Network Insights with P4 In-Band Network Telemetry

Sebastian Warter, Sebastian Gallenmüller*, Kilian Holzinger*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: sebastian.warter@tum.de, gallenmu@net.in.tum.de, holzinger@net.in.tum.de

***Abstract*—Understanding what happens to packets in layer 2 networks is inherently difficult. The transparent nature of network switches can make the identification of faulty network components a time-consuming search. The P4 In-Band Network Telemetry (INT) can help by aggregating switch telemetry information in the data packets. This paper explains the concepts of INT in a simple use case where we want to identify a high-latency link. With the P4 implementation from the GÉANT project, we demonstrate this scenario in a virtual network. The evaluation of this experiment shows that INT is suitable for the use case but has some overhead in the emulated network.**

***Index Terms*—P4, in-band network telemetry, INT, monitoring**

1. Introduction

The operation and administration of large networks can be a challenging task. For example, assume that we are dealing with a video conference company network. Such a network usually has to transmit a considerable amount of UDP packets with low latency. There is likely also a monitoring system in place that can detect abnormal high latencies or low data rates from an end-to-end perspective.

However, this information does not help to identify the problematic network component. If the network consists of layer 3 routers, a tool like `tracepath` can help to narrow the issue down. Unfortunately, it does not work in all cases. In complex networks, the routers might handle the echo request differently than the real traffic without triggering the problem. The problem might also involve a layer 2 switch or a link between two switches. They are transparent to the `tracepath` tool.

Ideally, we would have a technology that can track on-demand how regular data packets move through the switches in a layer 2 network. If a switch sends metadata to a monitoring system every time it encounters a tracked packet, we could reconstruct the layer 2 path of each packet including time information. This approach would still require a pre-existing identifier in the packets, which is not always available.

A different approach is to add the metadata to the packet itself at each switch it passes. The aggregated metadata then has to be removed from the packet once it leaves the network. This does not require unique IDs for every packet.

The second approach can be realized using P4 programmable switches and the In-Band Network Telemetry (INT) specification described in Section 2. The following Section 3 describes how to solve the use case described at the beginning with INT. In Section 4, we pick a suitable P4 implementation to create a test setup in Section 5. This test setup is then evaluated in Section 6 on our use case.

Similar demonstration setups were also described in related work. For example, Kim et al. briefly described a simple demonstration setup based on an older INT specification [1]. Parniewicz et al. described their results on more complex network environments [2]. This paper focuses on a simple use case instead.

2. Background

Traditionally, professional network hardware is highly specialized and has limited configuration possibilities. Even though this usually means it operates efficiently, it has the downside that new functionality or protocols usually require buying expensive new hardware. The P4 ecosystem presented in Subsection 2.1 aims to change that with a programming language for packet processing. In order to emulate networks with P4 switches, the mininet project described in Subsection 2.2 can be used. The flexibility of P4 also allows to implement more advanced monitoring systems like the In-band Network Telemetry (INT) in Subsection 2.3.

2.1. P4

P4 stands for "Programming Protocol-Independent Packet Processors" and was first described by Bosshart et al. [3]. It is a programming language for packet processing that describes detailed steps to perform on incoming network packets. It does not assume the usage of standardized protocols like IP or TCP, which is usually a prerequisite for traditional network hardware. Instead, it allows the programmer to define the header structures themselves. Due to this flexibility, existing P4-based hardware can also be used for network protocols that do not exist yet. [3].

The P4 language (more precisely, the 2016 revision $P4_{16}$) has two essential language constructs. "Parsers" parse the headers of an incoming packet. They use a programmer-defined state machine to identify and parse nested headers. "Control blocks" are imperative programs. They can use tables to trigger programmer-defined actions based on the value of header fields. The table entries are usually not part of the program and are configured by the control plane at startup or runtime (for example, a routing table). [4].

The possible actions a switch can perform, and the processing pipeline itself, is not enforced by P4. Instead,

 doi: 10.2313/NET-2023-11-1_22

it provides the syntax to describe the capabilities of switch architectures. For example, a certain switch model might provide functions to calculate CRC checksums, while other switch models do not support them. [4].

The reference switch architecture v1model[1] is based on a simple packet processing pipeline with six predefined steps (including ingress/egress pipeline and emitting headers at the end). The P4 project also provides a software implementation of a switch called "Behavior Model 2" ("bmv2") that supports the v1model [5].

2.2. Mininet

Developing P4 applications using real hardware is difficult. The hardware is usually expensive and difficult to reset to a clean state or to debug. For developing systems with P4-based switches, a virtual network setup is more convenient. Although such an emulated network fails to reflect real networks accurately, it can be a valuable tool for network experiments [6].

A popular tool for creating virtual networks is mininet. It is based on Linux's built-in virtualization capabilities for network interfaces. Similar to container solutions like Docker, each virtual host is represented by a process with virtual network interfaces. This way, hosts in a mininet network can efficiently run normal applications. Switches in a mininet network are OpenFlow-compatible software implementations. [7].

The mininet command-line interface directly supports the creation of simple, pre-defined network topologies. Once it is running, it offers commands to inspect the topology and run normal shell commands on the virtual hosts. This way, the network settings of hosts can be configured using normal Linux commands. If more complex topologies are required, they can be defined using an object-oriented Python API. [7].

In order to test P4 applications in mininet, the OpenFlow-based switch can be replaced with a bmv2 switch. The P4 ecosystem offers the tool `p4app` for this purpose. It is a convenient tool that can compile a P4 program, start a mininet topology, and configure the tables on the switches. Because it is based on docker containers, it offers a development environment with all necessary tools without using resource-hungry alternatives like virtual machines. [8].

2.3. P4 In-band Network Telemetry (INT)

Understanding what happens in layer 2 networks is inherently difficult. The switches in the network are, by design, supposed to be transparent to network traffic. This means in practice, that it is not directly possible to know which path a specific packet took through a network or how individual links affect the total latency. With sophisticated hardware, it is usually possible to access additional information like packet rates. Unfortunately, they only allow us to guess what is happening in the network.

To improve insights into networks, the P4 working group specified a protocol called "In-band network telemetry (INT)". It is designed to track the visited switches and their states of any packet flow in a P4-based network. For this purpose, INT allows storing the instruction to collect telemetry and the switch states in the already existing data packets. [9].

1. https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4

TABLE 1: INT Hop-by-hop header, adapted from [9]

0	1	2	3
Version | Flags	Reserved	Hop ML	Remaining Cnt
Instruction Bitmap	Reserved		
Last hop INT metadata			
...			
First hop INT metadata			

In INT 1.0, the instruction to collect telemetry originates from an "INT source" switch. It injects the header in Table 1 into the forwarded packets. In particular, it sets bits in the "Instruction Bitmap" which correspond to the information that should be collected. This switch and all "INT transit hop" switches can append their own state to this header while forwarding the packet. Eventually, an "INT sink" switch removes the header and sends the collected data to a monitoring system. The report format is not specified in INT 1.0. [9]

The collected metadata usually includes information like device identifiers or timestamps. This information can then be used to visualize the layer 2 path of packets. It also allows to calculate the link and switch latencies from timestamps in the metadata.

The previously described setup is called "INT-MD" in the most recent version 2.1 of INT. The new version also adds two additional modes. In "INT-MX" mode, the telemetry is sent directly to the monitoring system by each switch instead of appending it to the header. This avoids packets that grow too large and exceed the MTU. The "INT-XD" mode works similarly, but the switches do not create an INT header and use built-in instructions to send telemetry. [10].

3. A Simple Monitoring Use Case

In order to illustrate the potential of INT, this paper demonstrates its benefits in a simple use case. We assume that a network operator wants to localize latency issues in a layer 2 network. With traditional tooling, this would be difficult because layer 2 switches are usually transparent to network traffic.

This problem can be solved with a simple INT setup. In this setup, each monitored packet aggregates a history of switch states as it passes the network. When the packet leaves the network, this information is sent to a monitoring system. The monitoring system can then be used to analyze the data. In our use case, it can calculate latencies for each link based on the INT timestamp differences.

To demonstrate this in practice, we create a virtual test network. It uses

- mininet to create three bmv2 switches and two hosts
- P4 code to create and process the INT headers
- a simple INT-MD configuration with one INT source, one INT transit hop, and one INT sink

 doi: 10.2313/NET-2023-11-1_22

TABLE 2: INT switch implementations

Name	bmv2	INT version (mode)	working documentation
joshi	×	1.0/2.1 (all)	not tested
ONOS	✓	1.0 (MD)	×
GÉANT	✓	1.0 (MD)	✓

TABLE 3: INT collector implementations

Name	Backend	Works on Linux 5.19
INTCollector	InfluxDB, Prometheus	×
GÉANT	InfluxDB	✓

- node ids, ingress timestamps, and egress timestamps, and
- a monitoring system that can collect and visualize the INT data.

For a real INT deployment, it is usually desirable to collect additional data like congestion indicators. Even though they can provide valuable information to diagnose network issues, they are omitted here to avoid additional complexity.

4. P4 INT Implementations

Over time, many developers have implemented different versions of the INT specification in P4-based projects. This section briefly compares three well-documented implementations in Subsection 4.1. After that, it describes details about the GÉANT implementation in Subsection 4.2, which is used in the remaining sections of this paper.

4.1. Choosing a Suitable Implementation

In practice, we use two main software components for our test setup:

- a P4 implementation of a network switch with INT support, and
- an application that collects the INT headers and transforms them into a format compatible with an existing database system.

This paper considers and compares the three P4 implementations summarized in Table 2 and the two collectors summarized in Table 3. Numerous other implementations exist, but many are based on the outdated version 0.4 of INT or are not properly documented.

The implementation created by Joshi in [11] is one of the most recent ones and supports the latest INT version 2.1. Unfortunately, it is built solely for Intel Tofino hardware and does not support the bmv2. The INT 1.0 implementation in the Open Network Operating System (ONOS) uses the bmv2, but its documentation[2] is outdated and does not work in current versions of ONOS. The last implementation described by Parniewicz et al. [2] as part of a GÉANT project is similar but has a working documentation.

2. https://wiki.onosproject.org/display/ONOS/In-band+Network+Telemetry+(INT)+with+ONOS+and+P4

A frequently used collector is the INTCollector described by Tu et al. [12], which can send data to InfluxDB or Prometheus backends. Unfortunately, this implementation fails to start on current Linux versions. For demonstration purposes, we can also use the slower Python implementation included in the GÉANT project to store the data in InfluxDB.

Only the switch implementation from the GÉANT projects seems suitable for our virtual test setup. We use it in the following sections for our use case from Section 3. For simplicity, we also use the INT collector included in the GÉANT project.

4.2. The implementation of the GÉANT project

This P4 implementation of INT was created as part of a GÉANT project about network monitoring. It includes an implementation of INT 0.4 and 1.0 for both virtual bmv2 switches and Intel Tofino switches. It also has extensive documentation and also provides visualization tools. [13].

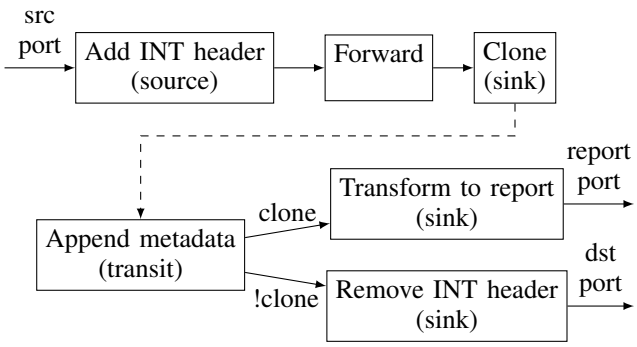

Figure 1: GÉANT P4 pipeline based on `int.p4`

The used P4 processing pipeline is visualized in Figure 1. It is configured using P4 tables [14].

In the ingress pipeline, the switch first adds the INT header if a packet should be monitored. This decision is made based on a flag to use a port as an INT source and a list of layer 3/4 endpoint addresses to monitor. Next, the egress port is picked from a static layer 2 address forward table. If it is a port configured as an INT sink, the packet is also cloned to the reporting port. [14].

The egress pipeline first appends the local metadata if there is already an INT header in the packet. If the destination is an INT sink port, the INT headers are removed in the next step. If it is a cloned packet sent to the reporting port, the packet headers are wrapped with a report header in order to send them to the IP address of the collector. The GÉANT project seems to use the same headers as the Telemetry Report 1.0 specification for this purpose [15]. [14].

5. Creating the P4 INT Test Setup

In the P4 INT demonstration setup, we have to install, configure, and start multiple software components for monitoring and the virtual network. For our simple requirements, the docker-based configurations shipped with the GÉANT project allow us to create an environment that matches our requirements:

1) Follow the instructions[3] to start and configure docker containers for InfluxDB and the grafana dashboard. These components act as the monitoring system that stores and visualizes the INT metrics. The collector itself is part of the next step.
2) Start the INT 1.0 mininet testbed that is shipped with the P4 implementation of the GÉANT project. The included instructions[4] describe how to start the collector and use `p4app` to create a virtual network in a docker container (external connectivity is not required). Note that the InfluxDB IP address should be a public IP of the host.

In the default configuration, the setup consists of three statically configured switches. Each can act as an INT source, transit hop, or sink. The network parts relevant to this paper are visualized in Figure 2 based on the `dump/net` output of mininet [7].

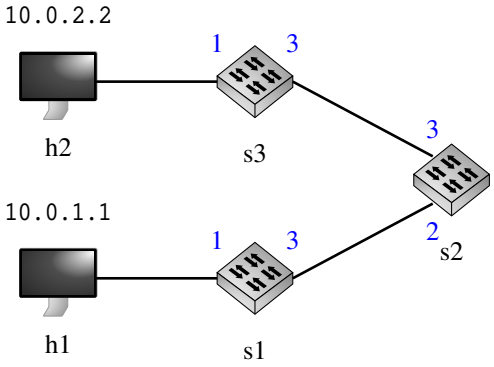

Figure 2: Layer 2 path between h1 and h2 with IP addresses and port numbers

This setup now includes all parts required for our monitoring use case from Section 3. For example, if we send data from host 1 to host 2, switch 1 acts as an INT source and adds instructions to collect all INT fields (due to the configuration in `commands1.txt`). All switches then add INT headers with telemetry while forwarding the packets. Switch 3 removes the headers and sends the INT data to the collector, which processes them and sends the telemetry to the InfluxDB server.

6. Evaluation of the Test Setup

We now use the previously created test setup to demonstrate how INT can help to localize latency issues. For this purpose, we first introduce a 5000 ms delay for data sent from `s1` to `s2`. Next, we start sending packets from `h1` to the IP address of `h2`. The GÉANT project provides us with the Python script `h1_h2_udp_flow.py` for this purpose. Both steps can be achieved by executing the commands in Figure 3 in the mininet prompt [7], [16].

The collected INT metadata can be analyzed in the Grafana dashboard. For our use case, we want to find

3. https://github.com/GEANT-DataPlaneProgramming/int-analytics
4. https://github.com/GEANT-DataPlaneProgramming/int-platforms/tree/master/platforms/bmv2-mininet

```
s1 tc qdisc add dev s1-eth3 root netem delay 5s
h1 python /tmp/host/h1_h2_udp_flow.py
```

Figure 3: Mininet commands used for our test setup

latency issues. The interesting values for this purpose are the pre-hop link delays (see Figure 4).

In our experiment, the delay between Switch 1 and 2 is about 6 s. This is higher than the near-zero delay between Switch 2 and 3. Therefore, we have identified our high-latency link.

Unfortunately, this experiment also reveals some limitations. There is an additional delay of about 1 s on the link `s1-s2` and not on the link `s2-s3`. We suspect that it is caused by the overhead of the software switch, but this hypothesis cannot be verified without tests on real hardware.

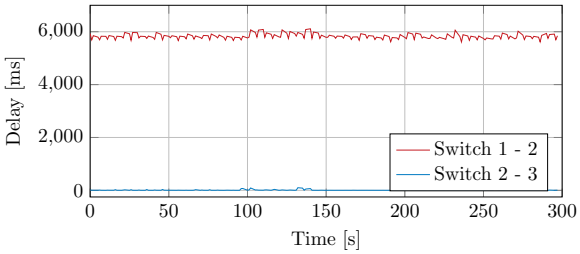

Figure 4: Link delays from experiment

Overall, this virtual experiment showed how it is possible to find high-latency links in an INT-capable network. Unfortunately, there is a significant overhead which would make it difficult to measure lower, more realistic latencies. Further evaluation of INT's accuracy in this use case would likely require P4-capable hardware and is out of the scope of this paper.

7. Conclusion

In this paper, we saw how INT can help to solve network problems in previously not possible ways. Based on a simple use case where we located latency issues in layer 2 networks, we explained INT and created a virtual demonstration setup. Our evaluation showed that INT is suitable for the use case but, at least in our emulated network, has a significant overhead.

Based on this setup, it is also possible to collect other potentially helpful INT data. For example, a network operator can decide to collect the queue occupancy or the exact layer 2 path of a packet. Accurately evaluating the precision of INT in these more complex use cases will require real hardware and future work. Such future work should also consider the newer version 2.1 of INT, which can provide additional possibilities to debug network issues.

References

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

[2] D. Parniewicz, T. Martínek, F. Pederzolli, D. Ding, M. Campanella, I. Golub, and T. Chown, "In-Band Network Telemetry Tests in NREN Networks," GÉANT Association, Tech. Rep., 2021.

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

[4] The P4 Language Consortium, "P4_16 Language Specification," https://p4.org/p4-spec/docs/P4-16-v1.2.4.pdf, 2023, [Online; accessed 15-June-2023].

[5] P4 Project, "The Reference P4 Software Switch," https://github.com/p4lang/behavioral-model, 2023, [Online; accessed 15-June-2023].

[6] R. Oliveira, C. Schweitzer, A. Shinoda, and L. Prete, "Using mininet for emulation and prototyping software-defined networks," 06 2014, pp. 1–6.

[7] Mininet Project Contributors, "Mininet Walkthrough," http://mininet.org/walkthrough/, 2022, [Online; accessed 15-June-2023].

[8] P4 Project, "P4app," https://github.com/p4lang/p4app, 2019, [Online; accessed 15-June-2023].

[9] The P4.org Applications Working Group, "In-band Network Telemetry (INT) Dataplane Specification - Version 1.0," https://p4.org/p4-spec/docs/INT_v1_0.pdf, 2018, [Online; accessed 15-June-2023].

[10] ——, "In-band Network Telemetry (INT) Dataplane Specification - Version 2.1," https://p4.org/p4-spec/docs/INT_v2_1.pdf, 2020, [Online; accessed 15-June-2023].

[11] M. Joshi, "Implementation and Evaluation of In-Band Network Telemetry in P4," Master's thesis, KTH Royal Institute of Technology, 2021.

[12] N. V. Tu, J. Hyun, G. Y. Kim, J.-H. Yoo, and J. W.-K. Hong, "Intcollector: A high-performance collector for in-band network telemetry," in *2018 14th International Conference on Network and Service Management (CNSM)*, 2018, pp. 10–18.

[13] D. Parniewicz, "Common P4-based INT implementation for bmv2-mininet and Tofino platforms," https://github.com/GEANT-DataPlaneProgramming/int-platforms, 2021, [Online; accessed 15-June-2023].

[14] ——, "INT Configuration Guide," https://github.com/GEANT-DataPlaneProgramming/int-platforms/blob/master/docs/configuration.md, 2021, [Online; accessed 15-June-2023].

[15] The P4.org Applications Working Group, "Telemetry Report Format Specification - Version 1.0," https://raw.githubusercontent.com/p4lang/p4-applications/master/docs/telemetry_report_v1_0.pdf, 2018, [Online; accessed 15-June-2023].

[16] F. Ludovici and H. P. Pfeifer, *tc-netem(8) Linux Manual Page*, 2011.