SQL joins, correlating over source and destination IP addresses and matching outbound interface with inbound interface. Listing 3.6 shows a snippet of the SQL query we use to correlate two ipfix messages together, which allows us to reconstruct a 4-hop path in the network. The full SQL queries that we have used can be visualized in Appendix C.

```
SELECT ipfix1.__time AS __time,
       ipfix1.ip_src AS ip_src,
       ipfix1.ip_dst AS ip_dst,
       ipfix1.link_in AS link_1,
       ipfix2.link_in AS link_2,
       ipfix2.link_out AS link_3,
       (...)
FROM "pt.ipfix.processed" ipfix1
INNER JOIN "pt.ipfix.processed" ipfix2 ON (ipfix1.link_out = ipfix2.link_in and
    ipfix1.ip_src = ipfix2.ip_src and ipfix1.ip_dst = ipfix2.ip_dst)
```

Listing 3.6: Druid SQL query with inner join

Listing 3.7 shows a snippet of a JSON message produced with the SQL query described above. Only the most important metrics to understand the inner join are shown here. Complete JSON messages are available in Appendix C.

```
{
    "__time": "2022-02-24T07:37:54.446Z",
    "ip_src": "fcbb:bb00:8::1",
    "ip_dst": "fcbb:bb00:1:f0ef::",
    "link_1": "br48",
    "link_2": "br34",
    "link_3": "br13",
    ...
  }
```

Listing 3.7: pt.ipfix.joined JSON message

SQL querying and joining functionality are fairly new features in Apache Druid, only available since 2021. The Pivot visualization tool supports them since February 2022. Turnilo, however, does not support SQL yet [46]. For this reason, and since anyway this alternative pipeline is mainly conceived as a proof of concept, we opted for an *offline* approach. Concretely, we downloaded the query results from the druid GUI and reingested them in a new **pt.ipfix.joined** datasource, ready to be visualized by Turnilo.

Chapter 4

Evaluation

This Chapter presents and discusses the various visualization capabilities of the two pipelines, whose design has been described in Chapter 3. In Section 4.1 we concentrate on the main passport-based pipeline by showing examples of what can be visualized and also relating to possible production use-cases for delay and forwarding path visualization. In Section 4.2 we focus on the alternative postcard-based pipeline which uses IPFIX. Finally, in Section 4.3, we compare this alternative pipeline against our main pipeline, discussing advantages, disadvantages and possible use-cases for each.

4.1 Main Visualization Pipeline

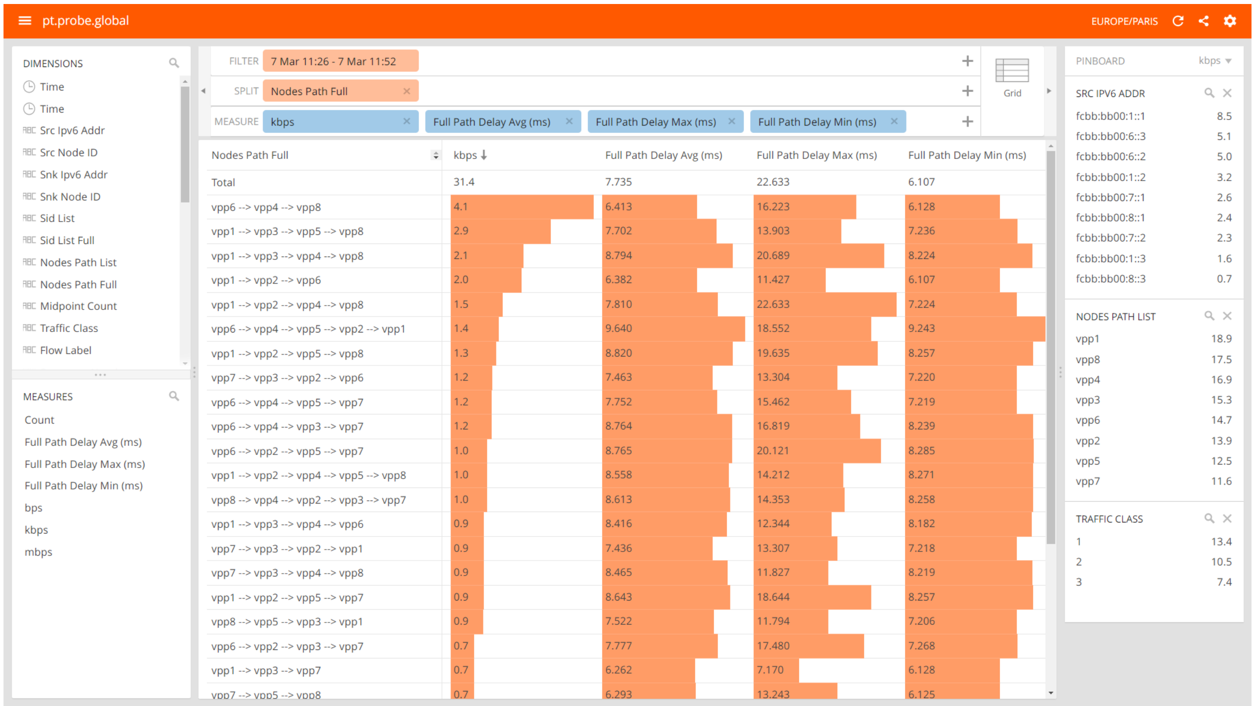
In order to give more context to the visualizations refer to the Network Topology introduced in Chapter 3. In our virtual network environment there are three business customers, and to distinguish each customer's packets we have configured the *traffic class* (tc) parameter to match the customer ID (e.g. customer 1's packets have tc = 1). Link delays range from 1ms to 2ms in the core (P-P links), whereas to the PE-P links we have configured 3ms delays. For each customer we initiated a set of probing sessions with both SRv6-steered as well as best effort (i.e. ECMP) paths; and also a combination of both.

The visualizations included in this Section can also be thought of as a live debugging procedure, starting from a high level network visibility looking at full paths and global end-to-end delay; then in a top-down approach applying filters to get more detailed and customer specific information. Finally, we end up with per-link or per-hop metrics.

4.1.1 Full-Path Visualizations

Figure 4.1 displays forwarding paths for all customers in the network, and for each path shows bandwidth (kbps) as well as average, maximum, and minimum delay computed over the selected time interval. It also introduces the Turnilo visualization platform and gives an idea on its capabilities. Turnilo allows for high flexibility with many filtering and splitting possibilities, while at the same time being very intuitive and easy to use. On the right side panel we see IPv6 source addresses for all customers, which we could use to filter traffic, as well as all nodes and traffic classes.

We have chosen to visualize multiple delay-related metrics, in order to give the most insight possible on the real behaviour of the network. Only visualizing for example the average delay might lead in not detecting some issues. It could be the case that most probes don't experience problems,



FILTER 7 Mar 11:26 - 7 Mar 11:52

SPLIT Nodes Path Full

MEASURE kbps | Full Path Delay Avg (ms) | Full Path Delay Max (ms) | Full Path Delay Min (ms)

Nodes Path Full	kbps ↓	Full Path Delay Avg (ms)	Full Path Delay Max (ms)	Full Path Delay Min (ms)
Total	31.4	7.735	22.633	6.107
vpp6 --> vpp4 --> vpp8	4.1	6.413	16.223	6.128
vpp1 --> vpp3 --> vpp5 --> vpp8	2.9	7.702	13.903	7.236
vpp1 --> vpp3 --> vpp4 --> vpp8	2.1	8.794	20.689	8.224
vpp1 --> vpp2 --> vpp6	2.0	6.382	11.427	6.107
vpp1 --> vpp2 --> vpp4 --> vpp8	1.5	7.810	22.633	7.224
vpp6 --> vpp4 --> vpp5 --> vpp2 --> vpp1	1.4	9.640	18.552	9.243
vpp1 --> vpp2 --> vpp5 --> vpp8	1.3	8.820	19.635	8.257
vpp7 --> vpp3 --> vpp2 --> vpp6	1.2	7.463	13.304	7.220
vpp6 --> vpp4 --> vpp5 --> vpp7	1.2	7.752	15.462	7.219
vpp6 --> vpp4 --> vpp3 --> vpp7	1.2	8.764	16.819	8.239
vpp6 --> vpp2 --> vpp5 --> vpp7	1.0	8.765	20.121	8.285
vpp1 --> vpp2 --> vpp4 --> vpp5 --> vpp8	1.0	8.558	14.212	8.271
vpp8 --> vpp4 --> vpp2 --> vpp3 --> vpp7	1.0	8.613	14.353	8.258
vpp1 --> vpp3 --> vpp4 --> vpp6	0.9	8.416	12.344	8.182
vpp7 --> vpp3 --> vpp2 --> vpp1	0.9	7.436	13.307	7.218
vpp7 --> vpp3 --> vpp4 --> vpp8	0.9	8.465	11.827	8.219
vpp1 --> vpp2 --> vpp5 --> vpp7	0.9	8.643	18.644	8.257
vpp8 --> vpp5 --> vpp3 --> vpp1	0.9	7.522	11.794	7.206
vpp6 --> vpp2 --> vpp3 --> vpp7	0.7	7.777	17.480	7.268
vpp1 --> vpp3 --> vpp7	0.7	6.262	7.170	6.128
vpp7 --> vpp5 --> vpp8	0.7	6.293	13.243	6.125

SRC IPV6 ADDR	
fcbb:bb00:1::1	8.5
fcbb:bb00:6::3	5.1
fcbb:bb00:6::2	5.0
fcbb:bb00:1::2	3.2
fcbb:bb00:7::1	2.6
fcbb:bb00:8::1	2.4
fcbb:bb00:7::2	2.3
fcbb:bb00:1::3	1.6
fcbb:bb00:8::3	0.7

NODES PATH LIST	
vpp1	18.9
vpp8	17.5
vpp4	16.9
vpp3	15.3
vpp6	14.7
vpp2	13.9
vpp5	12.5
vpp7	11.6

TRAFFIC CLASS	
1	13.4
2	10.5
3	7.4

Figure 4.1: Turnilo visualization: bandwidth and avg/max/min delay for all paths in the network.

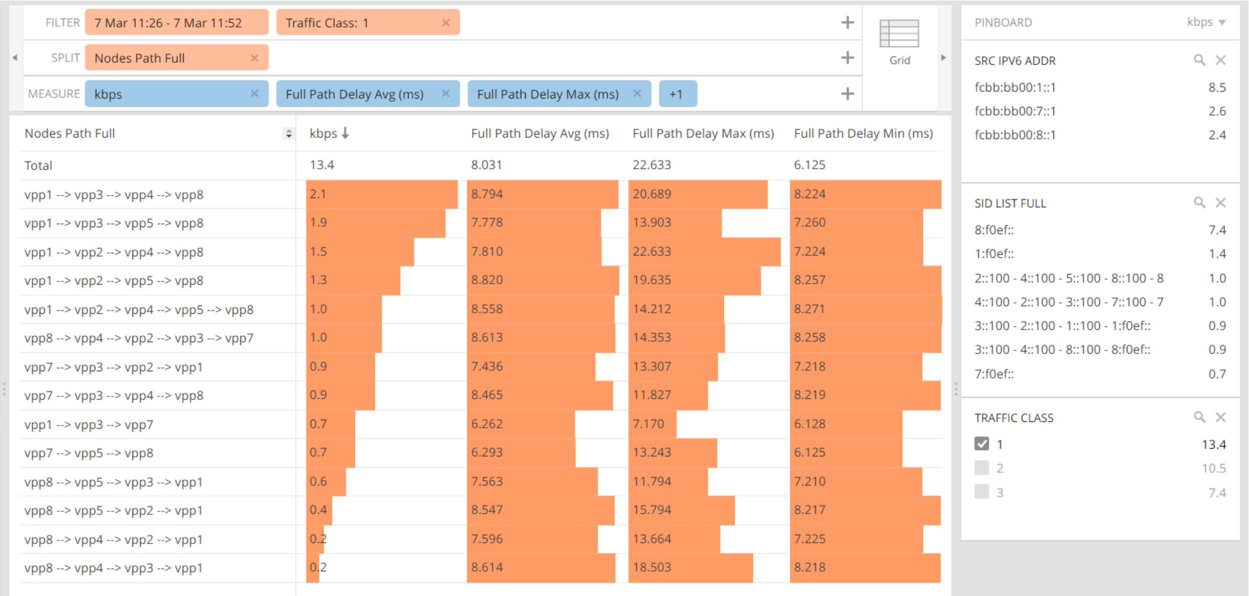
but due to networking or congestion issue some rare probes experience very high delays. This particular case would lead to a reasonable average delay, leading the network operator to think that everything is behaving normally. To overcome such issues, in our pipeline we also measure the maximum observed delay over the observation period. This metric would quickly help identify probes with strangely high values.

We also visualize the minimum value of the delays. This measurement provides an estimate of the delay due to propagation and transmission only. It can serve as an indication of the delay that will likely be experienced when the traversed path is lightly loaded [17]. Another useful metrics that could be implemented is the standard deviation of the delay measurements. This metric would give an indication of how far each measured value distances from the average value. It could provide an easy possibility to implement automated anomaly detection, i.e., by simply setting a treshold on the standard deviation value and triggering an alert if it gets too high. In our pipeline we weren't able to include standard deviation nor variance estimations, since those computation are not supported by the plywood[1] expressions used by Turnilo.

Figure 4.2 displays forwarding paths through the network for customer 1 only, by applying a filter for packets with traffic class 1. On the right side panel we can see all source IPv6 addresses belonging to customer 1, as well as the SRv6 SID lists we are using to steer those packets through the network.

Going one step further, Figure 4.3 focuses on customer 1's packets originating from *fcbb:bb00:1::1* directed towards vpp8, steered through the network via the following SID list: *fcbb:bb00:8::f0ef::*. This means that the packets are to be forwarded in a best effort manner towards vpp8, i.e. following the various ECMP paths existing in the network. This is verified by the visualization in Figure 4.3,

[1] `https://plywood.imply.io/expressions`



FILTER 7 Mar 11:26 - 7 Mar 11:52 | Traffic Class: 1

SPLIT Nodes Path Full

MEASURE kbps | Full Path Delay Avg (ms) | Full Path Delay Max (ms) | +1

Nodes Path Full	kbps ↓	Full Path Delay Avg (ms)	Full Path Delay Max (ms)	Full Path Delay Min (ms)
Total	13.4	8.031	22.633	6.125
vpp1 --> vpp3 --> vpp4 --> vpp8	2.1	8.794	20.689	8.224
vpp1 --> vpp3 --> vpp5 --> vpp8	1.9	7.778	13.903	7.260
vpp1 --> vpp2 --> vpp4 --> vpp8	1.5	7.810	22.633	7.224
vpp1 --> vpp2 --> vpp5 --> vpp8	1.3	8.820	19.635	8.257
vpp1 --> vpp2 --> vpp4 --> vpp5 --> vpp8	1.0	8.558	14.212	8.271
vpp8 --> vpp4 --> vpp2 --> vpp3 --> vpp7	1.0	8.613	14.353	8.258
vpp7 --> vpp3 --> vpp2 --> vpp1	0.9	7.436	13.307	7.218
vpp7 --> vpp3 --> vpp4 --> vpp8	0.9	8.465	11.827	8.219
vpp1 --> vpp3 --> vpp7	0.7	6.262	7.170	6.128
vpp7 --> vpp5 --> vpp8	0.7	6.293	13.243	6.125
vpp8 --> vpp5 --> vpp3 --> vpp1	0.6	7.563	11.794	7.210
vpp8 --> vpp5 --> vpp2 --> vpp1	0.4	8.547	15.794	8.217
vpp8 --> vpp4 --> vpp2 --> vpp1	0.2	7.596	13.664	7.225
vpp8 --> vpp4 --> vpp3 --> vpp1	0.2	8.614	18.503	8.218

PINBOARD kbps

SRC IPV6 ADDR	
fcbb:bb00:1::1	8.5
fcbb:bb00:7::1	2.6
fcbb:bb00:8::1	2.4

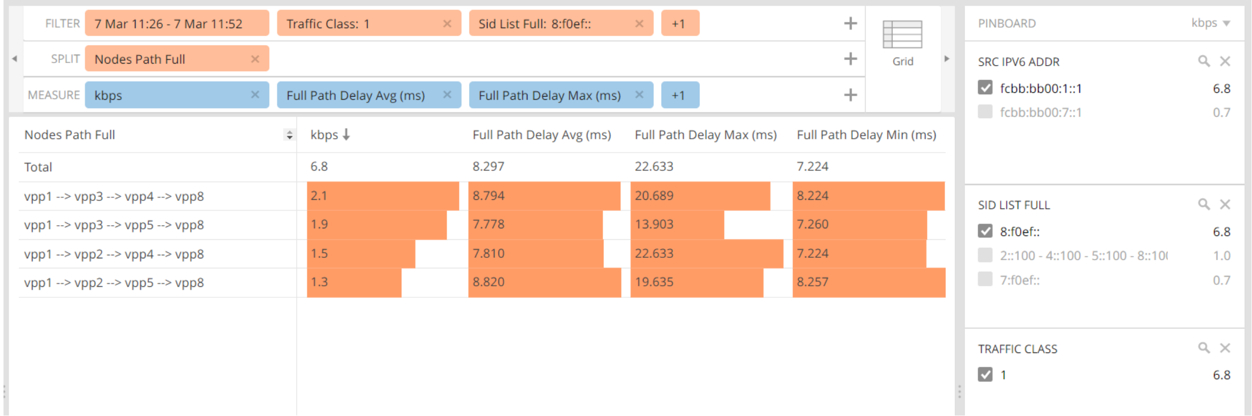
SID LIST FULL	
8:f0ef::	7.4
1:f0ef::	1.4
2::100 - 4::100 - 5::100 - 8::100 - 8	1.0
4::100 - 2::100 - 3::100 - 7::100 - 7	1.0
3::100 - 2::100 - 1::100 - 1:f0ef::	0.9
3::100 - 4::100 - 8::100 - 8:f0ef::	0.9
7:f0ef::	0.7

TRAFFIC CLASS	
☑ 1	13.4
□ 2	10.5
□ 3	7.4

Figure 4.2: Turnilo visualization: bandwidth and avg/max/min delay for customer 1's paths in the network (filtering for tc=1).

clearly showing that the packets are forwarded via four different paths. Here, again, the average, min, and max delay are displayed for each path. As can be seen in Figure 4.3, SIDs in the SID list have been shortened to provide an easier and cleaner visualization.

ECMP path discovery is one of the main scopes for Path Tracing, and is a very important feature for network operators as well, because it allows to visualize all existing forwarding paths in the network. Details on ECMP and how it works are provided in Chapter 3. This is especially useful when planning maintenance or in general any network changes, as operators could know in advance which paths will be selected in case some devices are switched off or some configuration parameters are changed.



FILTER 7 Mar 11:26 - 7 Mar 11:52 | Traffic Class: 1 | Sid List Full: 8:f0ef:: | +1

SPLIT Nodes Path Full

MEASURE kbps | Full Path Delay Avg (ms) | Full Path Delay Max (ms) | +1

Nodes Path Full	kbps ↓	Full Path Delay Avg (ms)	Full Path Delay Max (ms)	Full Path Delay Min (ms)
Total	6.8	8.297	22.633	7.224
vpp1 --> vpp3 --> vpp4 --> vpp8	2.1	8.794	20.689	8.224
vpp1 --> vpp3 --> vpp5 --> vpp8	1.9	7.778	13.903	7.260
vpp1 --> vpp2 --> vpp4 --> vpp8	1.5	7.810	22.633	7.224
vpp1 --> vpp2 --> vpp5 --> vpp8	1.3	8.820	19.635	8.257

PINBOARD kbps

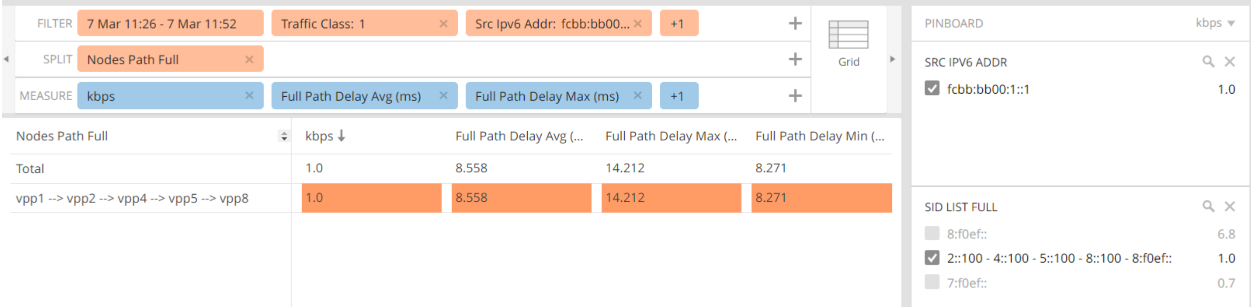
SRC IPV6 ADDR	
☑ fcbb:bb00:1::1	6.8
□ fcbb:bb00:7::1	0.7

SID LIST FULL	
☑ 8:f0ef::	6.8
□ 2::100 - 4::100 - 5::100 - 8::10(	1.0
□ 7:f0ef::	0.7

TRAFFIC CLASS	
☑ 1	6.8

Figure 4.3: Turnilo visualization: bandwidth and avg/max/min delay for customer 1's packets with source address *fcbb:bb00:1::1* and SID list: *fcbb:bb00:8::f0ef::*.

Figure 4.4 visualizes the forwarding path of SRv6 steered packets instead. This is also a very important feature, as it would allow the operators to verify correctness of the SRv6 configuration. Here we are filtering packets still originating from *fcbb:bb00:1::1* and directed towards vpp8, but this time steered via the following SID list: *fcbb:bb00:2::100* - *fcbb:bb00:4::100* - *fcbb:bb00:5::100* - *fcbb:bb008::100* - *fcbb:bb00:8::f0ef::*. This means that the packets should follow the following path: vpp1 - vpp2 - vpp4 - vpp5 - vpp8, which is true and verified by the visualization in Figure 4.4, since this path is the only one available if we filter for this specific SID list.



FILTER: 7 Mar 11:26 - 7 Mar 11:52 | Traffic Class: 1 | Src Ipv6 Addr: fcbb:bb00... | +1

SPLIT: Nodes Path Full

MEASURE: kbps | Full Path Delay Avg (ms) | Full Path Delay Max (ms) | +1

Nodes Path Full	kbps	Full Path Delay Avg (...	Full Path Delay Max (...	Full Path Delay Min (...
Total	1.0	8.558	14.212	8.271
vpp1 --> vpp2 --> vpp4 --> vpp5 --> vpp8	1.0	8.558	14.212	8.271

PINBOARD kbps

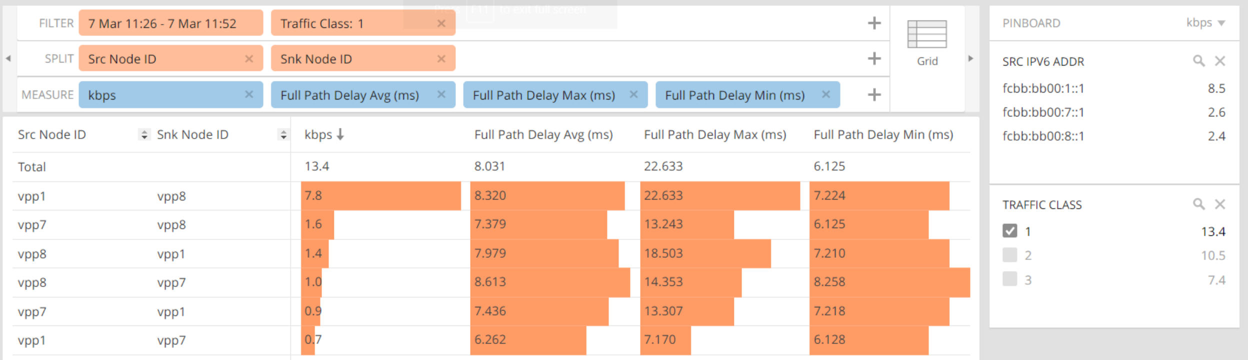
SRC IPV6 ADDR	
☑ fcbb:bb00:1::1	1.0

SID LIST FULL	
□ 8:f0ef::	6.8
☑ 2::100 - 4::100 - 5::100 - 8::100 - 8:f0ef::	1.0
□ 7:f0ef::	0.7

Figure 4.4: Turnilo visualization: bandwidth and avg/max/min delay for customer 1's packets with source address *fcbb:bb00:1::1* and SID list: *fcbb:bb00:2::100* - *fcbb:bb00:4::100* - *fcbb:bb00:5::100* - *fcbb:bb008::100* - *fcbb:bb00:8::f0ef::*.

Customer Exposable Metrics

Another interesting use-case for a Path Tracing visualization pipeline is the possibility to provide business customers with some live telemetry information concerning their packet forwarding through the ISP network. Important here is, of course, that we shouldn't share too much information regarding specific configuration of the ISP's core network and SRv6 policies.



FILTER: 7 Mar 11:26 - 7 Mar 11:52 | Traffic Class: 1

SPLIT: Src Node ID | Snk Node ID

MEASURE: kbps | Full Path Delay Avg (ms) | Full Path Delay Max (ms) | Full Path Delay Min (ms)

Src Node ID	Snk Node ID	kbps	Full Path Delay Avg (ms)	Full Path Delay Max (ms)	Full Path Delay Min (ms)
Total		13.4	8.031	22.633	6.125
vpp1	vpp8	7.8	8.320	22.633	7.224
vpp7	vpp8	1.6	7.379	13.243	6.125
vpp8	vpp1	1.4	7.979	18.503	7.210
vpp8	vpp7	1.0	8.613	14.353	8.258
vpp7	vpp1	0.9	7.436	13.307	7.218
vpp1	vpp7	0.7	6.262	7.170	6.128

PINBOARD kbps

SRC IPV6 ADDR	
fcbb:bb00:1::1	8.5
fcbb:bb00:7::1	2.6
fcbb:bb00:8::1	2.4

TRAFFIC CLASS	
☑ 1	13.4
□ 2	10.5
□ 3	7.4

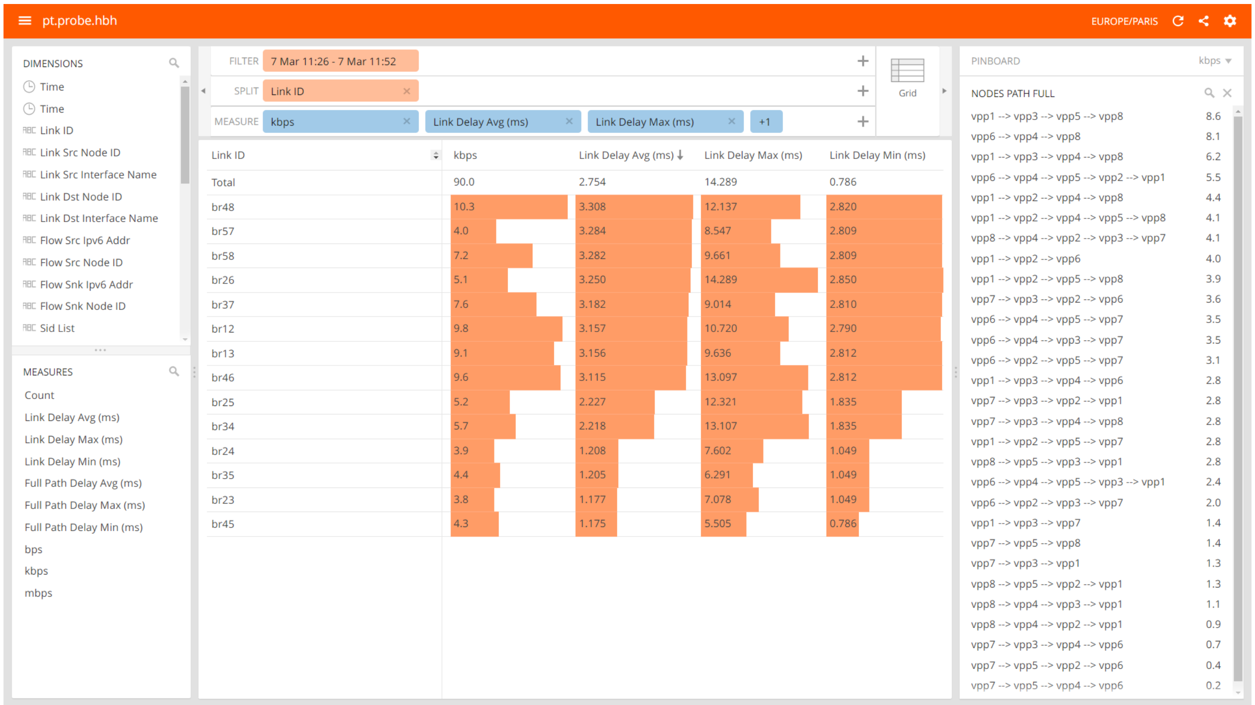
Figure 4.5: Turnilo visualization: bandwidth and avg/max/min delay for customer 1's packets only displaying source and sink node.

Figure 4.5 provides an example of metrics that could be shared with customer 1, displaying only the ISP network end-points, or PE routers. This information would provide the customer with live data on the bandwidth as well as average, max, and min delay of his flows through the

ISP network, providing a quicker way to figure out in the event of problems if high delays are the ISP's fault or not.

4.1.2 Hop-by-hop Visualizations

Now we go one step further down, and focus on hop-by-hop metrics by queriying to the **pt.probe.hbh** datasource. The visualization in Figure 4.6 depicts bandwidth as well as average, max, and min delay for all network's links considering all traffic. On the right side panel we see all forwarding paths in the network. The links are ordered by the average delay, which as we can see is consistent with the delays we artificially applied to the virtual Linux links. Starting from this visualization it would be extremely interesting to map the per-link metrics to a network topology graph which updates on real time. Some insights on this visualization idea are presented in Chapter 5.



FILTER 7 Mar 11:26 - 7 Mar 11:52

SPLIT Link ID

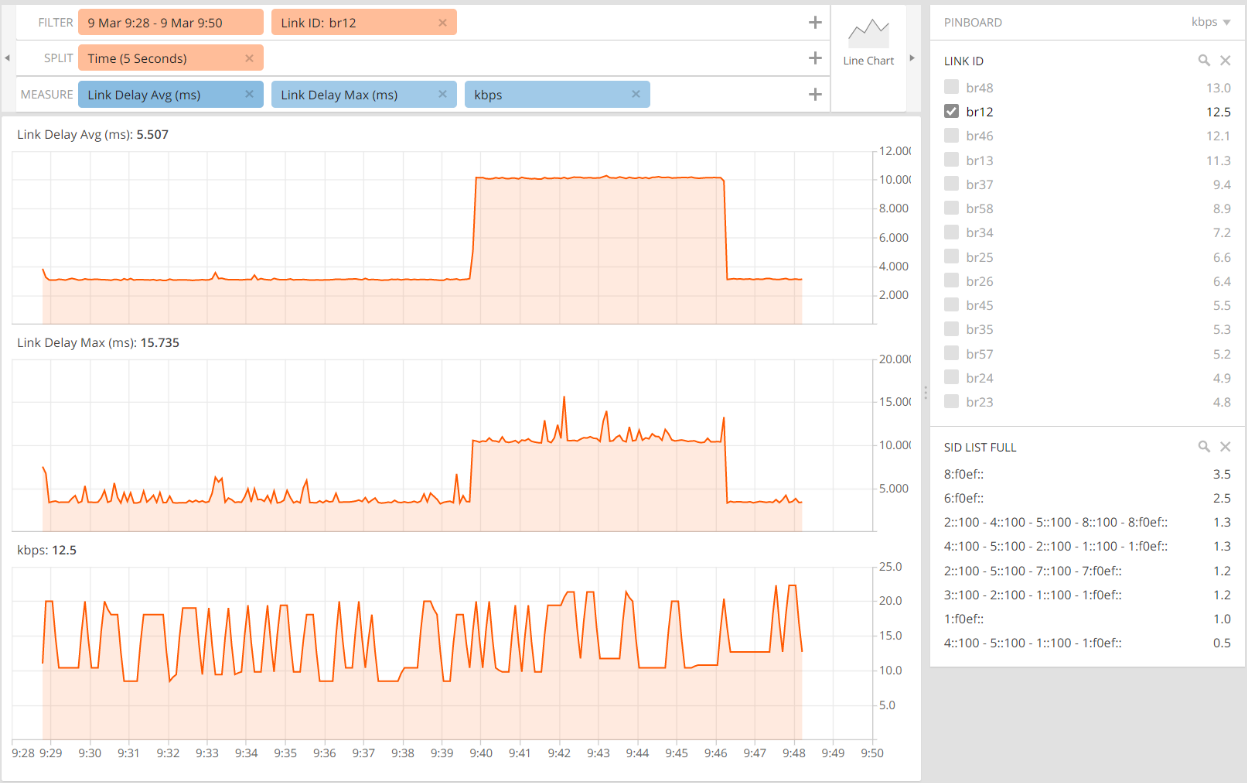
MEASURE kbps | Link Delay Avg (ms) | Link Delay Max (ms) | +1

Link ID	kbps	Link Delay Avg (ms) ↓	Link Delay Max (ms)	Link Delay Min (ms)
Total	90.0	2.754	14.289	0.786
br48	10.3	3.308	12.137	2.820
br57	4.0	3.284	8.547	2.809
br58	7.2	3.282	9.661	2.809
br26	5.1	3.250	14.289	2.850
br37	7.6	3.182	9.014	2.810
br12	9.8	3.157	10.720	2.790
br13	9.1	3.156	9.636	2.812
br46	9.6	3.115	13.097	2.812
br25	5.2	2.227	12.321	1.835
br34	5.7	2.218	13.107	1.835
br24	3.9	1.208	7.602	1.049
br35	4.4	1.205	6.291	1.049
br23	3.8	1.177	7.078	1.049
br45	4.3	1.175	5.505	0.786

NODES PATH FULL	
vpp1 --> vpp3 --> vpp5 --> vpp8	8.6
vpp6 --> vpp4 --> vpp8	8.1
vpp1 --> vpp3 --> vpp4 --> vpp8	6.2
vpp6 --> vpp4 --> vpp5 --> vpp2 --> vpp1	5.5
vpp1 --> vpp2 --> vpp4 --> vpp8	4.4
vpp1 --> vpp2 --> vpp4 --> vpp5 --> vpp8	4.1
vpp8 --> vpp4 --> vpp2 --> vpp3 --> vpp7	4.1
vpp1 --> vpp2 --> vpp6	4.0
vpp1 --> vpp2 --> vpp5 --> vpp8	3.9
vpp7 --> vpp3 --> vpp2 --> vpp6	3.6
vpp6 --> vpp4 --> vpp5 --> vpp7	3.5
vpp6 --> vpp4 --> vpp3 --> vpp7	3.5
vpp6 --> vpp2 --> vpp5 --> vpp7	3.1
vpp1 --> vpp3 --> vpp4 --> vpp6	2.8
vpp7 --> vpp3 --> vpp2 --> vpp1	2.8
vpp7 --> vpp3 --> vpp4 --> vpp8	2.8
vpp1 --> vpp2 --> vpp5 --> vpp7	2.8
vpp8 --> vpp5 --> vpp3 --> vpp1	2.8
vpp6 --> vpp4 --> vpp5 --> vpp3 --> vpp1	2.4
vpp6 --> vpp2 --> vpp3 --> vpp7	2.0
vpp1 --> vpp3 --> vpp7	1.4
vpp7 --> vpp5 --> vpp8	1.4
vpp7 --> vpp3 --> vpp1	1.3
vpp8 --> vpp5 --> vpp2 --> vpp1	1.3
vpp8 --> vpp4 --> vpp3 --> vpp1	1.1
vpp8 --> vpp4 --> vpp2 --> vpp1	0.9
vpp7 --> vpp3 --> vpp4 --> vpp6	0.7
vpp7 --> vpp5 --> vpp2 --> vpp6	0.4
vpp7 --> vpp5 --> vpp4 --> vpp6	0.2

Figure 4.6: Turnilo visualization: bandwidth and avg/max/min delay for all links in the ISP network.

This type of per-hop visualization is very important from the perspective of a network operator, as it gives visibility on all the links and thus makes it easier to identify a problem with one of the links on real time. Figure 4.7 shows a time split focusing on link *br12* connecting vpp1 and vpp2. Around 9h40 we have artificially changed the link delay from 3ms to 10ms, and this change is immediately detected on Turnilo. Around 9h46 we have reset the delay back to its default value of 3ms. This is a good example on how the visualizations enabled by this pipeline can be used to detect congestion on a link in real time.



FILTER 9 Mar 9:28 - 9 Mar 9:50 × Link ID: br12 ×

SPLIT Time (5 Seconds) ×

MEASURE Link Delay Avg (ms) × Link Delay Max (ms) × kbps ×

Link Delay Avg (ms): **5.507**

Link Delay Max (ms): **15.735**

kbps: **12.5**

PINBOARD kbps

LINK ID	
□ br48	13.0
☑ br12	12.5
□ br46	12.1
□ br13	11.3
□ br37	9.4
□ br58	8.9
□ br34	7.2
□ br25	6.6
□ br26	6.4
□ br45	5.5
□ br35	5.3
□ br57	5.2
□ br24	4.9
□ br23	4.8

SID LIST FULL	
8:f0ef::	3.5
6:f0ef::	2.5
2::100 - 4::100 - 5::100 - 8::100 - 8:f0ef::	1.3
4::100 - 5::100 - 2::100 - 1::100 - 1:f0ef::	1.3
2::100 - 5::100 - 7::100 - 7:f0ef::	1.2
3::100 - 2::100 - 1::100 - 1:f0ef::	1.2
1:f0ef::	1.0
4::100 - 5::100 - 1::100 - 1:f0ef::	0.5

Figure 4.7: Turnilo visualization: time split of link br12 visualizing bandwidth, average and maximum delay.

4.2 Postcard-based Pipeline

This Section evaluates our alternative pipeline, which was developed mainly as a *proof of concept* to show what is achievable with Path Tracing and the postcard export approach. Initially, we describe the main issues and limitations we encountered when designing the pipeline, then we validate it by comparing it with the main pipeline's results.

4.2.1 Issues and Limitations

As briefly introduced in Chapter 3, the design of this postcard-based pipeline had to overcome some limitations, mainly related to the IPFIX implementation in VPP. In fact, when packets are encapsulated with an SRv6 header, they are not appearing in the IPFIX exports. This is probably due to the fact that SRv6 was introduced later in the VPP platform, and the IPFIX functionality was not verified to be fully compatible alongside it.

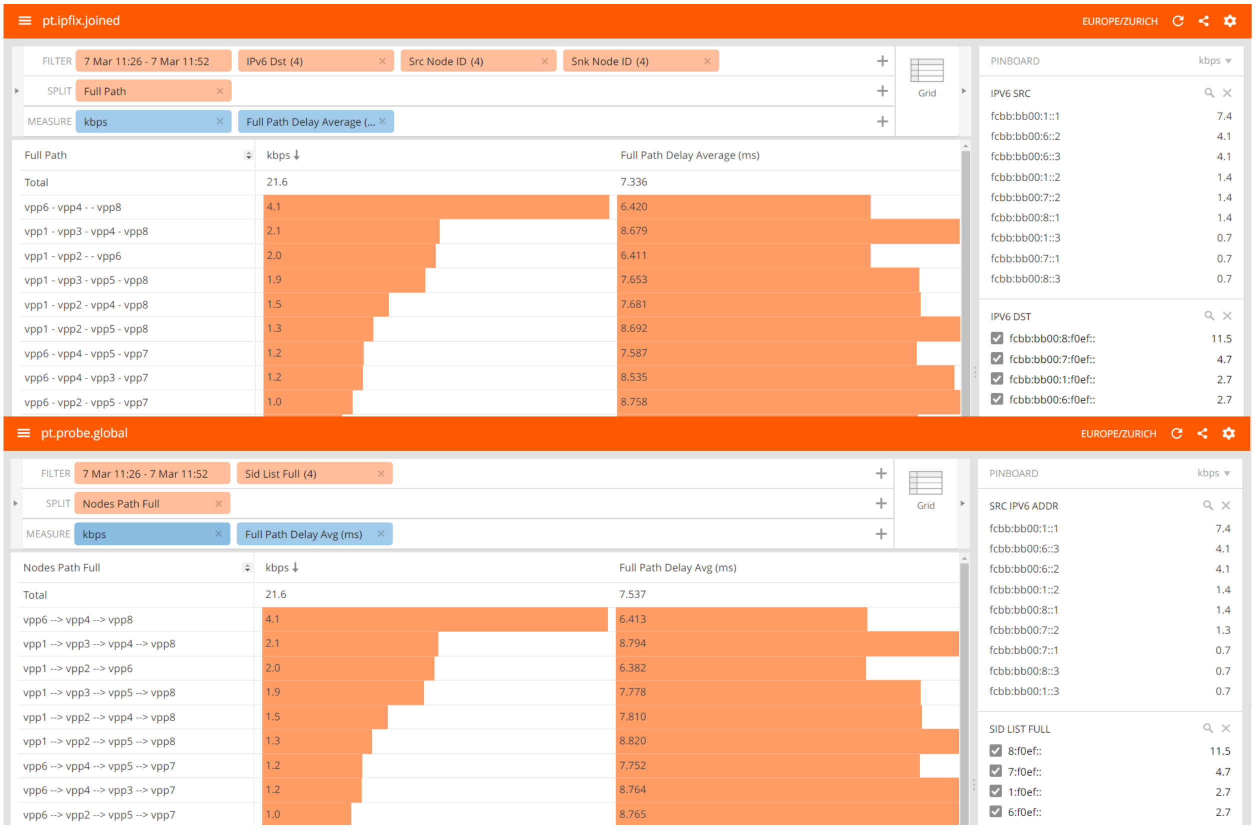
Another shortcoming is that we are able to reconstruct the forwarding path only for non SRv6 steered packets, where the destination IPv6 address doesn't change throughout the network. The reason behind this is that we don't receive SRv6 specific information from IPFIX exports. In an SRv6 steered packet, the destination IPv6 address changes at every hop, and without the SID list to correlate on, we have no way to reconstruct the path in this case. IETF draft [15] proposes new IPFIX elements to export metrics in the SRH, such as the full SID list (entity *ipv6SRHSegmentBasicList*). This would solve our issue, and enable path reconstruction for SRv6 steered paths as well.

Another issue was related to getting hop-by-hop as well as full-path delay measurements. Since Path Tracing doesn't support a postcard mode export approach yet, we don't have any delay information exported by IPFIX. This aspect is also discussed in Chapter 5. To overcome this issue in our pipeline, we are querying from the pt.probe.hbh datasource (which is populated by the main pipeline) in Druid. With this approach we are enriching IPFIX exports with average, max and min delay for the link over the sampling period. This method provides an estimate of the delay, which for our use case is very reasonable, but is not an exact measurement. The reason is that all flows going through that link on the sampling interval are included for the delay computation, and not only the specific flow which is being exported.

4.2.2 Pipeline Validation

In order to validate our alternative pipeline, we compared resulting metrics against our main passport-based pipeline. In Figure 4.8 we provide a visualization of all forwarding paths for non SRv6 steered packets, queried from both the main pipeline and the alternative postcard-based pipeline. We can see that all paths are correctly identified also from the alternative pipeline, with correct bandwidth measurements. The delays are not exactly equal for the reason explained above in this Section, but anyway very consistent.

Figure 4.9 provides time splits of bandwidth and average delay for both IPFIX and main pipeline. Also here we see that the results are consistent. The only difference is that with the IPFIX pipeline we don't have the same level of fine grained precision on the bandwidth measurements, and this is because the data arrives in ”chunks” at regular 30s intervals. With the passport-based pipeline, the telemetry data is processed as soon as the packets arrive at the sink node, which leads in having almost no delay between packet forwarding and telemetry data visualization.



pt.ipfix.joined

EUROPE/ZURICH

FILTER: 7 Mar 11:26 - 7 Mar 11:52 | IPv6 Dst (4) | Src Node ID (4) | Snk Node ID (4)

SPLIT: Full Path

MEASURE: kbps | Full Path Delay Average (...

Full Path	kbps ↓	Full Path Delay Average (ms)
Total	21.6	7.336
vpp6 - vpp4 - - vpp8	4.1	6.420
vpp1 - vpp3 - vpp4 - vpp8	2.1	8.679
vpp1 - vpp2 - - vpp6	2.0	6.411
vpp1 - vpp3 - vpp5 - vpp8	1.9	7.653
vpp1 - vpp2 - vpp4 - vpp8	1.5	7.681
vpp1 - vpp2 - vpp5 - vpp8	1.3	8.692
vpp6 - vpp4 - vpp5 - vpp7	1.2	7.587
vpp6 - vpp4 - vpp3 - vpp7	1.2	8.535
vpp6 - vpp2 - vpp5 - vpp7	1.0	8.758

PINBOARD kbps

IPV6 SRC	
fcbb:bb00:1::1	7.4
fcbb:bb00:6::2	4.1
fcbb:bb00:6::3	4.1
fcbb:bb00:1::2	1.4
fcbb:bb00:7::2	1.4
fcbb:bb00:8::1	1.4
fcbb:bb00:1::3	0.7
fcbb:bb00:7::1	0.7
fcbb:bb00:8::3	0.7

IPV6 DST	
☑ fcbb:bb00:8:f0ef::	11.5
☑ fcbb:bb00:7:f0ef::	4.7
☑ fcbb:bb00:1:f0ef::	2.7
☑ fcbb:bb00:6:f0ef::	2.7

pt.probe.global

EUROPE/ZURICH

FILTER: 7 Mar 11:26 - 7 Mar 11:52 | Sid List Full (4)

SPLIT: Nodes Path Full

MEASURE: kbps | Full Path Delay Avg (ms)

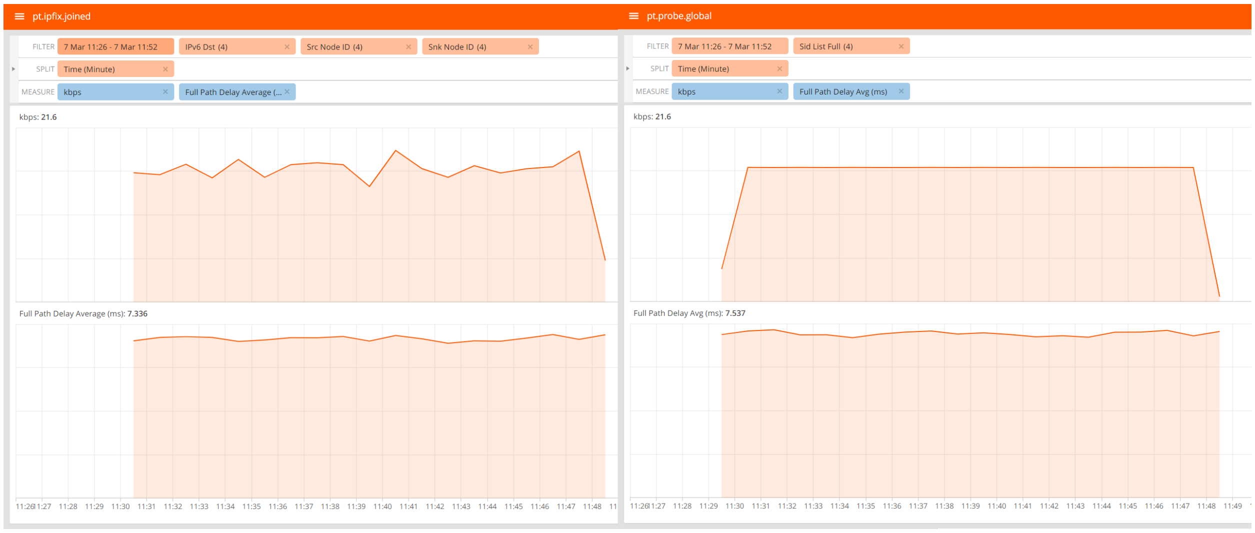
Nodes Path Full	kbps ↓	Full Path Delay Avg (ms)
Total	21.6	7.537
vpp6 --> vpp4 --> vpp8	4.1	6.413
vpp1 --> vpp3 --> vpp4 --> vpp8	2.1	8.794
vpp1 --> vpp2 --> vpp6	2.0	6.382
vpp1 --> vpp3 --> vpp5 --> vpp8	1.9	7.778
vpp1 --> vpp2 --> vpp4 --> vpp8	1.5	7.810
vpp1 --> vpp2 --> vpp5 --> vpp8	1.3	8.820
vpp6 --> vpp4 --> vpp5 --> vpp7	1.2	7.752
vpp6 --> vpp4 --> vpp3 --> vpp7	1.2	8.764
vpp6 --> vpp2 --> vpp5 --> vpp7	1.0	8.765

PINBOARD kbps

SRC IPV6 ADDR	
fcbb:bb00:1::1	7.4
fcbb:bb00:6::3	4.1
fcbb:bb00:6::2	4.1
fcbb:bb00:1::2	1.4
fcbb:bb00:8::1	1.4
fcbb:bb00:7::2	1.3
fcbb:bb00:7::1	0.7
fcbb:bb00:8::3	0.7
fcbb:bb00:1::3	0.7

SID LIST FULL	
☑ 8:f0ef::	11.5
☑ 7:f0ef::	4.7
☑ 1:f0ef::	2.7
☑ 6:f0ef::	2.7

Figure 4.8: Turnilo visualizations. Above: full-path of non SRv6 steered packets (i.e. best effort through the network) with bandwidth and average delay, queried from the IPFIX-based pipeline. Below: same visualization from above, but queried from the main pipeline.



pt.ipfix.joined

FILTER 7 Mar 11:26 - 7 Mar 11:52 | IPv6 Dst (4) | Src Node ID (4) | Snk Node ID (4)

SPLIT Time (Minute)

MEASURE kbps | Full Path Delay Average (...

kbps: 21.6

Full Path Delay Average (ms): 7.336

pt.probe.global

FILTER 7 Mar 11:26 - 7 Mar 11:52 | Sid List Full (4)

SPLIT Time (Minute)

MEASURE kbps | Full Path Delay Avg (ms)

kbps: 21.6

Full Path Delay Avg (ms): 7.537

Figure 4.9: Turnilo visualizations. Left: bandwidth and average delay of all non SRv6 steered paths in the network queried from the alternative pipeline. Right: same visualization, but queried from the main pipeline.

4.3 Discussion and Comparison

4.3.1 Path Tracing Validation

As briefly mentioned in Section 4.1.2, the delay measurements are very consistent with the artificial delays which we have introduced in the virtual Linux network using the *tc*[2] utility. The visualization in Figure 4.7 validates the measurements further, given that a live change of a link's delay is instantly reflected in Turnilo as expected.

On the other hand, bandwidth and forwarding path reconstruction are validated by the IPFIX pipeline. In fact, these two dimensions are computed in two independent ways in the two pipelines. The bandwidth is reconstructed thanks to the *payload_length* field (in bytes) in the Path Tracing messages, and thanks to the *bytes* field in IPFIX messages. The bandwidth measurements, as also previously shown in Figures 4.8 and 4.9, are very consistent between the two pipelines, which validates the result. The same reasoning is valid for the forwarding path reconstruction, which is performed in two different manners by the two pipelines. In the main pipeline, the information is produced by the network when the packet is forwarded, and at the collector we receive a packet with full path information included. In the IPFIX pipeline, we perform correlation matching on inbound, outbound interfaces and IP addresses as explained in Chapter 3.

4.3.2 Evaluation and Comparison of the two Pipelines

First, it is worth mentioning that there isn't an approach which is clearly better than the other. Both pipelines enable the same level of visibility on the network, the difference is on how the information is correlated and aggregated. The choice on which approach is better suited highly depends on the specific use case, and it mostly comes down to flexibility and scalablility reasons. The advantages that a passport-based pipeline has over a postcard-based pipeline can be summarized as follows:

- **Visualization Delay**: telemetry data reaches the database and thus is queryable by the visualization tool with an extremely low delay, that is only composed by the processing, druid ingestion and query time. We don't have an aggregation period in the minute range before data is exported, like is the case for IPFIX.
- **Forwarding path reconstruction**: the forwarding path is already included in the probe packet when it arrives at the probe collector, and thus no further correlation is required at the database level in order to get this information.

The main problem with the passport-based approach is that it cannot be scaled up too much, and it is most likely not feasible to implement with a passive telemetry approach alongside it, at least if we require to correlate the Path Tracing information with other protocols. The reason is that network-wide correlation is very expensive, since it requires a mapping of the whole network that needs to be queried at the database level for all Path Tracing packets, and this doesn't scale well. Also, if we were to append Path Tracing metadata on all customer packets (i.e. passive telemetry), only a very low export rate would be realistic in passport mode. It is not reasonable to export telemetry data proportional to an ISP's network throughput. For such a use case, the only solution is probably a postcard-based pipeline. The advantages of a postcard-based pipeline can be summarized as follows:

[2] `https://man7.org/linux/man-pages/man8/tc.8.html`

- **Data Aggregation**: telemetry data is only exported at regular intervals and a single export contains aggregated measures for multiple packets of a flow. This highly reduces the computation to be performed at the database level, thus increasing scalability.

- **Local Correlation**: with a telemetry data collector such as *pmacct*, it is possible to correlate per-node information with other protocols, without the need of doing it globally for the full network at the database level. This also increases scalability.

Chapter 5

Outlook

This Chapter introduces suggestions for possible future work on our pipeline. We also provide ideas on further development of the Path Tracing protocol.

5.1 Live Network Topology Graph Visualization

A useful addition to the visualizations illustrated in Chapter 4 could be a *live network topology*, i.e. a network graph showing per-hop metrics, such as bandwidth and delay directly on the links. This way instead of listing link measures, like in the visualization from Figure 4.6, we would append the metrics to each link, giving us a cleaner overall picture. We could even have a color code to help visually identify issues, for instance when the delay is way too high than normal. Figure 5.1 provides an example on how this could look like, displaying the average delay and standard deviation for each link. For now we couldn't implement it live, since Turnilo doesn't support tailor-made external visualization structures.

5.2 Path Tracing Features Extension

The Path Tracing project is currently only designed as an active telemetry protocol, supporting generation of probe packets with in-band telemetry data. It also only supports a passport-based data export approach. The protocol could be extended to support more functionality, so that it would fit more use cases, integrate better in existing telemetry frameworks, and thus in general granting more flexibility to network operators as to how they can configure it.

As previously mentioned throughout this Thesis, the protocol could be extended by allowing a postcard-based export approach. More concretely, IPFIX could be used to aggregate average, maximum, and minimum delay along with the other IPFIX entities, before exporting data to the collector. Referring to Path Tracing protocol specification, this would require very small modifications. In fact, the same Path Tracing SRH TLV could be used to forward source timestamp information along the network. Then each router would compute the delay of forwarding from source to current node and store it in the IPFIX cache, average the measurements, identify maximum and minimum delays and export the metrics when flushing the cache (standard IPFIX behaviour). The only difference is that the IPv6 HBH option is now not required anymore, since metrics are exported at each hop.

Another interesting possibility would be to extend the protocol to support passive in-band telemetry as well, and not only active probe generation. This means that in-band telemetry data would be available on all customer packets as well, and not only on probe packets. Concretely, this

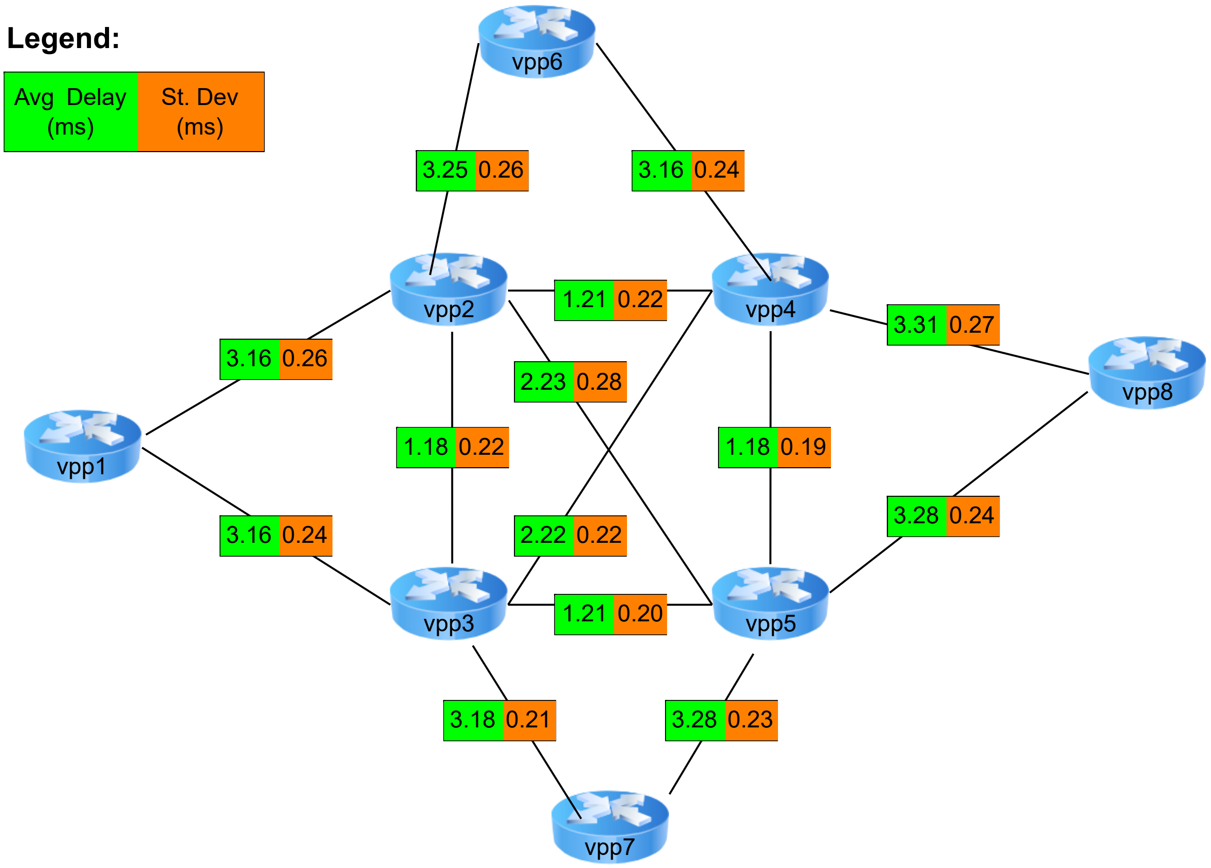


Figure 5.1: Example of possible topology visualization with live metrics update. For each link, average delay and standard deviation of the delay measurements are displayed. Measurements for the example are taken from the same test scenario from Figure 4.6.

means that when packets are encapsulated with the SRH based on a policy match, the Path Tracing TLV would be added as well. This makes sense only with a postcard-based export, otherwise it wouldn't scale.

The combination of passive and active in-band telemetry, as explained in Chapter 2 would guarantee monitoring capabilities for many possible scenarios. Passive telemetry can be used to accurately measure delay and forwarding paths of customer's packets through the ISP network, whereas active telemetry can be deployed for path discovery, backup path testing and preventing disruptions during maintenance.

Chapter 6

Summary

Network visibility is increasingly important within today's large scale ISP networks. With the Internet becoming more and more an essential need for today's society, networking failures and congestion need to be identified and solved as quickly as possible, preferably in an automated manner. The goal of constructing a fully independent network operating in a closed loop which can automatically apply configuration changes to react upon failures is only achievable together with a deep level of visibility on the network itself.

In this Thesis we designed and evaluated two Network Telemetry visualization pipelines based on the Path Tracing [10] protocol. Path Tracing is an in-band telemetry protocol, which enables generation of probe packets with telemetry metadata through an SRv6 network. The pipelines' purpose is to provide hop-by-hop and full-path delays as well as forwarding path visibility for packets traversing the network. These metrics are very important because they allow to verify the quality of the connections and detect congestion or failures in the network. Together with the capabilities of an SRv6 dataplane, which means having the possibility to steer packets from the source, such pipelines also provide a mean to test backup or non-used paths in the network. Visibility on the active paths is important to evaluate the network's current status and to detect issues. The possibility to test all other paths which are not being used is an extremely useful feature as it allows operators to forecast the future state of the network, for example when planning a maintenance window or estimating a failure scenario.

The pipelines we designed differ from each other based on the data export approach used. The first pipeline uses the passport approach, natively supported by the Path Tracing protocol, which consists in exporting telemetry information relative to the full forwarding path only at the network's egress point. The second pipeline exports data via the IPFIX protocol, thus uses the postcard approach. The latter refers to exposing locally aggregated telemetry data at each node along the forwarding path.

On one hand we found that the passport based pipeline is faster in terms of visualization delay, and provides an easy way to achieve forwarding path visibility. The reason is that the full-path information received at the collector already contains path information. We can say that the network automatically performs correlation for us. On the other hand, we need to consider scalability limitations. In fact, for the passport based pipeline, heavy post processing of the telemetry data with full network information is required in order to give meaning to the metrics. This highly limits the amount of probes that can be generated. The postcard based pipeline addresses the scalability issues thanks to local aggregation and correlation, limiting the amount of data being exported. The disadvantage with this approach is that the forwarding path now needs to be reconstructed at the database level.

Bibliography

[1] Aitken, P., Claise, B., and Trammell, B. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, Sept. 2013. Available at https://www.rfc-editor.org/info/rfc7011.

[2] Ali, Z., Gandhi, R., Filsfils, C., Brockners, F., Nainar, N. K., Pignataro, C., Li, C., Chen, M., and Dawra, G. Segment Routing Header encapsulation for In-situ OAM Data. Internet-Draft draft-ali-spring-ioam-srv6-05, Internet Engineering Task Force, Jan. 2022. Work in Progress. Available at https://datatracker.ietf.org/doc/html/draft-ali-spring-ioam-srv6-05.

[3] Babiarz, J., Krzanowski, R. M., Hedayat, K., Yum, K., and Morton, A. A Two-Way Active Measurement Protocol (TWAMP). RFC 5357, Oct. 2008. Available at https://datatracker.ietf.org/doc/html/rfc5357.

[4] Bhandari, S., and Brockners, F. In-situ OAM IPv6 Options. Internet-Draft draft-ietf-ippm-ioam-ipv6-options-07, Internet Engineering Task Force, Feb. 2022. Work in Progress. Available at https://datatracker.ietf.org/doc/html/draft-ietf-ippm-ioam-ipv6-options-07.

[5] Brockners, F., Bhandari, S., Bernier, D., and Mizrahi, T. In-situ OAM Deployment. Internet-Draft draft-ietf-ippm-ioam-deployment-00, Internet Engineering Task Force, Oct. 2021. Work in Progress. Available at https://datatracker.ietf.org/doc/html/draft-ietf-ippm-ioam-deployment/.

[6] Brockners, F., Bhandari, S., and Mizrahi, T. Data Fields for In-situ OAM. Internet-Draft draft-ietf-ippm-ioam-data-17, Internet Engineering Task Force, Dec. 2021. Work in Progress. Available at https://datatracker.ietf.org/doc/html/draft-ietf-ippm-ioam-data.

[7] Cheng, W., Filsfils, C., Li, Z., Decraene, B., Cai, D., Voyer, D., Clad, F., Zadok, S., Guichard, J., Liu, A., Raszuk, R., and Li, C. Compressed SRv6 Segment List Encoding in SRH. Internet-Draft draft-ietf-spring-srv6-srh-compression-01, Internet Engineering Task Force, Mar. 2022. Work in Progress. Available at https://datatracker.ietf.org/doc/html/draft-ietf-spring-srv6-srh-compression-01.

[8] Clemm, A., and Voit, E. Subscription to YANG Notifications for Datastore Updates. RFC 8641, Sept. 2019. Available at https://www.rfc-editor.org/info/rfc8641.

[9] Fedor, M., Schoffstall, M. L., Davin, J. R., and Case, D. J. D. Simple Network Management Protocol (SNMP). RFC 1157, May 1990. Available at https://www.rfc-editor.org/rfc/rfc1157.txt.

[10] Filsfils, C., Abdelsalam, A., Camarillo, P., Yufit, M., Graf, T., Su, Y., and Matsushima, S. Path Tracing in SRv6 networks. Internet-Draft draft-filsfils-spring-path-tracing-00, Internet Engineering Task Force, Mar. 2022. Work in Progress. Available at `https://datatracker.ietf.org/doc/html/draft-filsfils-spring-path-tracing`.

[11] Filsfils, C., Camarillo, P., Leddy, J., Voyer, D., Matsushima, S., and Li, Z. Segment Routing over IPv6 (SRv6) Network Programming. RFC 8986, Feb. 2021. Available at `https://www.rfc-editor.org/info/rfc8986`.

[12] Filsfils, C., Dukes, D., Previdi, S., Leddy, J., Matsushima, S., and Voyer, D. IPv6 Segment Routing Header (SRH). RFC 8754, Mar. 2020. Available at `https://www.rfc-editor.org/info/rfc8754`.

[13] Filsfils, C., Previdi, S., Ginsberg, L., Decraene, B., Litkowski, S., and Shakir, R. Segment Routing Architecture. RFC 8402, July 2018. Available at `https://www.rfc-editor.org/info/rfc8402`.

[14] Gandhi, R., Ali, Z., Brockners, F., Wen, B., Decraene, B., and Kozak, V. MPLS Data Plane Encapsulation for In-situ OAM Data. Internet-Draft draft-gandhi-mpls-ioam-03, Internet Engineering Task Force, Feb. 2022. Work in Progress. Available at `https://datatracker.ietf.org/doc/html/draft-gandhi-mpls-ioam`.

[15] Graf, T., and Claise, B. Export of Segment Routing IPv6 Information in IP Flow Information Export (IPFIX). Internet-Draft draft-tgraf-opsawg-ipfix-srv6-srh-02, Internet Engineering Task Force, Mar. 2022. Work in Progress. Available at `https://datatracker.ietf.org/doc/html/draft-tgraf-opsawg-ipfix-srv6-srh-02`.

[16] Iurman, J., Donnet, B., and Brockners, F. Implementation of ipv6 ioam in linux kernel, Aug. 2020.

[17] Kalidindi, S., Zekauskas, M. J., and Almes, D. G. T. A One-way Delay Metric for IPPM. RFC 2679, Sept. 1999. Available at `https://datatracker.ietf.org/doc/html/rfc2679`.

[18] Mizrahi, T., Fabini, J., and Morton, A. Guidelines for Defining Packet Timestamps. RFC 8877, Sept. 2020. Available at `https://datatracker.ietf.org/doc/html/rfc8877`.

[19] Rekhter, Y., Hares, S., and Li, T. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Jan. 2006. Available at `https://www.rfc-editor.org/info/rfc4271`.

[20] Scudder, J., Fernando, R., and Stuart, S. BGP Monitoring Protocol (BMP). RFC 7854, June 2016. Available at `https://www.rfc-editor.org/info/rfc7854`.

[21] Song, H., Gafni, B., Zhou, T., Li, Z., Brockners, F., Bhandari, S., Sivakolundu, R., and Mizrahi, T. In-situ OAM Direct Exporting. Internet-Draft draft-ietf-ippm-ioam-direct-export-07, Internet Engineering Task Force, Oct. 2021. Work in Progress. Available at `https://datatracker.ietf.org/doc/html/draft-ietf-ippm-ioam-direct-export-07`.

[22] Song, H., Mirsky, G., Filsfils, C., Abdelsalam, A., Zhou, T., Li, Z., Shin, J., and Lee, K. In-Situ OAM Marking-based Direct Export. Internet-Draft draft-song-ippm-postcard-based-telemetry-11, Internet Engineering Task Force, Nov. 2021. Work in Progress. Available at `https://datatracker.ietf.org/doc/html/draft-song-ippm-postcard-based-telemetry-11`.

[23] Song, H., Qin, F., Martinez-Julia, P., Ciavaglia, L., and Wang, A. Network Telemetry Framework. Internet-Draft draft-ietf-opsawg-ntf-13, Internet Engineering Task Force, Dec. 2021. Work in Progress. Available at https://datatracker.ietf.org/doc/html/draft-ietf-opsawg-ntf-13.

[24] Tan, L., Su, W., Zhang, Z., Miao, J., Liu, X., and Li, N. In-band network telemetry: A survey, Aug. 2020.

[25] Trammell, B., Wagner, A., and Claise, B. Flow Aggregation for the IP Flow Information Export (IPFIX) Protocol. RFC 7015, Sept. 2013. Available at https://datatracker.ietf.org/doc/html/rfc7015.

[26] What will happen when the routing table hits 1024k? https://blog.apnic.net/2021/03/03/what-will-happen-when-the-routing-table-hits-1024k/. [Accessed February-2022].

[27] Druid. https://druid.apache.org/docs/latest/design/index.html. [Accessed March-2022].

[28] Ip flow information export (ipfix) entities. https://www.iana.org/assignments/ipfix/ipfix.xhtml. [Accessed February-2022].

[29] Interior gatewas protocol. https://en.wikipedia.org/wiki/Interior_gateway_protocol. [Accessed February-2022].

[30] Typical isp network architecture. https://www.packetnetworking.com/typical-isp-network-architecture/. [Accessed February-2022].

[31] Isp network design. http://blog.rapidlinksnetworks.co.uk/isp-network-design/. [Accessed February-2022].

[32] Isp network design (presentation). https://au.int/sites/default/files/documents/31363-doc-session_8-1-_isp-network-design.pdf. [Accessed February-2022].

[33] Layer 3 vpn (l3vpn). https://www.techopedia.com/definition/30757/layer-3-vpn-l3vpn. [Accessed February-2022].

[34] Multiprotocol label switching (mpls). https://en.wikipedia.org/wiki/Multiprotocol_Label_Switching. [Accessed February-2022].

[35] In-band network telemetry (int) dataplane specification. https://p4.org/p4-spec/docs/INT_v2_1.pdf. [Accessed February-2022].

[36] Pmacct project website. http://www.pmacct.net/. [Accessed February-2022].

[37] Next-gen network telemetry is within your packets: In-band oam (cisco live 2018). https://www.ciscolive.com/c/dam/r/ciscolive/emea/docs/2018/pdf/BRKSDN-2901.pdf. [Accessed February-2022].

[38] Swisscom network analytics visibility for a closed loop operated network. https://www.swinog.ch/wp-content/uploads/2021/12/Thomas-Graf-and-Marco-Tollini-Swisscom-Network-Analytics-with-BMP-IPFIX-and-YANG-Push.pdf. [Accessed February-2022].

[39] Segment routing. https://support.huawei.com/enterprise/en/doc/EDOC1100092117. [Accessed February-2022].

[40] What is segment routing? https://www.juniper.net/us/en/research-topics/what-is-segment-routing.html. [Accessed February-2022].

[41] Segment routing. https://en.wikipedia.org/wiki/Segment_routing. [Accessed February-2022].

[42] What is srv6? https://info.support.huawei.com/info-finder/encyclopedia/en/SRv6.html. [Accessed February-2022].

[43] Network telemetry. https://info.support.huawei.com/info-finder/encyclopedia/en/Telemetry.html. [Accessed February-2022].

[44] Type-length-value. https://en.wikipedia.org/wiki/Type%E2%80%93length%E2%80%93value. [Accessed February-2022].

[45] Turnilo. https://github.com/allegro/turnilo. [Accessed March-2022].

[46] Adapting turnilo to sql backend? https://github.com/allegro/turnilo/issues/674. [Accessed March-2022].

[47] What is vpp? https://wiki.fd.io/view/VPP/What_is_VPP. [Accessed February-2022].

[48] Vpp inband oam (ioam). https://docs.fd.io/vpp/17.04/ioam_plugin_doc.html. [Accessed February-2022].

[49] Vpp technology. https://fd.io/gettingstarted/technology/. [Accessed February-2022].

[50] Scalar vs vector packet processing. https://fd.io/docs/vpp/v2101/whatisvpp/scalar-vs-vector-packet-processing.html. [Accessed February-2022].

[51] Virtual routing and forwarding. https://en.wikipedia.org/wiki/Virtual_routing_and_forwarding. [Accessed February-2022].

[52] Equal-cost multi-path routing. https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing. [Accessed March-2022].

[53] Viswanathan, A., Rosen, E. C., and Callon, R. Multiprotocol Label Switching Architecture. RFC 3031, Jan. 2001. Available at https://www.rfc-editor.org/info/rfc3031.

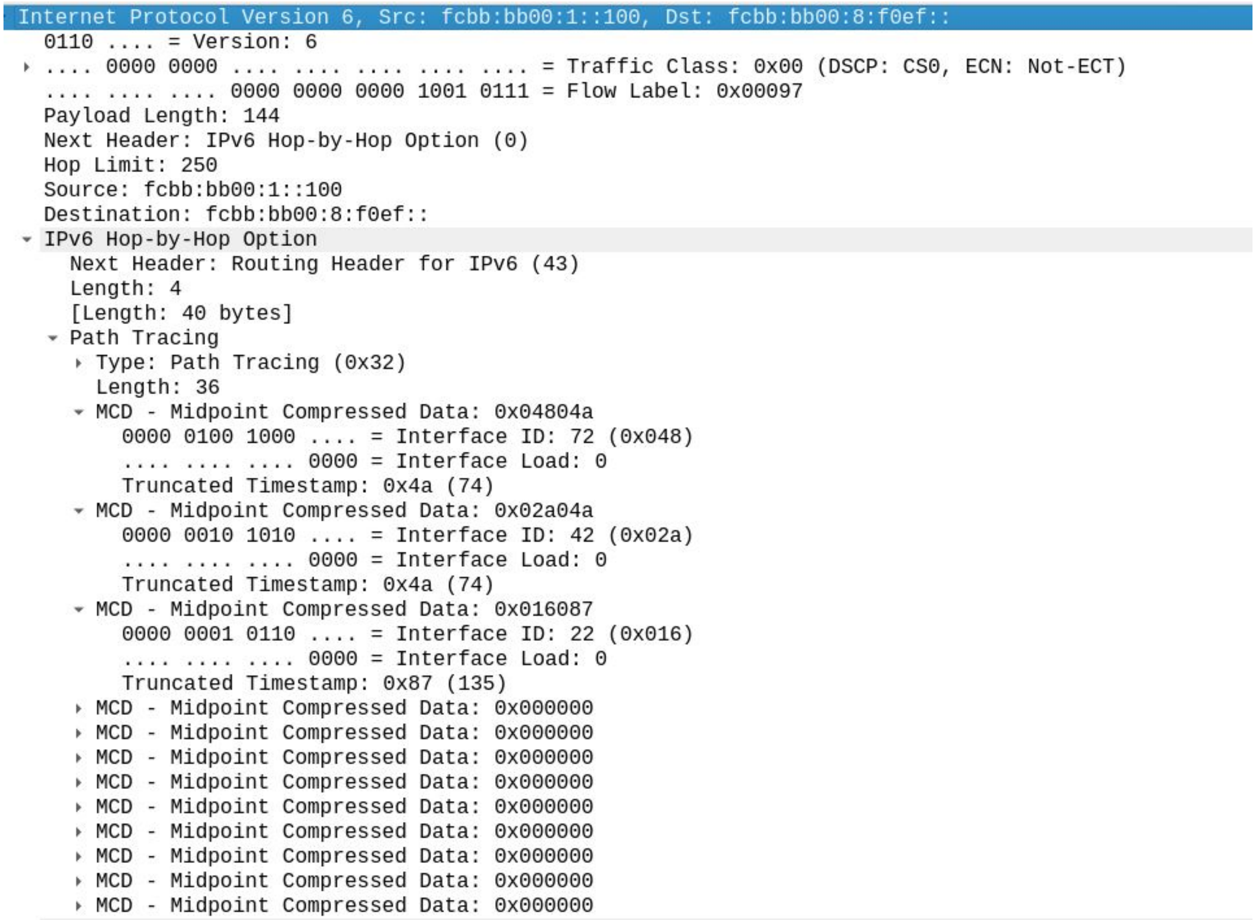
Appendices

Appendix A

Wireshark Path Tracing Captures

Here are Wireshark packet captures of path tracing probe packets. The Wireshark patch supporting path tracing is available at: `https://github.com/path-tracing/wireshark`.

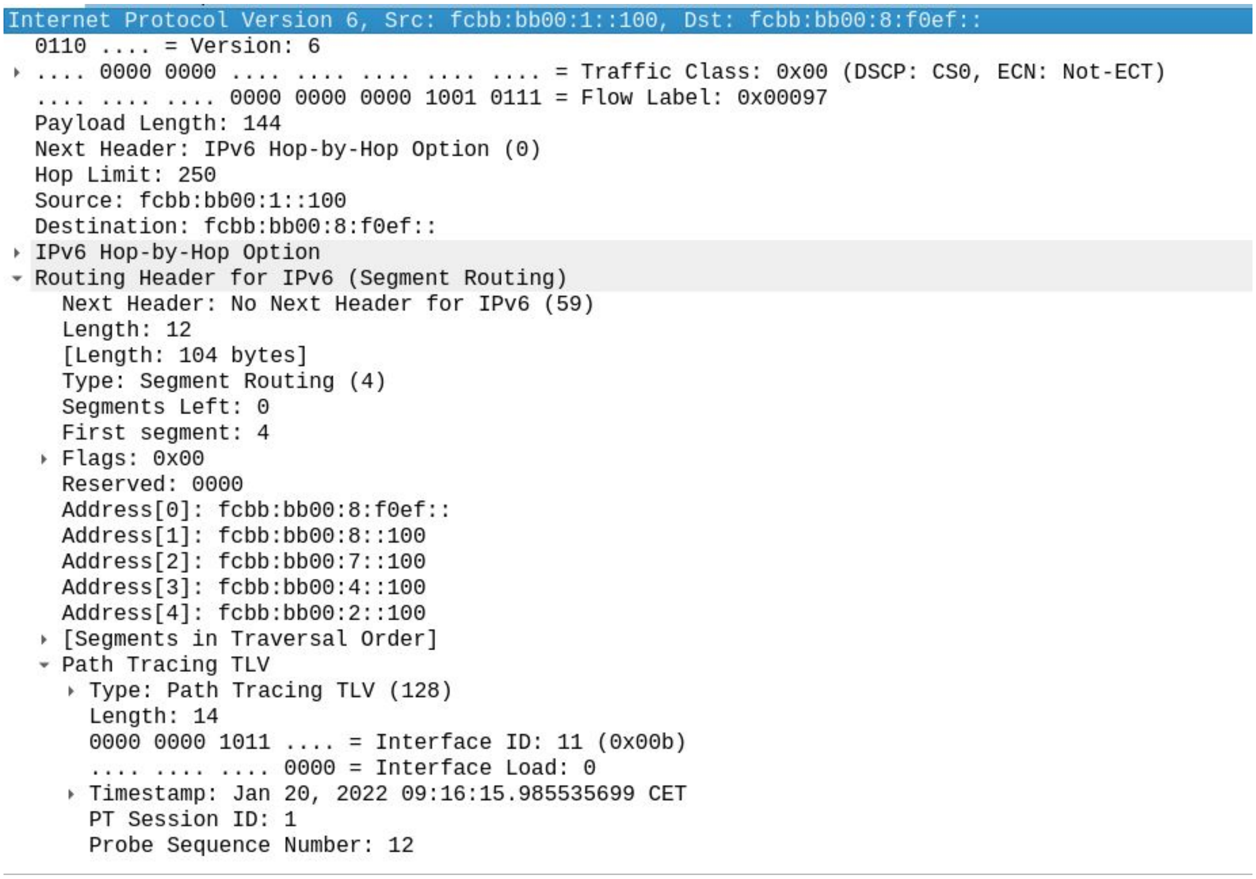
A.1 IPv6 HBH Option



```
 Internet Protocol Version 6, Src: fcbb:bb00:1::100, Dst: fcbb:bb00:8:f0ef::
   0110 .... = Version: 6
 ▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
   .... .... .... 0000 0000 0000 1001 0111 = Flow Label: 0x00097
   Payload Length: 144
   Next Header: IPv6 Hop-by-Hop Option (0)
   Hop Limit: 250
   Source: fcbb:bb00:1::100
   Destination: fcbb:bb00:8:f0ef::
 ▾ IPv6 Hop-by-Hop Option
     Next Header: Routing Header for IPv6 (43)
     Length: 4
     [Length: 40 bytes]
   ▾ Path Tracing
     ▸ Type: Path Tracing (0x32)
       Length: 36
     ▾ MCD - Midpoint Compressed Data: 0x04804a
         0000 0100 1000 .... = Interface ID: 72 (0x048)
         .... .... .... 0000 = Interface Load: 0
         Truncated Timestamp: 0x4a (74)
     ▾ MCD - Midpoint Compressed Data: 0x02a04a
         0000 0010 1010 .... = Interface ID: 42 (0x02a)
         .... .... .... 0000 = Interface Load: 0
         Truncated Timestamp: 0x4a (74)
     ▾ MCD - Midpoint Compressed Data: 0x016087
         0000 0001 0110 .... = Interface ID: 22 (0x016)
         .... .... .... 0000 = Interface Load: 0
         Truncated Timestamp: 0x87 (135)
     ▸ MCD - Midpoint Compressed Data: 0x000000
     ▸ MCD - Midpoint Compressed Data: 0x000000
     ▸ MCD - Midpoint Compressed Data: 0x000000
     ▸ MCD - Midpoint Compressed Data: 0x000000
     ▸ MCD - Midpoint Compressed Data: 0x000000
     ▸ MCD - Midpoint Compressed Data: 0x000000
     ▸ MCD - Midpoint Compressed Data: 0x000000
     ▸ MCD - Midpoint Compressed Data: 0x000000
     ▸ MCD - Midpoint Compressed Data: 0x000000
```

Figure A.1: IPv6 Path Tracing HBH Option Header.

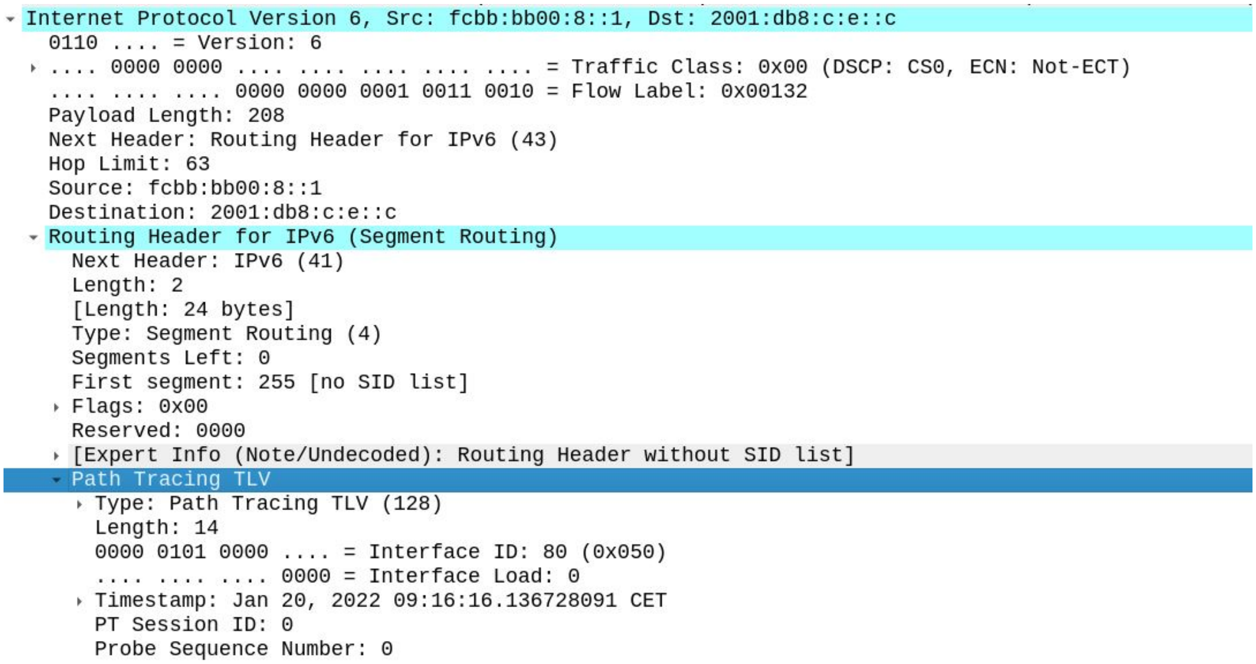
A.2 Source SRH TLV



```
Internet Protocol Version 6, Src: fcbb:bb00:1::100, Dst: fcbb:bb00:8:f0ef::
  0110 .... = Version: 6
▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
  .... .... .... 0000 0000 0000 1001 0111 = Flow Label: 0x00097
  Payload Length: 144
  Next Header: IPv6 Hop-by-Hop Option (0)
  Hop Limit: 250
  Source: fcbb:bb00:1::100
  Destination: fcbb:bb00:8:f0ef::
▸ IPv6 Hop-by-Hop Option
▾ Routing Header for IPv6 (Segment Routing)
    Next Header: No Next Header for IPv6 (59)
    Length: 12
    [Length: 104 bytes]
    Type: Segment Routing (4)
    Segments Left: 0
    First segment: 4
  ▸ Flags: 0x00
    Reserved: 0000
    Address[0]: fcbb:bb00:8:f0ef::
    Address[1]: fcbb:bb00:8::100
    Address[2]: fcbb:bb00:7::100
    Address[3]: fcbb:bb00:4::100
    Address[4]: fcbb:bb00:2::100
  ▸ [Segments in Traversal Order]
  ▾ Path Tracing TLV
    ▸ Type: Path Tracing TLV (128)
      Length: 14
      0000 0000 1011 .... = Interface ID: 11 (0x00b)
      .... .... .... 0000 = Interface Load: 0
    ▸ Timestamp: Jan 20, 2022 09:16:15.985535699 CET
      PT Session ID: 1
      Probe Sequence Number: 12
```

Figure A.2: Source SRH Header with Path Tracing TLV.

A.3 Sink SRH TLV



```
▾ Internet Protocol Version 6, Src: fcbb:bb00:8::1, Dst: 2001:db8:c:e::c
    0110 .... = Version: 6
  ▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 0000 0000 0001 0011 0010 = Flow Label: 0x00132
    Payload Length: 208
    Next Header: Routing Header for IPv6 (43)
    Hop Limit: 63
    Source: fcbb:bb00:8::1
    Destination: 2001:db8:c:e::c
  ▾ Routing Header for IPv6 (Segment Routing)
      Next Header: IPv6 (41)
      Length: 2
      [Length: 24 bytes]
      Type: Segment Routing (4)
      Segments Left: 0
      First segment: 255 [no SID list]
    ▸ Flags: 0x00
      Reserved: 0000
    ▸ [Expert Info (Note/Undecoded): Routing Header without SID list]
    ▾ Path Tracing TLV
      ▸ Type: Path Tracing TLV (128)
        Length: 14
        0000 0101 0000 .... = Interface ID: 80 (0x050)
        .... .... .... 0000 = Interface Load: 0
      ▸ Timestamp: Jan 20, 2022 09:16:16.136728091 CET
        PT Session ID: 0
        Probe Sequence Number: 0
```

Figure A.3: Sink SRH Header with Path Tracing TLV.

Appendix B

Path Tracing JSON Objects

B.1 pt.probe.raw

```
{
    "src_node": {
        "node_id": "",
        "addr": "/Lu7AAABAAAAAAAAAAAAAAQ==",
        "t64": 7049743939458283446,
        "out_interface_id": 11,
        "out_interface_load": 0,
        "out_interface_name": ""
    },
    "snk_node": {
        "node_id": "",
        "addr": "/Lu7AAAIAAAAAAAAAAAAAAQ==",
        "t64": 7049743940258383836,
        "in_interface_id": 81,
        "in_interface_load": 0,
        "tef_sid": "IAENuAAMAA4AAAAAAAAAADA==",
        "in_interface_name": ""
    },
    "traffic_class": 0,
    "flow_label": 1,
    "payload_length": 64,
    "hop_limit": 250,
    "hbh_opt_length": 36,
    "midpoint_count": 0,
    "mcd_stack": "A+ApArAuAWAyAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA",
    "srh_tag": 0,
    "srh_flag": 0,
    "segments_left": 0,
    "sid_list": null,
    "session_id": 1,
    "sequence_number": 7,
    "path_info": [
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        }
    ]
}
```

Listing B.1: pt.probe.raw JSON message.

B.2 pt.probe.processed

```
{
    "src_node": {
        "node_id": "vpp1",
        "addr": "fcbb:bb00:1::1",
        "t64": 7057538678431652836,
        "out_interface_id": 11,
        "out_interface_load": 0,
        "out_interface_name": "tap11"
    },
    "snk_node": {
        "node_id": "vpp8",
        "addr": "fcbb:bb00:8::1",
        "t64": 7057538678452034330,
        "in_interface_id": 81,
        "in_interface_load": 0,
        "tef_sid": "2001:db8:c:e::c",
        "in_interface_name": "tap81"
    },
    "traffic_class": 0,
    "flow_label": 105,
    "payload_length": 144,
    "hop_limit": 250,
    "hbh_opt_length": 36,
    "midpoint_count": 3,
    "mcd_stack": "A+BHAOBGAVBGAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA",
    "srh_tag": 0,
    "srh_flag": 0,
    "segments_left": 0,
    "sid_list": [
        "fcbb:bb00:2::100",
        "fcbb:bb00:5::100",
        "fcbb:bb00:6::100",
        "fcbb:bb00:8::100",
        "fcbb:bb00:8:f0ef::"
    ],
    "session_id": 1,
    "sequence_number": 5,
    "path_info": [
        {
            "node_id": "vpp6",
            "t64": 7057538678441902080,
            "out_interface_id": 62,
            "out_interface_load": 0,
            "out_interface_name": "tap62"
        },
        {
            "node_id": "vpp5",
            "t64": 7057538678441639936,
            "out_interface_id": 52,
            "out_interface_load": 0,
            "out_interface_name": "tap52"
        },
        {
            "node_id": "vpp2",
            "t64": 7057538678441639965,
            "out_interface_id": 21,
            "out_interface_load": 0,
            "out_interface_name": "tap21"
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        },
        {
            "node_id": "",
            "t64": 0,
            "out_interface_id": 0,
            "out_interface_load": 0,
            "out_interface_name": ""
        }
    ]
}
```

Listing B.2: pt.probe.processed JSON message.

B.3 pt.probe.global

```
{
    "src_node": {
        "node_id": "vpp1",
        "addr": "fcbb:bb00:1::1",
        "t64": 7057539335188787818,
        "out_interface_id": 11,
        "out_interface_load": 0,
        "out_interface_name": "tap11"
    },
    "snk_node": {
        "node_id": "vpp8",
        "addr": "fcbb:bb00:8::1",
        "t64": 7057539335209112773,
        "in_interface_id": 81,
        "in_interface_load": 0,
        "tef_sid": "2001:db8:c:e::c",
        "in_interface_name": "tap81"
    },
    "traffic_class": 0,
    "flow_label": 105,
    "payload_length": 144,
    "hop_limit": 250,
    "hbh_opt_length": 36,
    "midpoint_count": 3,
    "mcd_stack": "A+C4A0C4AVC4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA",
    "srh_tag": 0,
    "srh_flag": 0,
    "segments_left": 0,
    "sid_list": [
        "fcbb:bb00:2::100",
        "fcbb:bb00:5::100",
        "fcbb:bb00:6::100",
        "fcbb:bb00:8::100",
        "fcbb:bb00:8:f0ef::"
    ],
    "session_id": 1,
    "sequence_number": 5,
    "path_info": {
        "nodes_path": "vpp1 --> vpp2 --> vpp5 --> vpp6 --> vpp8",
        "nodes_path_list": [
            "vpp1",
            "vpp2",
            "vpp5",
            "vpp6",
            "vpp8"
        ],
        "interface_id_path": "11 --> 21 --> 52 --> 62 --> 81",
        "interface_name_path": "tap11 --> tap21 --> tap52 --> tap62 --> tap81",
        "delay": 20.32512
    },
    "sid_list_full": "fcbb:bb00:2::100 - fcbb:bb00:5::100 - fcbb:bb00:6::100 -
        fcbb:bb00:8::100 - fcbb:bb00:8:f0ef::"
}
```

Listing B.3: pt.probe.global JSON message.

B.4 pt.probe.hbh

```
{
    "src_node": {
        "node_id": "vpp1",
        "addr": "fcbb:bb00:1::1",
        "t64": 7057539335188787818,
        "out_interface_id": 11,
        "out_interface_load": 0,
        "out_interface_name": "tap11"
    },
    "snk_node": {
        "node_id": "vpp8",
        "addr": "fcbb:bb00:8::1",
        "t64": 7057539335209112773,
        "in_interface_id": 81,
        "in_interface_load": 0,
        "tef_sid": "2001:db8:c:e::c",
        "in_interface_name": "tap81"
    },
    "traffic_class": 0,
    "flow_label": 105,
    "payload_length": 144,
    "hop_limit": 250,
    "hbh_opt_length": 36,
    "midpoint_count": 3,
    "mcd_stack": "A+C4A0C4AVC4AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA",
    "srh_tag": 0,
    "srh_flag": 0,
    "segments_left": 0,
    "sid_list": [
        "fcbb:bb00:2::100",
        "fcbb:bb00:5::100",
        "fcbb:bb00:6::100",
        "fcbb:bb00:8::100",
        "fcbb:bb00:8:f0ef::"
    ],
    "session_id": 1,
    "sequence_number": 5,
    "path_info": {
        "nodes_path": "vpp1 --> vpp2 --> vpp5 --> vpp6 --> vpp8",
        "nodes_path_list": [
            "vpp1",
            "vpp2",
            "vpp5",
            "vpp6",
            "vpp8"
        ],
        "interface_id_path": "11 --> 21 --> 52 --> 62 --> 81",
        "interface_name_path": "tap11 --> tap21 --> tap52 --> tap62 --> tap81",
        "delay": 20.32512
    },
    "link_info": {
        "src_node_id": "vpp6",
        "dst_node_id": "vpp8",
        "src_interface_id": 62,
        "dst_interface_id": 81,
        "src_interface_name": "tap62",
        "dst_interface_name": "tap81",
        "link_id": "br68",
        "src_t64": 7057539335198867456,
        "dst_t64": 7057539335209112773,
        "delay": 10.245376
    },
    "sid_list_full": "fcbb:bb00:2::100 - fcbb:bb00:5::100 - fcbb:bb00:6::100 -
        fcbb:bb00:8::100 - fcbb:bb00:8:f0ef::"
}
```

Listing B.4: pt.probe.hbh JSON message.

B.5 pt.ipfix.raw

```
{
    "event_type": "purge",
    "peer_ip_src": "192.168.0.5",
    "iface_in": 5,
    "iface_out": 3,
    "ip_src": "fcbb:bb00:6::2",
    "ip_dst": "fcbb:bb00:7:f0ef::",
    "ip_proto": "0",
    "timestamp_start": "1644234403.000000",
    "timestamp_end": "0.000000",
    "timestamp_arrival": "1644234403.322197",
    "timestamp_min": "1644234403.000000",
    "timestamp_max": "1644234403.000000",
    "timestamp_export": "1644234403.000000",
    "stamp_inserted": "1644234360",
    "stamp_updated": "1644234411",
    "packets": 181,
    "bytes": 21720,
    "writer_id": "ptipfix/146530"
}
```

Listing B.5: pt.ipfix.raw JSON message.

B.6 pt.ipfix.processed

```json
{
    "event_type": "purge",
    "peer_ip_src": "192.168.0.4",
    "iface_in": 3,
    "iface_out": 6,
    "ip_src": "fcbb:bb00:1::1",
    "ip_dst": "fcbb:bb00:8:f0ef::",
    "ip_proto": "ipv6-hbh",
    "timestamp_start": "1645613144.000000",
    "timestamp_end": "0.000000",
    "timestamp_arrival": "1645613144.969221",
    "timestamp_min": "1645613144.000000",
    "timestamp_max": "1645613145.000000",
    "timestamp_export": "1645613144.000000",
    "stamp_inserted": "1645613100",
    "stamp_updated": "1645613151",
    "packets": 64,
    "bytes": 7680,
    "writer_id": "ptipfix/489921",
    "peer_id": "vpp4",
    "iface_in_id": "41",
    "iface_in_name": "tap41",
    "link_in": "br34",
    "link_in_connected_iface": "31",
    "link_in_connected_node": "vpp3",
    "iface_in_avg_delay": 2.0519147610619424,
    "iface_in_count_considered_for_delay": 226,
    "iface_in_max_delay": 2.359296,
    "iface_in_min_delay": 1.835008,
    "iface_out_id": "44",
    "iface_out_name": "tap44",
    "link_out": "br48",
    "link_out_connected_iface": "80",
    "link_out_connected_node": "vpp8",
    "iface_out_avg_delay": 3.128008517110266,
    "iface_out_count_considered_for_delay": 263,
    "iface_out_max_delay": 3.41376,
    "iface_out_min_delay": 2.777088
}
```

Listing B.6: pt.ipfix.processed JSON message.

Appendix C

SQL Queries and Table Joins

C.1 SQL Query - 3-hop paths

```
SELECT ipfix1.__time AS __time,
       ipfix1.ip_src AS ip_src,
       ipfix1.ip_dst AS ip_dst,
       ipfix1.link_in_connected_node AS src_node,
       ipfix1.link_in AS link_1,
       ipfix1.peer_id AS mid_node_1,
       ipfix1.link_out AS link_2,
       ipfix1.link_out_connected_node AS snk_node,
       ipfix1.iface_in_avg_delay AS link_1_delay_avg, ipfix1.iface_in_max_delay AS
           link_1_delay_max, ipfix1.iface_in_min_delay AS link_1_delay_min,
       ipfix1.iface_out_avg_delay AS link_2_delay_avg, ipfix1.iface_out_max_delay AS
           link_2_delay_max, ipfix1.iface_out_min_delay AS link_2_delay_min,
       ipfix1.iface_in_avg_delay + ipfix1.iface_out_avg_delay AS full_path_delay_avg,
       ipfix1.packets AS link_1_packets, ipfix1.packets AS link_2_packets,
       ipfix1.bytes AS link_1_bytes, ipfix1.bytes AS link_2_bytes,
       ipfix1.iface_in_count_considered_for_delay AS link_1_count_considered_for_delay,
       ipfix1.iface_out_count_considered_for_delay AS link_2_count_considered_for_delay
FROM "pt.ipfix.processed" ipfix1
WHERE ipfix1.__time >= CURRENT_TIMESTAMP - INTERVAL '1' HOUR
ORDER BY ipfix1.__time DESC
```

Listing C.1: Druid SQL query (no join, for 3-hop paths)

C.2 pt.ipfix.joined - 3-hop paths

```
{
    "__time": "2022-02-24T07:36:09.318Z",
    "ip_src": "fcbb:bb00:7::2",
    "ip_dst": "fcbb:bb00:1:f0ef::",
    "src_node": "vpp7",
    "link_1": "br37",
    "mid_node_1": "vpp3",
    "link_2": "br13",
    "snk_node": "vpp1",
    "link_1_delay_avg": 3.147104000000000,
    "link_1_delay_max": 3.67488,
    "link_1_delay_min": 2.808832,
    "link_2_delay_avg": 3.10636246913580,
    "link_2_delay_max": 3.91936,
    "link_2_delay_min": 2.83136,
    "full_path_delay_avg": 6.253466469135803,
    "link_1_packets": 31,
    "link_2_packets": 31,
    "link_1_bytes": 3720,
    "link_2_bytes": 3720,
    "link_1_count_considered_for_delay": 328,
    "link_2_count_considered_for_delay": 324
  }
```

Listing C.2: pt.ipfix.joined JSON message (no join, 3-hop paths)

C.3 SQL Query - 4-hop paths

```
SELECT ipfix1.__time AS __time,
       ipfix1.ip_src AS ip_src,
       ipfix1.ip_dst AS ip_dst,
       ipfix1.link_in_connected_node AS src_node,
       ipfix1.link_in AS link_1, ipfix1.peer_id AS mid_node_1,
       ipfix2.link_in AS link_2, ipfix2.peer_id AS mid_node_2,
       ipfix2.link_out AS link_3,
       ipfix2.link_out_connected_node AS snk_node,
       ipfix1.iface_in_avg_delay AS link_1_delay_avg, ipfix1.iface_in_max_delay AS
           link_1_delay_max, ipfix1.iface_in_min_delay AS link_1_delay_min,
       ipfix2.iface_in_avg_delay AS link_2_delay_avg, ipfix2.iface_in_max_delay AS
           link_2_delay_max, ipfix2.iface_in_min_delay AS link_2_delay_min,
       ipfix2.iface_out_avg_delay AS link_3_delay_avg, ipfix2.iface_out_max_delay AS
           link_3_delay_max, ipfix2.iface_out_min_delay AS link_3_delay_min,
       ipfix1.iface_in_avg_delay + ipfix2.iface_in_avg_delay +
           ipfix2.iface_out_avg_delay AS full_path_delay_avg,
       ipfix1.packets AS link_1_packets, ipfix2.packets AS link_2_packets,
           ipfix2.packets AS link_3_packets,
       ipfix1.bytes AS link_1_bytes, ipfix2.bytes AS link_2_bytes, ipfix2.bytes AS
           link_3_bytes,
       ipfix1.iface_in_count_considered_for_delay AS link_1_count_considered_for_delay,
       ipfix2.iface_in_count_considered_for_delay AS link_2_count_considered_for_delay,
       ipfix2.iface_out_count_considered_for_delay AS link_3_count_considered_for_delay,
       TIMESTAMP_TO_MILLIS(ipfix1.__time) - TIMESTAMP_TO_MILLIS(ipfix2.__time) AS
           time_difference
FROM "pt.ipfix.processed" ipfix1
INNER JOIN "pt.ipfix.processed" ipfix2 ON (ipfix1.link_out = ipfix2.link_in and
    ipfix1.ip_src = ipfix2.ip_src and ipfix1.ip_dst = ipfix2.ip_dst)
WHERE ipfix1.__time >= CURRENT_TIMESTAMP - INTERVAL '1' HOUR AND ipfix2.__time >=
    CURRENT_TIMESTAMP - INTERVAL '1' HOUR
     AND TIMESTAMP_TO_MILLIS(ipfix1.__time) - TIMESTAMP_TO_MILLIS(ipfix2.__time) < 10000
     AND TIMESTAMP_TO_MILLIS(ipfix1.__time) - TIMESTAMP_TO_MILLIS(ipfix2.__time) > -10000
ORDER BY ipfix1.__time DESC
```

Listing C.3: Druid SQL query (1 join, for 4-hop paths)

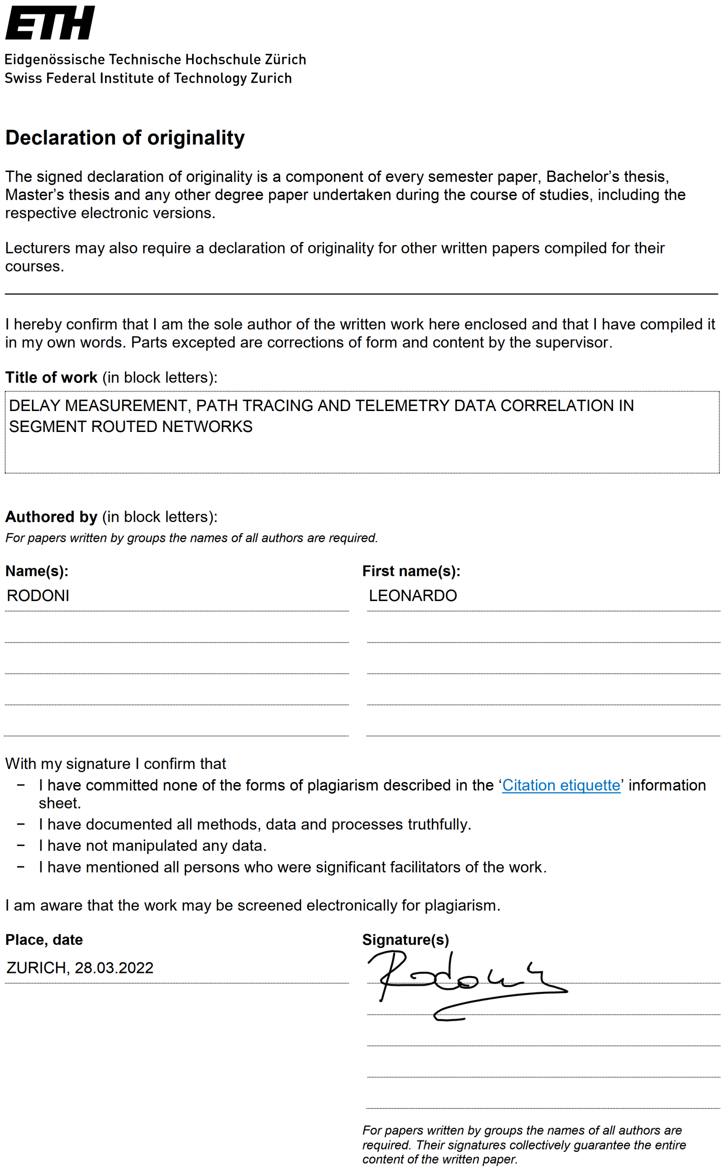
C.4 pt.ipfix.joined - 4-hop paths

```
{
    "__time": "2022-02-24T07:37:54.446Z",
    "ip_src": "fcbb:bb00:8::1",
    "ip_dst": "fcbb:bb00:1:f0ef::",
    "src_node": "vpp8",
    "link_1": "br48",
    "mid_node_1": "vpp4",
    "link_2": "br34",
    "mid_node_2": "vpp3",
    "link_3": "br13",
    "snk_node": "vpp1",
    "link_1_delay_avg": 3.235917772151899,
    "link_1_delay_max": 4.175872,
    "link_1_delay_min": 2.875648,
    "link_2_delay_avg": 2.2065686260869537,
    "link_2_delay_max": 6.029312,
    "link_2_delay_min": 1.835008,
    "link_3_delay_avg": 3.1524461359223293,
    "link_3_delay_max": 3.637248,
    "link_3_delay_min": 2.829056,
    "full_path_delay_avg": 8.594932534161181,
    "link_1_packets": 6,
    "link_2_packets": 6,
    "link_3_packets": 6,
    "link_1_bytes": 720,
    "link_2_bytes": 720,
    "link_3_bytes": 720,
    "link_1_count_considered_for_delay": 158,
    "link_2_count_considered_for_delay": 115,
    "link_3_count_considered_for_delay": 103
  }
```

Listing C.4: pt.ipfix.joined JSON message (1 join, 4-hop path)

Appendix D

Declaration of Originality



ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

DELAY MEASUREMENT, PATH TRACING AND TELEMETRY DATA CORRELATION IN SEGMENT ROUTED NETWORKS

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
RODONI	LEONARDO

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZURICH, 28.03.2022	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.