University of Macedonia

Graduate Program

Department of Applied Informatics

IMPLEMENTATION OF NETWORK TELEMETRY SYSTEM WITH P4 PROGRAMMING LANGUAGE

Master Thesis

of

Demertzis Christos

Thessaloniki, June 2020

IMPLEMENTATION OF NETWORK TELEMETRY SYSTEM WITH P4 PROGRAMMING LANGUAGE

Demertzis Christos

Bachelor in Business Administration, Open University, 2018

Master Thesis

submitted for the partial fulfillment of the demands for the

MASTER OF SCIENCE IN APPLIED INFORMATICS

Supervising Professor
Dr. Papadimitriou Panagiotis

Approved by the three-member Inquiry Committe on . ./. ./. . . .

Panagiotis Papadimitriou	Eleftherios Mamatas	Ilias Sakellariou
....	....	....

Demertzis Christos

....

Abstract

A very important factor for the network to operate is the monitoring. In-band Network Telemetry is a mechanism for monitoring Software Defined Networks providing fine grained telemetry information with less overhead on the control plane because the monitoring takes place in the data plane (in-band). In this thesis we examine the P4 language and the In-band Network Telemetry and propose a simple network monitoring architecture to monitor the network state information, identify and avoid congestions and apply load balancing.

Keywords: Inband Network Telemetry(INT), P4, $P4_{16}$, Software Defined Networks(SDN), Control Plane, Data Plane, Mininet, Equal-Cost Multi-Path Routing (ECMP)

Acknowledgements

It is with immense gratitude that I acknowledge the support and help of my Professor Panagiotis Papadimitriou at University of Macedonia. This thesis would not have been possible without his guidance.

I would also like to thank my family whose support have allowed me to pursue my academic interests. Without their help none of this would have been possible.

Contents

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Objectives . . . 2
- 1.3 Structure of the Thesis . . . 3

2 Programmable Networks 4
- 2.1 Software Defined Networking (SDN) . . . 4
- 2.2 OpenFlow (OF) . . . 6
- 2.3 P4 Language Specifications . . . 9
  - 2.3.1 $P4_{16}$ Overview . . . 9
  - 2.3.2 $P4_{16}$ Benefits . . . 14
  - 2.3.3 $P4_{16}$ Architecture . . . 15
- 2.4 P4 Compiler (p4c) . . . 16
- 2.5 Behavioral Model (BMV2) . . . 17
- 2.6 P4 Runtime . . . 18

3 In-band Network Telemetry (INT) 20
- 3.1 Introduction . . . 20
- 3.2 Terms . . . 21
- 3.3 Reports . . . 23
- 3.4 Modes . . . 24
  - 3.4.1 Postcard Mode,INT-XD and INT-MX . . . 25
  - 3.4.2 Inband Mode, INT-MD . . . 25
  - 3.4.3 INT-Marked Synthetic Traffic . . . 28
- 3.5 Monitoring . . . 29
  - 3.5.1 Ingress Information . . . 29
  - 3.5.2 Egress Information . . . 30
- 3.6 INT Headers . . . 31
  - 3.6.1 INT-MD Metadata Header Format . . . 35
- 3.7 Related Work . . . 38

4 Design and Implementation 40
- 4.1 Queue Depth - Collision Detection . . . 41
- 4.2 Collision Detection and Load Balancing . . . 44
  - 4.2.1 Adding the Telemetry Header . . . 44
  - 4.2.2 Congestion Detector . . . 48
  - 4.2.3 "Load Balancing" . . . 49

5 Evaluation 50

6 Conclusion and Future Work 59

List of Figures

1 Software Defined Networking (SDN) Definition 4
2 Network Operating System (NOS) 5
3 Example of OpenFlow Paradigm 7
4 Designing NOS in a Fixed Data Plane VS a Programmable Data Plane . 10
5 P4 Typical Workflow 12
6 P4 Historical Road Map 13
7 P4 Program Sections 16
8 P4 Runtime Reference architecture[1] 19
9 Telemetry Report Modes 24
10 Postcard Mode 26
11 Inband (In-situ) Mode 27
12 INT-MD Metadata Header[2] 35
13 Linear Topology[3] 41
14 Topology 44
15 Packet from Switch 6/Port 5 46
16 Packet-Analysis from Switch 6/Port 5 47
17 Packet from Switch 6/Port 5 47
18 Packet-Analysis from Switch 6/Port 5 48
19 Queue depth and Latency 50
20 Switch:1 - Port:eth1 52
21 Switch:1 - Port:eth2 52
22 Switch:1 - Port:eth3 53
23 Switch:1 - Port:eth4 53
24 Queue depth and Latency - Comparison 54
25 Latency(ms) with and without INT-LoadBalancing 55
26 Switch:1 - Port: eth1 56
27 Switch:1 - Port: eth2 56
28 Switch:1 - Port: eth3 57

29 Switch:1 - Port: eth4 57
30 Switch1: Throughput Comparison 58

Listings

1 headers.p4 41
2 parser.p4 42
3 int1.p4 43
4 int.p4 Ingress Pipeline 45

List of Tables

1 INT Instruction Field Bit Mapping[2] 37
2 Switch Port Mapping 46
3 Switch1 eth1-eth4 ports Throughput 51
4 Switch1 eth1-eth4 ports Throughput with Load Balancing 55

1 Introduction

Software Defined Networking and OpenFlow reshaped the way of configure network forwarding devices and network behavior determination, enabling a "top-down" networking approach. This networking evolution though, was hampered by the inability to control or change the switch behavior that forces the operators to create systems with a "bottom-up" approach[4]. Software Defined Networking provide methods for network monitoring and management, but these techniques and methods increase the monitoring costs and suffer from high network overhead, incompatibility and low accuracy[5].

In the past, programmable switch chips could process packets in a very low rate compared to the fixed-function ASICs, but today we can find in the market as fast programmable switch chips as the fastest fixed-function switches. The introduction of a programmable chip (Protocol Independent Switch Architecture-PSA) that run as fast as a fixed function ASIC, opened a new era in networking, making it possible to specify exactly how the packets are being processed in the networking devices. What was missing though, was a language to be defined in order to program these fast switches or use the same language for programming the slower ones (e.g., FPGAs etc.) and software switches.

In 2013, Google, Microsoft, Intel, Princeton, Stanford and Barefoot set out to define P4, a programming language specific, to program the data plane of the programmable networking devices (e.g. switches, routers, etc.), indicating how to handle the packets, with a "top-down" approach in mind.

The ability to monitor networking devices in real time to improve network management is very important for the evolution of the Software Defined Networking. Various monitoring technologies were developed to monitor network traffic, including NetFlow-SFlow[6], OpenTM[7] and more. In-band Network Telemetry is a mechanism used to collect and report network state in real time from the data plane without the intervention of the control plane making it powerful and very fast. In this thesis we will design and implement a network monitoring and congestion avoidance, load balancing solution based on the P4 programming language using In-band Network Telemetry.

1.1 Motivation

Networking devices like SmartNICs, routers and switches are commonly designed with a bottom-up approach. P4 opened a new door providing a different approach similar to how the Graphics Processing Unit(GPU) work based in code similar to C language, same principals (compiled and loaded to the processor) but for networking devices. This is a top-down approach versus the bottom-up approach. P4 allows us to program the data plane and collect information in real time about the network state without using any traditional network tools (management plane or control plane). Using In-band Network Telemetry we can collect telemetry metadata for any packet specified, ingress and egress timestamps, latency, queue occupancy, etc. These metrics can be created by any networking device and send the INT information to an INT Collector (monitoring system).

This research was motivated by an active interest in investigating the capabilities of INT using P4 and the solutions to problems it can provide.

1.2 Objectives

The main objective of this thesis is to implement a network telemetry system using P4 language and present the new capabilities of emerging programmable switches to encapsulate processing metadata information. For this purpose, we demonstrate an implementation of In-band Network Telemetry using P4. The implementation is used to define custom headers, tables and processing logic in order to insert information regarding the network state (e.g. hop-by-hop delays) into the packets, for monitoring networks and services with high accuracy and level of detail[4]. Additionally to see how INT metrics affects the network performance, we will simulate two experiments. In the first we will apply a linear simple topology where we will demonstrate how to collect queue depth information from the switches. In the second experiment we will demonstrate how to apply logic on the switches using INT to avoid congestions and apply load balancing using a simple algorithm.

1.3 Structure of the Thesis

This thesis is organized in eight sections:

– *Section 1* presents the introduction to the thesis, the motivation and objective of the researcher and the structure of the thesis.

– *Section 2* covers the necessary concepts and tools used in the thesis.

– *Section 3* presents the In-band Network Telemetry in depth and discusses some related work.

– *Section 4* covers the implementation and design, aiming to give necessary technical details.

– *Section 5* presents the results of the In-band Network Telemetry implementation to avoid congestion with load balancing.

– *Section 7* presents the conclusion of this thesis and discusses direction for future work.

2 Programmable Networks

2.1 Software Defined Networking (SDN)

In spite of their widespread adoption, traditional IP networks, typically consists of network devices such as routers and switches and various types of middleboxes (i.e., firewalls) are complicated and very hard to manage[8]. To apply new high-level network policies, network administrators need to configure each network device separately, using low-level and often vendor-specific commands. Additionally, network environments have to encounter the dynamics of liabilities and adapt to load changes[9]. Traditional networks being vertically integrated, increase the complexity. The control plane (responsible to handle the network traffic) and the data plane (responsible to forward the traffic packets based on the decisions of the control plane) are packed inside the various network devices, hindering the flexibility, innovation and evolution of the network infrastructure.

Software Defined Networking (SDN) promises to significantly simplify network management by decoupling the forwarding hardware from the control decisions. SDN introduce new abstractions in networking, simplifying network management and promoting a centralized and programmable network management architecture as shown in Fig.1.

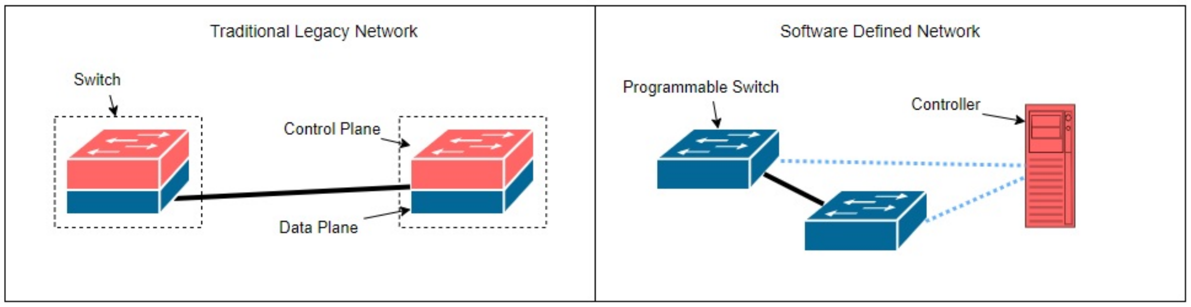


Figure 1: Software Defined Networking (SDN) Definition

Additionally to the control and data planes, there is also a third plane, the network management plane which manages the network policies, that simplifies the policy enforcement[9] as shown in Fig.2

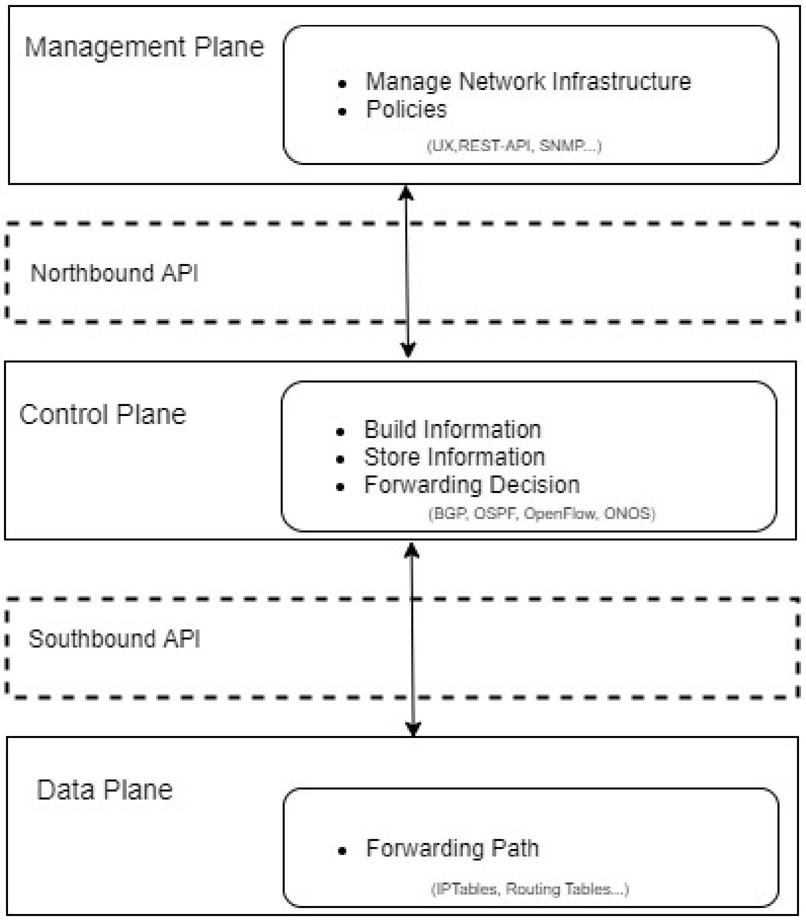


Figure 2: Network Operating System (NOS)

The Northbound and Southbound APIs between the planes, make the communication between the higher and the lower level components across the layers to be compatible. The Northbound API which is located between the management plane and the control plane, usually is provided by a REST API. The Southbound API which is located between the control plane and the data plane, allow the controllers (from the control plane) to communicate with the network devices on the data plane.

2.2 OpenFlow (OF)

The OpenFlow[10] standard is the result of a six year research collaboration between Stanford University and the University of California at Berkeley and allows users to manage network equipment using software that can run on servers that communicate with switches rather than directly on the switches or routers. This research began as a project in 2007[11].

The Open Networking Foundation (ONF)[12], is a non-profit consortium dedicated to the transformation of networking through the development and standardization of Software Defined Networking which brings direct software programmability to networks. This was launched in 2011 by Deutsche Telekom, Facebook, Google, Microsoft, Verizon and Yahoo. ONF is dedicated to rethinking networking and quickly and collaboratively bringing to market SDN standards.

In 2011, ONF announced the first standard communications interface defined between the control and forwarding layers of an SDN architecture, The OpenFlow which is probably the most popular Southbound API, defining the way the SDN controller interacts with the Data plane in order to make adjustments to the network, e.g., what kind of matching rules and actions would be applied on the incoming packets in the data plane switch, which determines what actions to be applied depending on the packet header fields matched.

It is important to clarify that OF is not SDN but it is merely one instantiation of SDN principles (other APIs could be used to control forwarding behavior[1], e.g., ForCES, representing other instantiations of SDN than differ from OF[11]). It is a protocol that extracts the control of a switch (packet routing, ACLs, etc.) from the switch and move it to a centralized server, allowing the switches and controllers to communicate to each other using the OF protocol. This network programmability independently from the baremetal networking devices, created a network system more flexible, agile and cost effective.

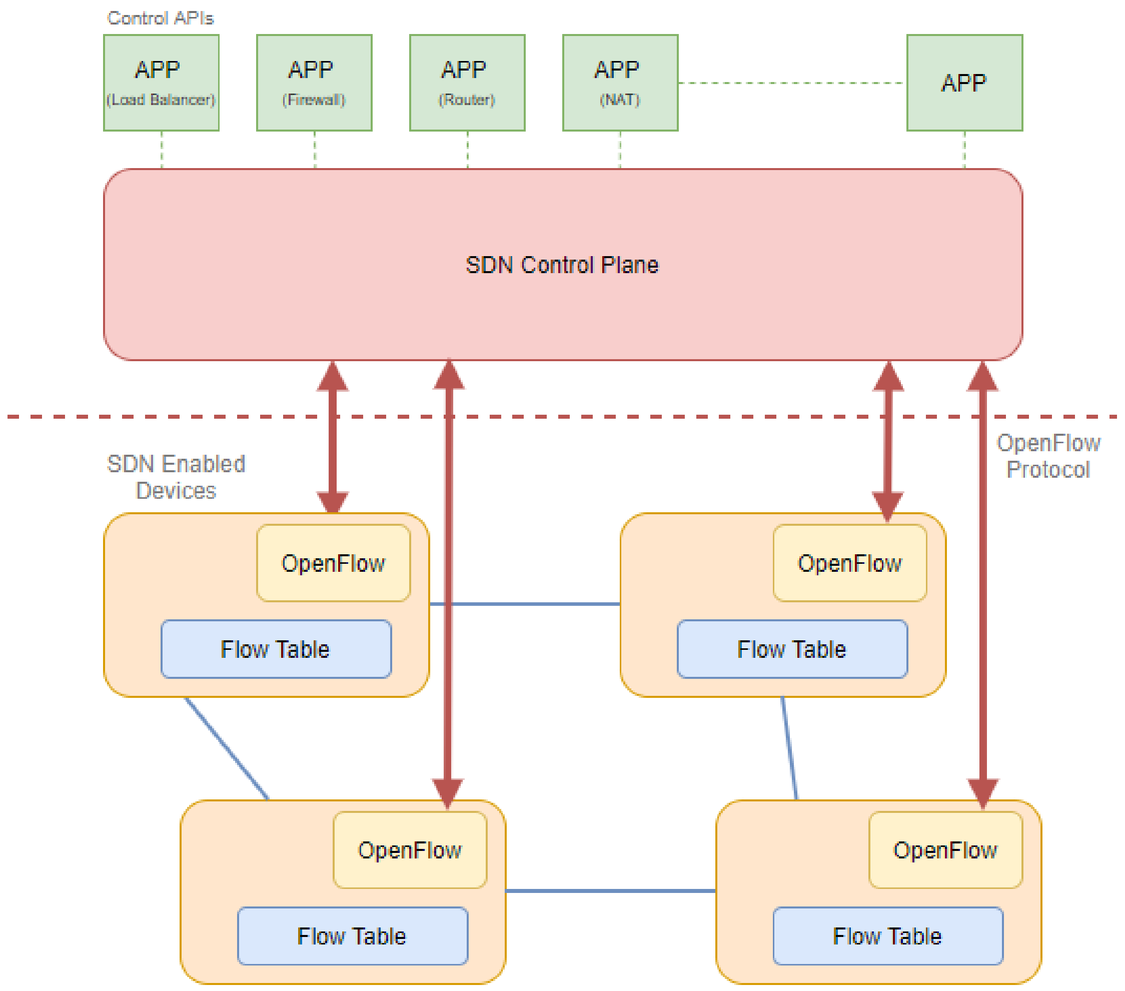


Figure 3: Example of OpenFlow Paradigm

[1]In the SDN terminology, the data plane networking devices are often referred as *forwarding devices*[13]

The topology in an OF is managed centrally while the data plane is still resides on the networking device (switch/router), while the control functions (high level routing decisions) are relocated to a separate controller.

In Fig.3 we can see the SDN controller that is going to interface to the SDN enabled networking device (switch or router) using this OpenFlow interface. The SDN controller is going to configure a flow table within the remote networking device, using standard off-the-shelf silicon or merchant silicon. Then we could make decisions on how to forward the packets based on their source and destination MAC addresses, their source and destination IP addresses, what action to take on specific TCP Ports and also we can maintain statistics. For example one function could be to switch or route based on the MAC address, so for a specified destination address we can configure to forward the packets to designated port and keep count of the packets going through that particular port and feed this information up to the central controller. We can also make decisions based on the IP address and more complex decisions using the destination address and possible the source MAC address to make decisions on what compromises of a flow and then make the decision on what action to take, like which port to forward that flow onto. Also we can have applications like Firewall to instruct the flow table to drop or forward the packets.

It is important to clarify that OF expects the switches to have a fixed behavior (not programmable switch chips) as described on their device datasheet and these high performance switch chips support a fixed set of protocols directly implemented IEEE and IETF protocol standards in silicon[14]. The behavior of these networking devices cannot be changed like adding new protocols or defining new ways of measuring and controlling the data path[2], for example in the first version of OF we could populate flow tables just for only 4 common protocols (Ethernet, IPv4, VLANs and ACLs). Although while the market demand was growing, more header types were added to OF counting today more than 50 different header types.

[2]It takes about 4 years to add a new protocol to a fixed function ASIC

2.3 P4 Language Specifications

2.3.1 $P4_{16}$ Overview

The name P4 comes from the paper which introduced the language "Programming Protocol-Independent Packet Processors"[15] and it is a high-level language expressing how the packets are being processed by the data plane of a programmable component such as software or hardware switch, router, network interface card or network appliance[16].

P4 is designed and used to define the data plane functionality of the target and not the control plane although it partially specify the interface with which the control plane and data plane communicate. P4 was designed with three main goals as described in[15]:

- **Reconfigurability.** The controller should be able to specify the packet parsing and processing.
- **Protocol Independence.** The network device (switch, router etc.) should not be tied to any specific packet format but the controller to be able to define both, a packet parser to extract header fields with specific fields types, names and a number of typed match and action tables that process these headers.
- **Target Independence.** The programmer of the controller should not need to know the details of the underlying network device but the capabilities to take in account when turning a target-independent description (programmed in P4) into a target-dependent program (used to configure the network device)[15].

In the past, the data plane was not a part of the network design and building a network was just designing the management and the control plane without having any control on how the packets were being processed since network devices had a fixed chip (Application-specific integrated circuit or ASIC), known as fixed-function devices. The differences between a traditional fixed data plane switch and a P4 programmable switch, as shown in Fig.4 is that in a traditional switch the manufacturer defines the data plane functionality and the control-plane controls the data plane by configuring specialized objects, managing entries in tables and processing control packets.

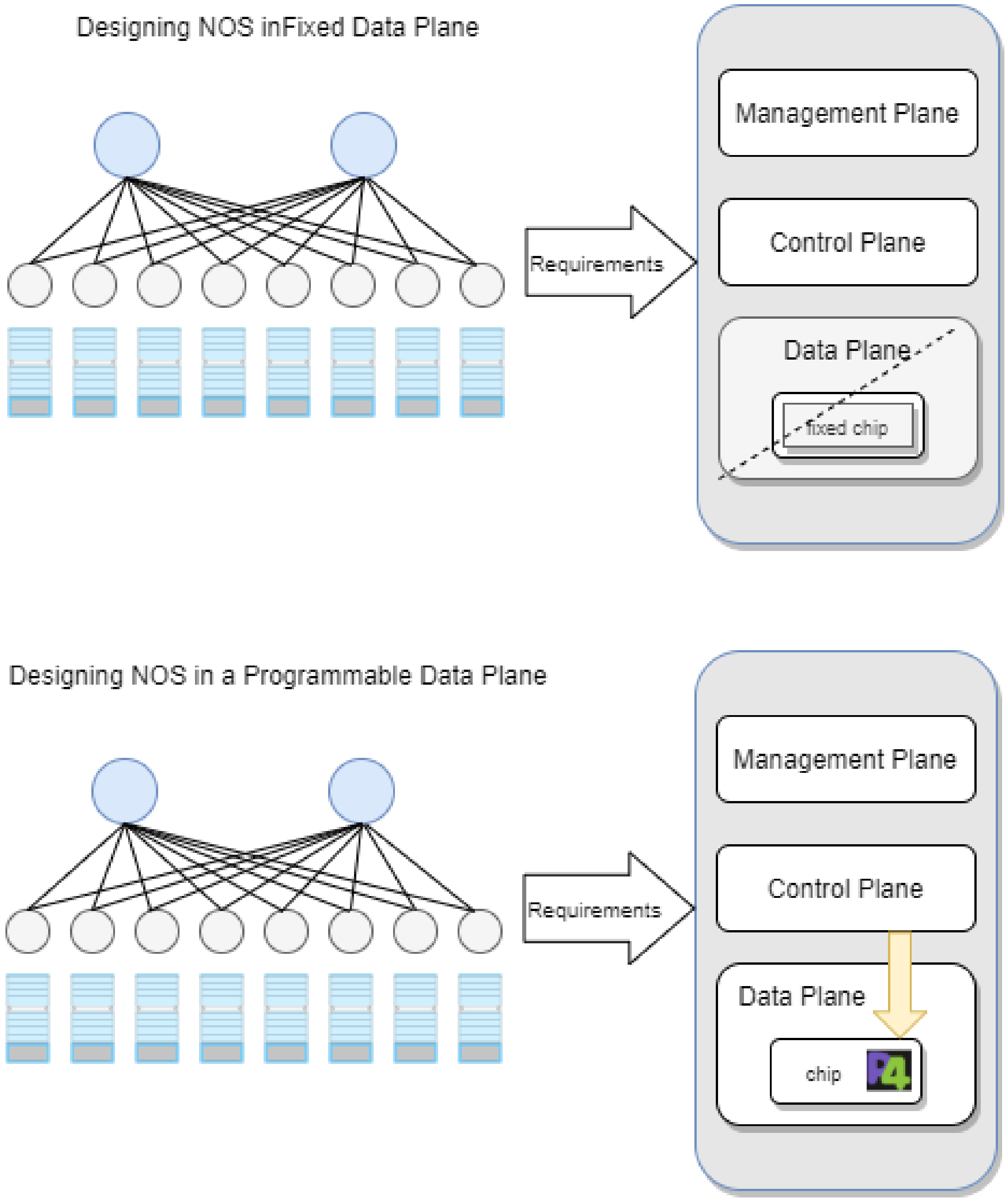


Figure 4: Designing NOS in a Fixed Data Plane VS a Programmable Data Plane

A P4 programmable switch (or network device) differs from the traditional switch (network device) in two fundamental ways:

- A P4 program specifies the data plane functionality which does not have any built-in knowledge of existing networks, also the p4 program is configured at initialization time in order to implement the functionality of the P4 program.
- The same channels are being used for the communication between the control plane and the data plane as in a fixed function network device but the set of tables and other objects are now specified by the P4 program. The P4 compiler generates the APi which the control plane use for the communication with the data plane[16].

P4 is protocol independent, enabling the programmers to express a big range of protocols and data plane behaviors while providing core abstractions as:

- Header types which describe the format of each header in the packet (fields and sizes).
- Parsers which describes the permission sequence of the headers in the received packets, identifying the header sequences and the header fields to extract from the packets.
- Tables, match and action tables is the mechanism that performs the packet processing. The P4 program specifies the fields on which the table may match and the actions to execute.
- Actions are pieces of program that describes how packet header fields and the metadata are being manipulated. Actions support data supplied by the control plane at runtime.
- Control flow, program that determines the order of match and action tables being applied to packets.

- Extern objects which are architecture specific constructs which can be manipulated by a P4 program through an API, but the internal behavior of the objects being hard wired (e.g., checksums) and not programmable with P4.

- User-defined metadata. Structures associated with packets.

- Intrinsic metadata, provided by the architecture combined with each packet.

In Fig.5 we illustrate a typical workflow of how to program a target - a switch model which is called Protocol Independent Switch Architecture (PSA)[17] - using P4.

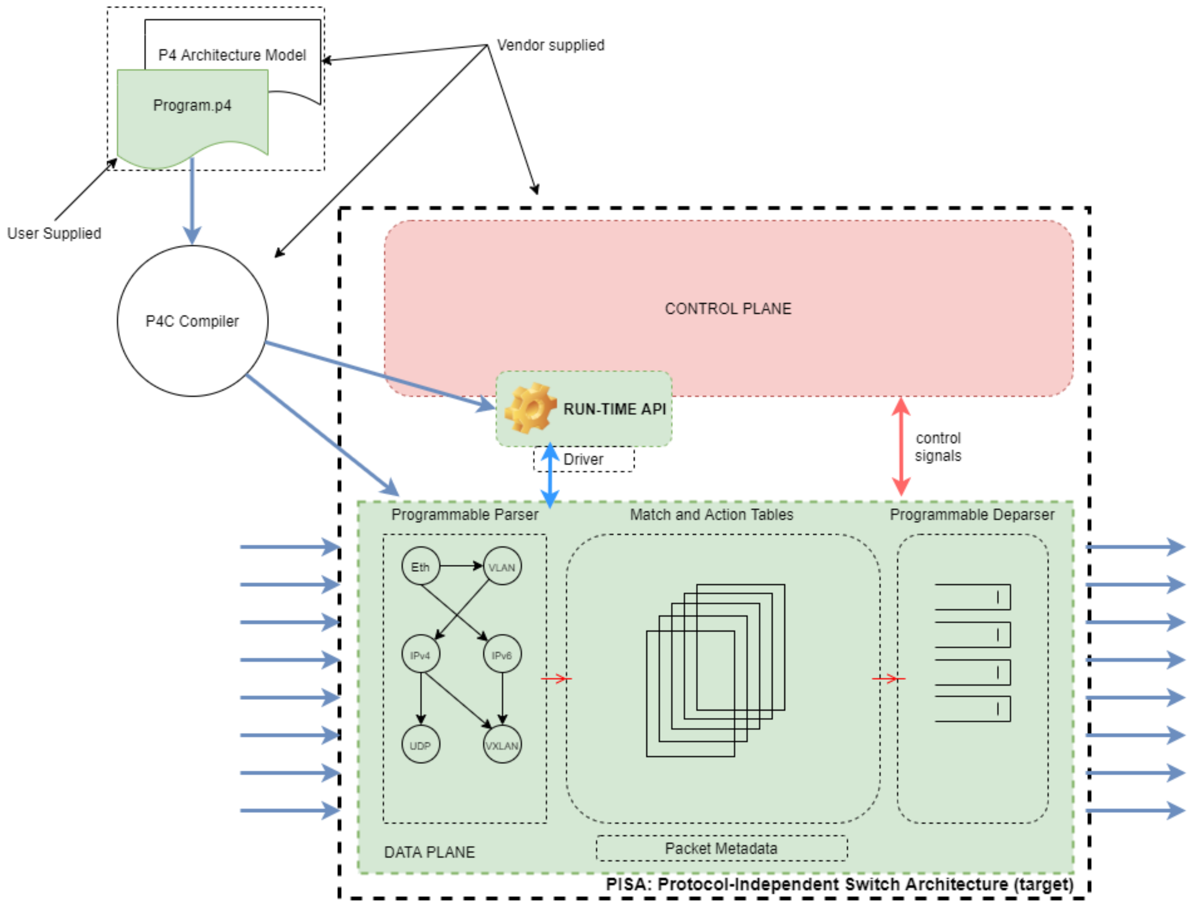


Figure 5: P4 Typical Workflow

In reality the target vendors provide the architecture definition and the p4 compiler specified for the target so the programmers can write specific code for this architecture and the specific P4 programmable components. When compiling the program, the compiler produce the data plane configuration which implements the forwarding logic and also produces an API to manage the state of the data plane objects from the control plane[16]. Also what we can see is that P4 is designed to program pipelines which have four components.

The first component, the parser is used when the packets comes in which are a series of bits or bytes. The parser is able to identify particular networking headers inside the sequence of bytes and convert it to what we call the parsed representation of the packet. The other component, the Match and Action tables are basic functions which map certain fields or headers into the outcome. These tables can be sequenced and as a result we can get fairly complex actions expressed in the language. The third component is the Deparser which function is the opposite of the parser. It takes the parsed representation of the packet and serialize it. The last component and the most important is the Metadata Bus. When the packet pass through the previous components, we need to pass the intermediate results and this can be achieved by caring the metadata (like a very wide register array). This is how the computations are passing from one stage to another.

P4 is a specific domain language, designed to be implemented on a big variety of network devices. Below in Fig.6 we can see a quick historical recap.

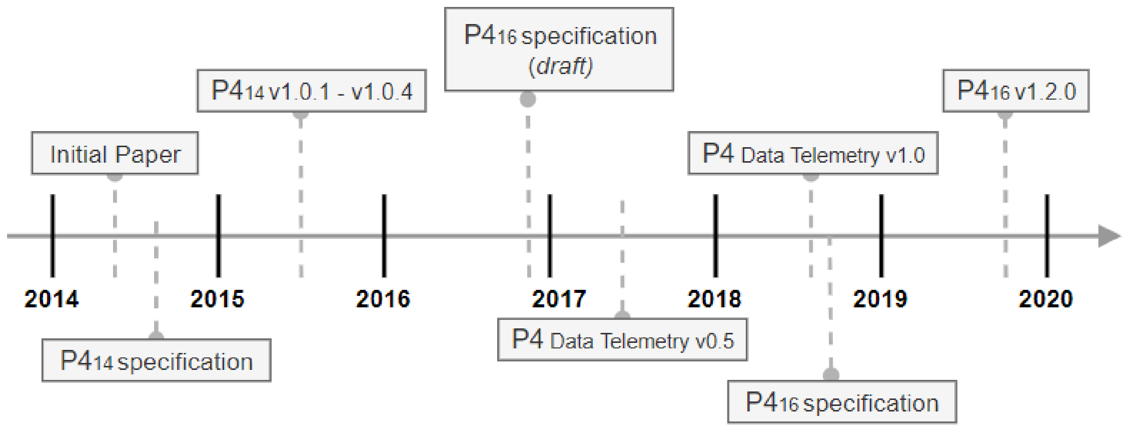


Figure 6: P4 Historical Road Map

2.3.2 $P4_{16}$ Benefits

Compared with fixed function network devices or with state of the art packet processing systems, P4 approach provides many advantages and benefits like:

- Full Control and Customization. Make the device, the execution of the programs , the algorithms and the processing of the protocols behave exactly as programmed.
- Exclusivity and Differentiation. Many customers requests exclusivity, meaning that they don't want to share their custom protocols, knowledge and know-how with the silicon vendors
- Additional new features. Since the device is programmable, additional new protocols can be added, replaced or removed according to the network or customer demands.
- Efficiency. The silicon devices ASICs have only finite amount of resources, fixed function devices, and when those functions are not in use (not needed) then a lot of resources are just either dead silicon, turned off but very often still leaking the energy. With the programmable device the resources can be dedicated only to the processes required. If not much processing is required then those resources can be easily shut off, achieving either the scale or energy efficiency or both.
- Reliability. Reducing the risk by removing unused features means less prone to exploits, bugs.
- Telemetry. Be able to see inside the Data Plane by mixing in the data headers processed with the intermediate results that occur when the packet is processed and easily export them through the packet headers. As a result it provides an insight into what is happening in the data plane at the data plane speeds[18].

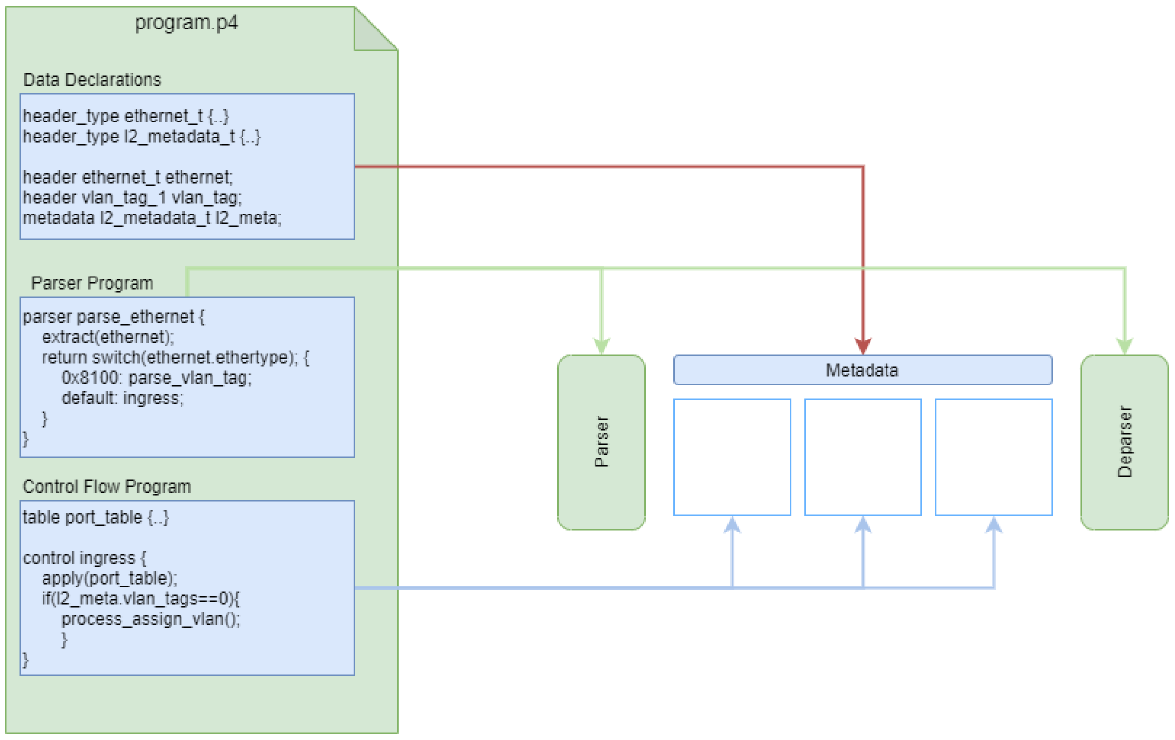
2.3.3 $P4_{16}$ Architecture

The $P4_{16}$ architecture specifies the P4 programmable blocks and their data plane interfaces, it is the programming model. Each network device vendor provides both a P4 compiler and an accompanying architecture which act as a contract-settlement between the program and the target. These specified by the vendors' architecture blocks (e.g. Ingress Port) are programmed by a separate chunk of P4 code with which the target interfaces over a set of signals and control registers. P4 programs can also work on the output controls (e.g. Egress port). The control registers and signals are defined in P4 as intrinsic metadata and the P4 programs have the ability to store and manipulate data related to each packet as it is defined by the user metadata.

To process a packet, the architecture model acts as an interpreter (architecture specific as specified by the vendor) of the bits written by the P4 program to intrinsic metadata.

The P4 programs can also call functions or services (e.g. calling a checksum function which is built-in on a target) that are executed by extern objects which are provided by the vendors' architecture.

As previous described, P4 programs are specific and not compatible between the different architectures, for example one P4 program that runs in one architecture which supports a vendors' custom control registers will not run on another architecture which will not have those control registers. In Fig.7 we can see an example of a P4 program and the blocks-program sections.



```
header_type ethernet_t {..}
header_type l2_metadata_t {..}

header ethernet_t ethernet;
header vlan_tag_1 vlan_tag;
metadata l2_metadata_t l2_meta;
```

```
parser parse_ethernet {
    extract(ethernet);
    return switch(ethernet.ethertype); {
        0x8100: parse_vlan_tag;
        default: ingress;
    }
}
```

```
table port_table {..}

control ingress {
    apply(port_table);
    if(l2_meta.vlan_tags==0){
        process_assign_vlan();
    }
}
```

Figure 7: P4 Program Sections

2.4 P4 Compiler (p4c)

P4C is a reference compiler for the P4 (both $P4_{14}$ and $P4_{16}$) and many backends (e.g. p4c-graphs, p4c-ebpf). In this thesis we will use p4c with bmv2, target the behavioral model[19]. The P4C compiler has an Apache 2 license and it is open-source. The main goals of the P4C compiler are:

- support current and future versions of P4
- support various back-ends (e.g., code generation for ASICs, SmartNICs, FPGAs, software switches and other networking devices)
- support for other tools (IDEs, control planes, etc.)
- the front end to be Open Source
- the architecture to be extensible

- using modern compiler techniques (immutable intermediate representation-IR, visitor patterns, etc.)
- comprehensive testing.

The P4 compiler generates two output files[]:

1. the target specific binaries which are used to realize the switch pipeline
2. the .p4info file, which is a target independent protobuf based file used as a schema of pipeline for the runtime control.

2.5 Behavioral Model (BMV2)

The Behavioral Model is a software switch written in C++11 which takes as input a JSON file generated from a P4 program by the p4c compiler in order to interpret it so to implement the packet processing behavior as defined in the p4 program[20].

The BMV2 switch is not made for production but to be used as testing tool in development and debugging P4 data planes and control planes written for the switch. The performance of the switch is less (throughput and latency) comparing a production switch like Open vSwitch, mainly the performance is impacted by:

- complexity of the P4 program (number of parsed-deparsed headers, match-action tables, etc.). Simpler P4 program equals more throughput and less latency.
- the compiler used (e.g., the legacy p4c-bm compiler produce faster JSON files than the p4c, although the p4c-bm is no longer maintained).
- the version of BMv2 running code.
- the flags used to build the BMv2.
- the command line options used when starting the BMv2.
- the number of table entries installed in the tables applied while processing the packets.

- the hardware performance (number of CPU cores, cache, etc.).
- the traffic flow pattern used.
- the machine which it will run (e.g., baremetal Linux or VM).

It is observed that running the BMv2 in a Linux VM over a hypervisor can cause a significant performance degradation due to under-utilization of the CPU indicating some I/O poor throughput.[21]

2.6 P4 Runtime

The P4Runtime API is a control plane specification for controlling the data plane elements of a networking device or program, defined by a P4 program[1]. It is an interface between a P4 Controller and a P4 programmable networking device which:

- is similar to OpenFlow
- is used to load a compiled P4 program into the switch silicon
- can add or delete flow entries in the match-action tables and
- collect statistics from the switch.

In Fig.8 we can see a P4Runtime Reference Architecture where the networking device (target) is a the bottom and the controller or controllers, at the top. In order for more that one controller to participate, a multi master protocol[1] is used, and with the roles assigned it is ensured that only one controller will have read access to any entity or at the pipeline config. The P4Runtime API is the interface between the server and the client(s) defining the semantics and messages as specified by the p4runtime Protobuf file. With the P4Info metadata we can declare the P4 entities which the controller can have access to. The controller also can set the "ForwardingPipelineConfig" with which it is installing and running the P4 compiled program which is included in the p4 device config Protobuf message field and also installing the P4Info associated metadata. In addition the controller can retrieve the device P4Info and config by a query to the networking device(target) for the "ForwardingPipelineConfig".

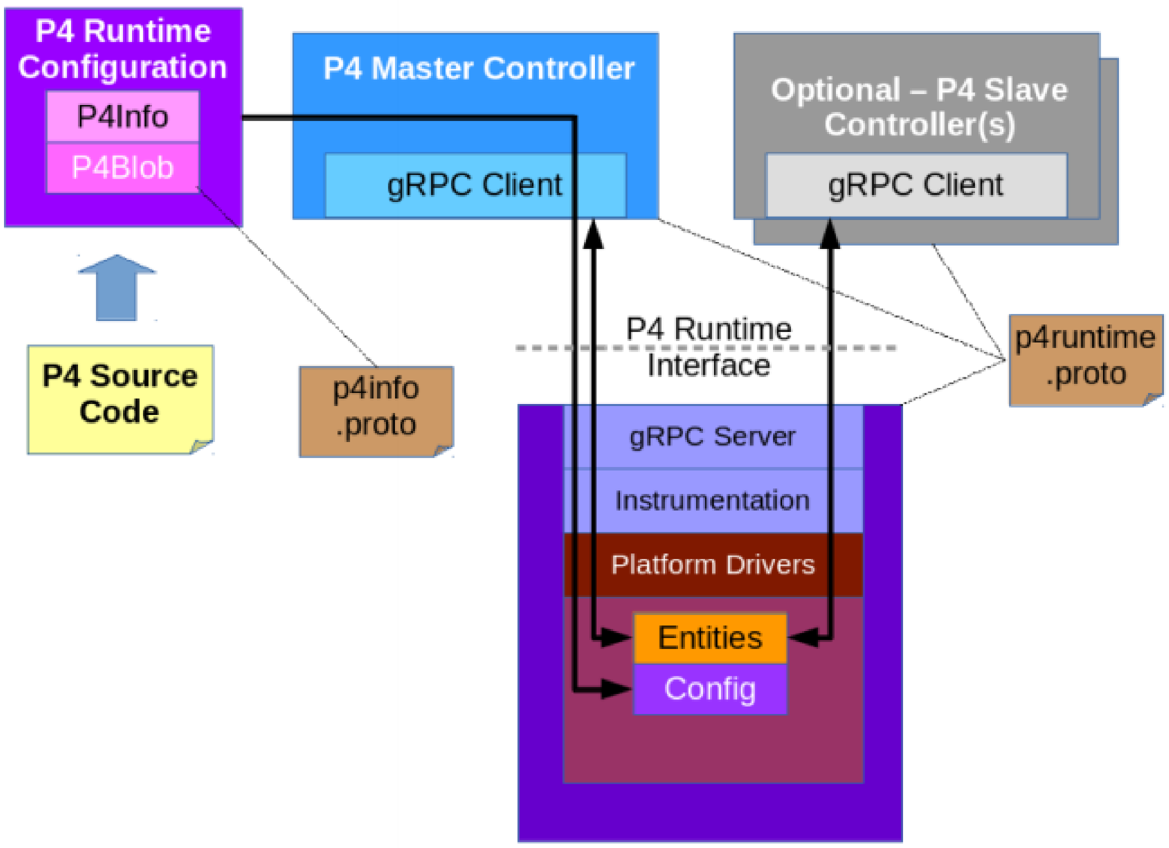


Figure 8: P4 Runtime Reference architecture[1]

3 In-band Network Telemetry (INT)

3.1 Introduction

Inband Network Telemetry (INT)[22] is a mechanism for collecting and reporting real time network state (INT-metadata) directly in the data plane. The control plane is used only for decision making on what information the data plane will collect and for which flows. The packets that contains the header fields are interpreted from the P4 Programmable networking devices as "telemetry instructions"[22] and specifying the instructions each device should collect or write into the packet as it flows through the network.

The "telemetry instructions" can be embedded in normal packets, cloned packet copies or in special probe packets from the INT sources network devices (e.g. Smart NICs, ToRs, applications, etc.) The instructions can be programmed accordingly in the data plane to match on network flows in order to execute specified commands on these matched flows and then order each INT capable networking device to define what network state will collect. The network state data of each networking device can be collected-exported straight from the data plane to the Telemetry reporting and monitoring system or the INT capable network device can just write the network state data in the packet while the packet moves over the network. Finally, the INT Sink networking device fetch the INT information-results from the instructions and forwards the precise data plane state, as described with the packets, to the monitoring and reporting (optional) device.

Below we can see some examples of INT traffic sink functions:

- OAM (Operations, Administration and Management): the INT sink network device first collects the information from the packets which describes the network state and then export the network state to a controller. The export can be either in a raw format or can be processed (e.g. compressed, de-duplicated..)

- Real time control: the INT sink network device will use the data plane information in oder to provide back control information to the INT Source network devices, which in turn will use this information and make modifications to the network traffic or the packet forwarding. For example this can be used as a precise congestion notification system)
- Network Event Detection: The INT sink network device can generate instantaneous actions to react to the network events if the collected network path state shows marks of a situation that needs resolution. For example, an intense congestion can cause this and the INT sink network device might form a feedback control loop in a centralized or decentralized form.

Some interesting high level applications that can be empowered with the INT architectural model are:

- Network troubleshooting and performance monitoring
- Micro-burst detection and packet history
- Advanced congestion control
- Advanced routing[2]

3.2 Terms

- Monitoring System: A reporting system that gathers all the telemetry information sent from various INT capable network devices. The monitoring-reporting system parts can be physically distributed but logically centralized.
- INT Header: is the packet header that carries the INT data. There are three types:
  * embed data or MD-Type

    * embed Instruction or MX-Type, and
    * Destination type (*Details for the INT headers in 3.6*)

- iNT Packet: is the packet that contains the INT Header.
- INT Instruction: are the instructions pointing which INT Metadata each INT networking device will collect. Instructions can be configured at each INT capable networking device (Flow Watchlist) or can be written on the INT header.
- Flow Watchlist: a data plane match-action table which matches the packet headers and apply INT data at the matched flow(a set of packets that have identical values at the selected header fields)
- INT Source: the networking device that apply and inserts the INT Headers into the packets. (A Flow Watchlist is configured to choose the flows to insert the INT headers in.)
- Int Sink: it is the networking device that pulls out the INT headers from the packets and gather up the network path state which is included in the INT Headers. The INT Sink networking devices are trusted for the extraction of the INT headers and also for making INT mechanism transparent to the upper layers. The INT Sink networking device, after extracting the INT information from the packets, it sends it to the monitoring and reporting system.
- INT Transit: is the networking device which gathers the metadata from the data plane as specified in the INT instructions. According to the INT Instruction, the information can be exported straight from the data plane to the Telemetry reporting and monitoring system or just update the network state data in the INT Header while the packet continues to move over the network.
- One device can be at the same time an INT Source, Transit or Sink meaning that it can embed metadata into the packets as INT Source, for the same flow and act also as an INT Transit for the same or different flow.

- INT Metadata: Is the data which the INT Source and Transit networking devices add in the INT Header.

- INT Domain: a group of INT devices that shares a common administration. Various networking devices (e.g. switches, etc.) with various packet header formats within the same domain are configured in such a way to ensure interoperability between the INT networking devices. Administrators are configuring the INT Sink networking device at the domain edges in order to avoid INT data to leak out form the domain.

3.3 Reports

While there are many reasons why network Administrators or Operators would want telemetry reports, there are three main categories why to create one:

- Tracked Flows: A telemetry report can be created by the INT system while tracking the rules matched by the packets of a flow[23] (the Flow Watchlist does the matching) while the packets traveling through the network, at real time. So the telemetry report is being generated from the packets that matches the flow watchlist. Also the telemetry report includes information related to the path that the packets followed such as hop latency and queue occupancy.

- Dropped Packets: When a drop watchlist is applied, a telemetry report can be generated from all the dropped packets that match with the watchlist. This functionality offers a transparency of the packet drops.

- Congested Queues: A telemetry report can be generated from the packets that enters a queue through the period of queue congestion. This functionality offers a transparency in the network traffic in order to prolong queue congestions when for instance in the network we have large elephant flows[24] which can suppress a queue and in the same time reduce the traffic of the mice[24] flows by the congestion. Microburst origin from many mice flows that arrive in the queue at the same time is also another problem and this additional information can also be present in the telemetry report by defined association bits.

The networking devices must be configured to discover when to create the telemetry reports and also what information the reports will contain. The telemetry reports are triggered from the networking devices while packet processing although not all packets are triggering a telemetry report no matter if matching the watchlist, as networking devices can apply filters to specify if the occured events are important to create a report or not. This is known as "*Event Detection*[25].

3.4 Modes

Data packets are being monitored and can be embedded with INT instructions and metadata. There are three modes of Inband Network Telemetry based on the level of packet variation as show in Fig.9

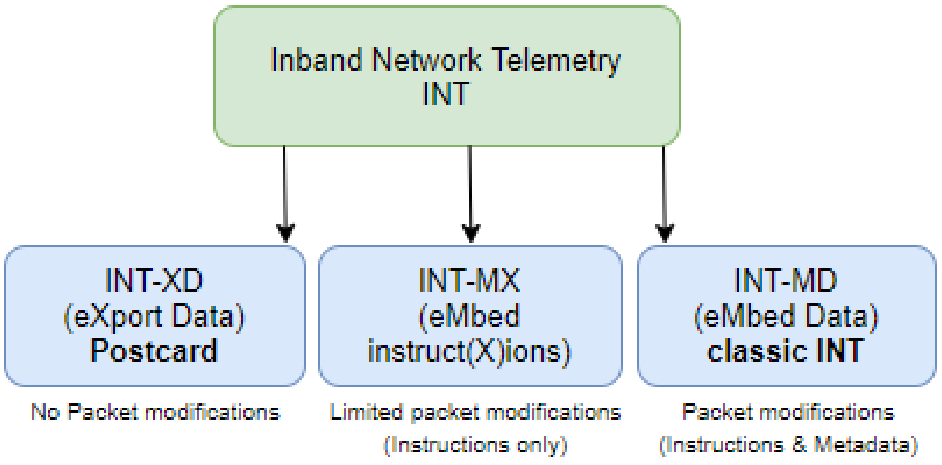


Figure 9: Telemetry Report Modes

3.4.1 Postcard Mode,INT-XD and INT-MX

In the postcard mode as was specified in the previous telemetry report specification, now called INT-XD, each networking device is exporting metadata straight from the data plane and based on the INT instructions as configured on their Flow WatchList[16], to the telemetry monitoring system without any modification of the packets, see Fig.10. The Monitor is getting the reports from all the INT capable networking devices and the information contained in the metadata can be the Switch ID, Port ID or latency for each hop.

In the INT-MX (embed instructions) the INT Source networking devices embed INT instructions in the packet headers and then each of the following either INT Source or INT Transit networking devices directly sends the metadata to the monitoring system, being complied with the INT instructions embedded in the packets. The INT Sink networking devices at the end are stripping the instruction header before it forwards the packets to the end host. The packet size stays the same while the packet travels through the networking devices and the modification of the packet is fixed to the instruction header. This mode is inspired by IOAM's \Immediate Export"[26].

3.4.2 Inband Mode, INT-MD

In the third telemetry mode, the telemetry metadata is embedded in between the original headers of the data packets as they travel the network. This can be achieved using any of the telemetry data plane specifications such as INT or IOAM(In-situ Operations, administration, Maintenance)[26]. INT-MD (embed Data) mode is the classic hop-by-hop INT where:

- INT Source networking device embeds INT instructions.
- INT Source and Transit networking device embed INT metadata, and

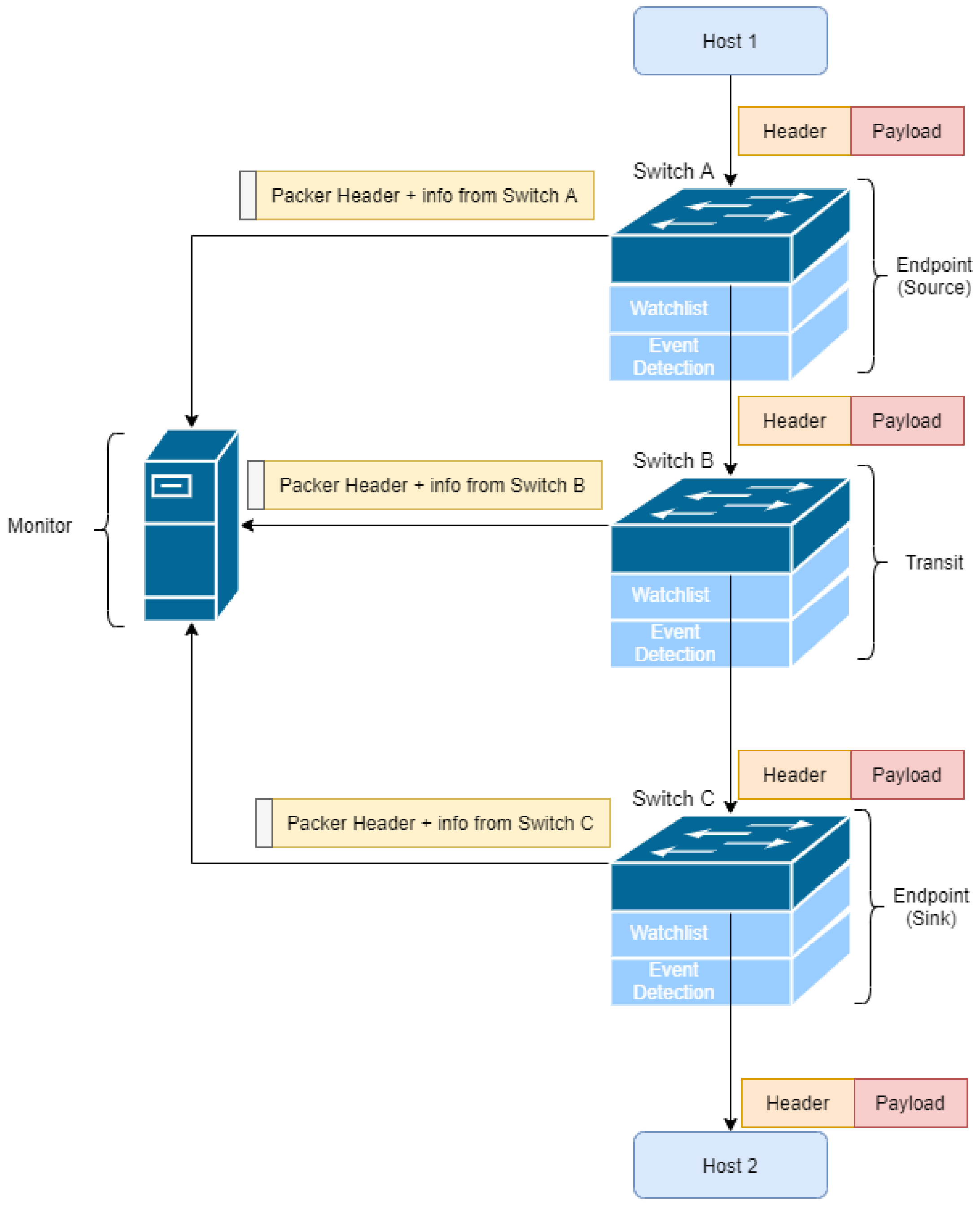


Figure 10: Postcard Mode

- INT Sink networking device strips the instructions and the total metadata out of the packet and send the data to the telemetry monitoring system. The packet size is increased and modified in this mode whereas it reduces the overhead at the telemetry monitoring system to collect reports from multiple networking devices.

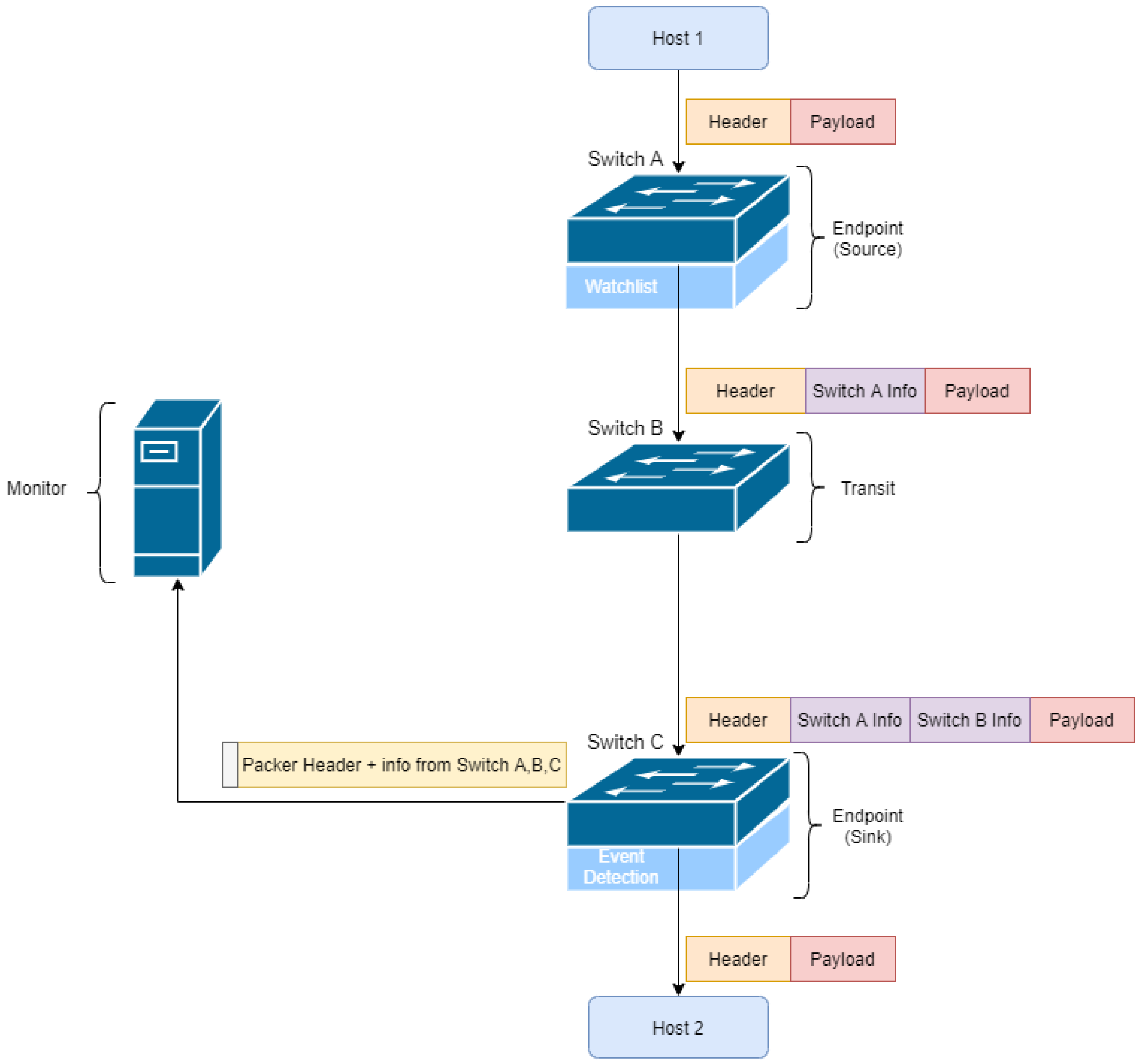


Figure 11: Inband (In-situ) Mode

When the packet enters in the INT Source networking device (Switch A), as shown in Fig.11, the INT instructions are embedded to the packet and in each hop the INT transit networking devices are adding the INT metadata to the packet. In the end, the INT Sink networking device will extract the INT metadata from the packet before sending the packet to the final destination and according to the "Event Detection" it will generate the telemetry report which it will contain all of the telemetry information from all the INT networking devices. To reduce complexity at the INT Sink networking device, a mixture of telemetry metadata embedded with the original packet headers is being used for the telemetry reports, a portion of which is specified in the INT or

IOAM telemetry data plane specifications. By reducing the data plane complexity, all the telemetry reporting can be done in the data plane without the need of the control plane. The INT Sink networking device can select to add its local telemetry metadata in the telemetry report header or in the embedded metadata mixed with the original packet headers.

3.4.3 INT-Marked Synthetic Traffic

INT Source networking devices can generate INT marked synthetic traffic in two ways. First by cloning the original data packets and second by creating special probe packets. The INT Transit networking devices are applying the INT to the synthetic traffic like the live traffic, the difference though is that in the synthetic traffic, the INT Sink networking devices will discard all network synthetic traffic after extracting the collected INT information, in contradictory in live traffic it will forward the traffic. All INT capable networking devices can be used on these synthetic and probe packets as defined by the INT Source networking device.

In particular the INT-MD mode applied to Synthetic/probe packets allows functionality comparable to IFA[27] (Inband Flow Analyzer - method to collect data on a a per hop basis across a network and perform localized or end to end analytics operations on the data). It is expected that the INT Sink networking device will discard the synthetic traffic and mark the probe packets for forwarding or discarding determined by the use case, although the INT Source networking device will have to define what the INT Sink networking device will do, forward or discard the packets after the INT data extraction.

3.5 Monitoring

With P4 and INT we have the ability to define and also collect any switch internal information, e.g., the **Switch id:** the distinctive-unique ID of a switch within the INT domain as assigned by the administrator.

In this section we will define a small set of metadata which are available on bigger range of networking devices because the exact definition of the metadata below can vary from one device to another and from one vendor to another due to the heterogeneity of the device architecture, resources, etc. Therefore, specifying the precise meaning of each metadata is beyond the scope of this thesis. Rather we conclude that the semantics of each metadata per networking device model used is communicated with the entities analyzing and interpreting the reported information in an out of the band way.

3.5.1 Ingress Information

- **Ingress port**: Using the ingress_port we can monitor the port on which the INT packets were received. The packets could be received on a random stack of port set ups starting from a physical port. For instance the physical port can belong to a link aggregation port group that may be part of a Layer 3 Switched Virtual Interface while at Layer 3 this packet could be received through a tunnel. The INT v.2 specifications[2] allows to be monitored up to two levels of ingress port identifiers, although the whole stack can be monitored (in theory). A 16-bit field is enough for monitoring the physical port on which the packets were received at the first level of ingress port identifier. For monitoring the logical port where the packets were received, a 4-Byte field is engaged by the second level of ingress port identifier. This 32 bit space in the second level identifier is enough to allow a big number of logical ports at each networking device. The port identifiers semantics can be different across the networking devices and each INT hop choose between the two levels and the port type to report.

- **Ingress timestamp**: The networking device local time when the INT packet is received at the ingress port (physical or logical).

3.5.2 Egress Information

- **Egress port**: Using the egress_port we can monitor the port on which the INT packets were transmitted. The packets could be transmitted on a random stack of port set ups ending in a physical port. For instance, the packet could be sent on a tunnel, from a Layer 3 Switched Virtual Interface and on a link aggregation group and out of a physical port which it belongs to this link aggregation group.The INT v.2 specifications[2] allows to be monitored up to two levels of egress port identifiers, although the whole stack can be monitored (in theory). A 16-bit field is enough for monitoring the physical port on which the packets were sent at the first level of egress port identifier. For monitoring the logical port where the packets were transmitted, a 4-Byte field is engaged by the second level of egress port identifier. This 32 bit space in the second level identifier is enough to allow a big number of logical ports at each networking device. The port identifiers semantics can be different across the networking devices and each INT hop choose between the two levels and the port type to report.
- **Egress timestamp**: The networking device local time when the INT packet is transmitted at the egress port (physical or logical).
- **Hop latency**: It is the time that it takes for an INT packet to be "switched" inside the networking device.
- **Egress port TX Link utilization**: The current utilization of the egress port TX Link from which the INT packet was transmitted. The networking devices

can use various methods to retain the current state, like moving average or bin bucketing. For example, although moving average is superior to bin bucketing, the INT Framework is leaving the networking device vendors to select the methodology they prefer.

- **Queue occupancy**: The accumulate traffic in the queue, which can be in bytes, cells or packets that the INT packet monitors in the networking device while traverse the network. The 4-octet metadata format is switch vendors specific and using the data language model YANG[28] we can specify the format and the units of the fields in the metadata stack[3].

- **Buffer occupancy**: The accumulate traffic in the buffer, which can be in bytes, cells or packets that the INT packet monitors in the networking device while traverse the network (Practical when the buffer is shared among many queues). The 4-octet metadata format is switch vendors specific and using the data language model YANG we can specify the format and the units of the fields in the metadata stack.

3.6 INT Headers

Three types of INT Headers exist: MD-type, MX-type and Destination-type. A given INT packet can carry either headers of type MD and MX and/or type Destination. If both types of INT Headers are present, the deader of type MD / MX must go before the deader of type Destination.

- **MD-type**: This type of INT Header must be processed by intermediate devices (INT Transit). This header format is defined in section 3.6.2 and is the scope of the experiment of this thesis.

[3]YANG model details: https://github.com/p4lang/p4-applications/blob/master/telemetry/code/models/p4-dtel-metadata-semantics.yang

- **Destination-type**:
  - This type of header is consumed by the INT Sink networking device while the intermediate (INT Transit) networking devices ignore such type.
  - Using the Destination type of header we can enable Edge-to-Edge communication between the INT Source and INT Sink networking devices. For example:
    * INT Source networking device could add a sequence number to detect INT packet loss.
    * Assuming that each of the networking devices is a L3 Hop, the INT Source can add the original IP TTL (Time To Live) and INT Remaining Hop Count values, allowing the INT Sink to discover networking devices that are not compatible with INT by comparing both decrements of IP TTL and INT Remaining Hop Count.
  - The Destination-type headers format is not yet defined and some Edge-to-Edge issues can be addressed by "source-only" metadata which is part of the the Domain Specific Instructions in the MD type Header.
- **MX-type**: The processing of this type of INT Header must be done by the intermediate networking devices (INT Transit) and generate reports to the monitoring system as directed.

Each INT capable networking device while forwarding the packets in the network paths, create extra space in the INT Header when required to add the INT metadata information. By configuring the Remaining Hop Count field in the INT Header we can limit the number of total INT metadata fields applied by the networking devices allowing in such way the avoidance e.g., of exhausting header due to a forwarding loop.

While the packet flows through the INT networking devices, each hop generates extra space in the INT Header in order to add the metadata information hence the packet size increases causing the egress link MTU to overcome at an INT switch. Solutions for this problem are:

- The MTU for links between INT Sources and INT Sink networking devices is suggested to be configured to a higher value for previous links. Setting a difference in bytes configuration of MTU, based on moderate values of Per-Hop Metadata length and total number of INT hops, we can avoid the exceeding of egress MTU while INT metadata inserted at each INT hop.
- A dynamical discovery of the MTU path for flows that are monitored by INT can optionally be carried out by the INT source or INT transit networking devices by transmitting ICMP messages via the Path MTU discovering mechanisms from the corresponding L3 protocol[4]. A moderate MTU may be reported in the ICMP message by an INT Source or INT Transit networking device, provided that the packet goes beyond the maximum number of permitted INT hops, checking for the additional metadata inserted at all INT hops assuming that at all the downstream INT hops that the egress MTU is the same as its own egress link MTU. Following this way, the path MTU discovery source will converge to a path MTU estimate faster but moderate.

Each INT Transit networking device regardless if it will participate in the path MTU discovery, if it cannot add the specified metadata information to the packet because it will increase the length to exceed the egress MTU, then it will have to indicate that the egress MTU is exceeded by setting the M bit (M bit described in Section 3.6.2) of the INT header at each INT hop. Also if the INT Source networking device will add the fixed headers (12 bytes) and also the Per-Hop metadata length*4 bytes of its own metadata[29], this can exceed the egress link MTU and thus INT cannot be be available for these packets. The INT Source networking device will have to set the M bit in the INT header if it will be programmed to add its own metadata information on the packet with the fixed INT headers but without extra space for the INT metadata information.

Theoretically, the INT Transit networking device while adding its own metadata

[4]RFC 1191 for IPv4 and RFC 1981 for IPv6

information, is performing IPv4 fragmentation but this can impact applications adversely. IPv6 packets though, cannot be fragmented with INT Transit networking devices.

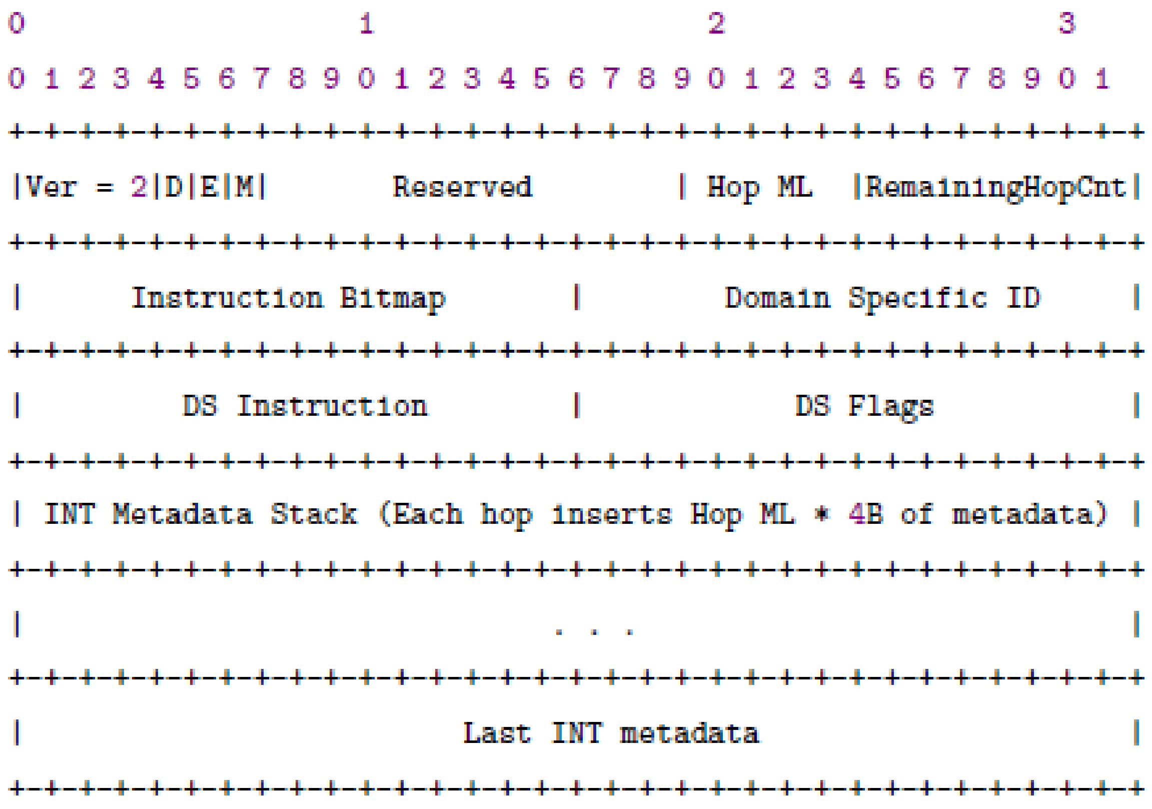
Intentionally the specific location for INT Headers is not specified. An INT Header can be added in a packet as option or payload of any type of encapsulation. What is required besides the sufficient space for the INT metadata information, is an agreement between all the INT networking devices (Source, Transit and Sink) on the location of the INT Headers.

Below we can see some encapsulations that use a normal protocol stack. Beside the options below though, we can select any encapsulation which fits better to our development needs.

* INT over VXLAN
* INT over Geneve
* INT over NSH
* INT over GRE
* INT over TCP
* INT over UDP

3.6.1 INT-MD Metadata Header Format

In Fig.12 we can see the INT-MD Metadata Header Format.



```
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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver = 2|D|E|M|       Reserved        | Hop ML  |RemainingHopCnt|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Instruction Bitmap       |       Domain Specific ID      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         DS Instruction        |            DS Flags           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| INT Metadata Stack (Each hop inserts Hop ML * 4B of metadata) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             . . .                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Last INT metadata                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 12: INT-MD Metadata Header[2]

The INT header has a length of 12 bytes followed by an INT metadata stack. Each metadata length is either 4 bytes or 8 bytes long. The same metadata length is applied to each INT hop so the total length of the metadata stack varies as different packets can pass through various paths and consequently, different INT hops. The INT metadata header fields as described in the specifications[2]:

- Ver (4b): is the INT metadata version which should be 2 for this version.

- D (1b): this is the Discard or Clone bit. The INT Sink networking device is discarding the packet after the INT data extraction.

- E (1b): Max Hop Count exceeded.
    * this flag is set when a networking device cannot add its own metadata because the Remaining Hop Count is reaching zero.
    * INT Source networking device must set the E bit to 0.
- M (1b): MTU exceeded:
    * When a networking device cannot add all the requested metadata, then the M bit must be set in the INT Header in order to avoid the packet length to exceed the egress link MTU. By doing so, the networking device is not adding any INT metadata to the packet. The networking devices use this flag as an indication that INT metadata applied or not at one or more devices, in order to proceed in MTU reconfiguration at some links, especially when the INT networking devices are not participating in the MTU path discovery. The purpose of the M bit is not to identify which networking device(s) has not added the INT metadata due to egress MTU limitation.
- Reserved (12b): Reserved bits
- Hop ML (5b): Per-hop Metadata Length which includes the Domain Specific Metadata(in 4 Byte words) added at every INT Transit hop.
- Remaining Hop Count (8b): The remaining number of hops that are allowed to add their INT metadata to the packet.
- INT instructions are encoded as a bitmap in the 16-bit INT Instruction field and each bit corresponds to a certain metadata, as show in Table.1:
- Domain Specific ID(16b): It is the unique ID of the INT Domain.
- DS Instruction (16b): It is an instruction bit map particular to the INT domain identified by the Domain Specific ID.
- Every INT Transit networking device, when supported, is adding its own metadata information as defined in the DS instructions and instruction bitmap right

bit0 (MSB)	Switch ID
bit1	Level 1 Ingress Port ID (16bits) + Egress Port ID (16bits)
bit2	Hop Latency
bit3	Queue (8 bits) + Queue occupancy (24 bits)
bit4	Ingress timestamp (8 bytes)
bit5	Egress timestamp (8 bytes)
bit6	Level 2 Ingress Port ID + Egress Port ID (4 bytes each)
bit7	Egress port Tx Utilization
bit8	Buffer ID (8bits) + Buffer occupancy (24 bits)
bit15	Checksum Complement
	The Remaining bits are reserved

Table 1: INT Instruction Field Bit Mapping[2]

after the INT metadata header. If an INT Transit networking device will not add metadata to a packet due to reasons as specified in[20], then it must not decrement the remaining INT hop count in the INT metadata header.

- The INT metadata stack length, in bytes, should always be a multiple of "HOP ML * 4" plus (if added by the source) the size of 'source-only' Domain Specific Metadata. By subtracting the total (12 bytes) INT fixed header sizes from the "shim header length * 4" we can determine the total stack length.

In summary the INT Source networking device must :

∗ set the Ver, D, M, Hop ML, Remaining Hop Count, and Instruction Bitmap.

∗ set all reserved bits to zero, and

∗ set the Domain specific fields

and the intermediated transit networking devices can set the following fields:

∗ E, M, Remaining Hop Count, Domain specific fields.

3.7 Related Work

Other work related to data plane programmability and implementation of network telemetry systems with P4 for monitoring include:

- Design and implementation of network monitoring and scheduling architecture based on P4[30] which monitors the network state information, like switch id, queue occupancy and latency without adding additional detection packets while implementing visualization from the network state information metadata. Monitoring of the network is in real time based on the network state.

- Fast network congestion detection and avoidance using P4[31], proposed a P4 based technique which enables the measuring delays and rerouting at the data plane. This has two advantages, one is that the reduction of the congestion detection time and detection per flow meaning that only the affected flows will be rerouted. Second the reaction time is also reduced on the local control, after the detection and no new flow rules required to be installed reducing the load on the central controller. Additionally the solution can be extended providing a bandwidth reservation and priority queuing.

- Towards Knowledge-Defined Networking using In-band Network Telemetry[32], proposed as the number of networking devices increase rapidly and network management becomes more difficult and complex, a closed loop network management might be a solution addressing this problem using network telemetry, Software Defined Networking (SDN) and Knowledge Defined Networking (KDN) to create a self driving network.

- Language-Directed Hardware Design for Network Performance Monitoring[33], proposed a performance query language, Marple which uses INT like performance metadata. Also presented a Marple compiler to target the P4 programmable networking devices and run queries enabling the network operators to quickly diagnose problems and improve visibility, demystifying the network performance for the applications.

- INT monitors all existing packets, meaning significant overhead can be caused by per packet INT header that can overload the INT monitoring engine. Selective In-band Network Telemetry for Overhead Reduction[34], to avoid this issue, proposed a selective INT (sINT) where the ratio of the packets to be monitored can be adjusted according to the frequency of the significant changes in the network.

- Towards ONOS-based SDN monitoring using in-band network telemetry[35], proposed an INT architecture for the UDP in Open Network Operating System (ONOS) controller, introducing the first INT monitoring system in ONOS.

- A Multi-Feature DDos Detection Schema on P4 Network Hardware[36], proposed an in-network architecture for DDos Attack detection using important traffic metrics of malicious traffic to identify packet anomalies. When identified, alarms are triggered and transmitted to the external mitigation systems. The DDoS detection schema applied in P4-enabled SmartNICs, using real time publicly available traces and high speed packet generators.

- Telemetry-Driven Optical 5G Serverless Architecture for Latency-Sensitive Edge Computing[37], proposed serverless subfunctions which are latency sensitive, optimized and deployed at cloud and edge based on telemetry data retrieved from a 5G transport infrastructure providing extremely fast invocation time (less than 450ms).

- Detection of Fog Network Data Telemetry Using Data Plane Programming[38], proposed a software defined architecture using a programmable data plane and P4, to describe data plane abstraction of Fog networking devices and collect telemetry data based on In-band Network Telemetry (INT), without the intervention of the control plane or additional overhead, within a Mininet simulation.

4 Design and Implementation

Equal-Cost Multi-Path Routing (ECMP) is used to load-balance traffic across multiple parallel paths uniformly. Although ECMP is not taking into account the size of the flows, for example the elephant and mice flows. Even if the flows will be uniformly distributed over multiple paths, this can cause congestion if the elephant flows would be concentrated on the same path. We will take advantage of the P4 Inband Network Telemetry which allows us to read queueing information, like the number of the packets waiting to be transmitted-queue occupancy, and use this information to identify congestion (i.e. if the queue size is big) and move the flows that suffer from congestion to another path.

To avoid the congestion, we will use a simple technique where every time the egress (in our case this will be the switch before the destination host) will detect a packet which suffers from congestion, then it will notify the ingress switch with a notification message and upon the receiving it will move the flow to another path which it will be selected randomly. *This scenario is not the best way to explore the available paths but it is well enough to show in emulated environment the collection of information as packets traverse the network while communicate between the switches and cause them react to network states.*

The design is split it two parts. Section 5.1 describes how the flows collide and how to read queue information using a simple topology, while in Section 5.1 we will provide the network the means to detect collisions and react in a more complex topology.

For both parts we used mininet to implement the topology,nload[39](a tool which monitors network traffic and bandwidth in real time using visualization for incoming and outgoing traffic) and Wireshark[40] in 1 Virtual Machine with characteristics as follows:

- Memory: 4096MB
- CPU: 3
- OS: Ubuntu 16.04 LTS

4.1 Queue Depth - Collision Detection

We implemented a network topology as shown in Fig. 13 with host 1 and 2 connected to hosts 3 and 4 via 3 switches connected in line with 10Mps bandwidth per link, in order to demonstrate how to read the queue information. To do so we will have to add the telemetry header, if the packet will not have it (at the ingress-first switch) and while updating the queue depth value, set it to the highest along the path.

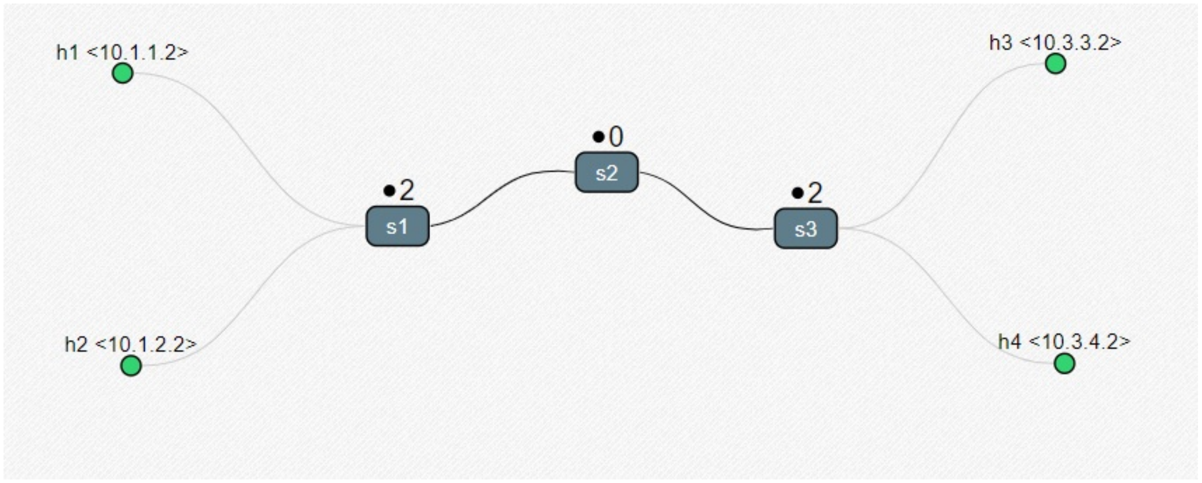


Figure 13: Linear Topology[3]

First we will create a header and name it telemetry in headers.p4 file as shown in line 2 in Listings. 1.The INT header has two fields (line 13 and line 14) the enq_qdepth and the nextHeaderType, both are 16 bits and are placed between the ethernet and ipv4 headers.

```
1  const bit<16> TYPE_IPV4 = 0x800;
2  const bit<16> TYPE_TELEMETRY = 0x7777;
3
4  typedef bit<9>  egressSpec_t;
5  typedef bit<48> macAddr_t;
6  typedef bit<32> ip4Addr_t;
7
8
9  header ethernet_t {...
10 }
11
```

```
header telemetry_t {
  bit<16> enq_qdepth;
  bit<16> nextHeaderType;
}

header ipv4_t {...
}

header tcp_t{...
}

struct metadata {...
}

struct headers {
    ethernet_t   ethernet;
    telemetry_t  telemetry;
    ipv4_t       ipv4;
    tcp_t        tcp;
}
```

Listings 1: headers.p4

Accordingly we will update the deparser as shown in Listings.2. We add the logic when extracting the ethernet packet to select on the value of the etherType (0x7777), if it is Type IPv4 or Telemetry, otherwise to accept (default). Also we extract the telemetry header and then we select on the value of the "nextHeaderType" that the parser is using to know which will be the next header (next header will be the IPv4, 0x800)[41].

```
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType){
            TYPE_IPV4: parse_ipv4;
            TYPE_TELEMETRY: parse_telemetry;
            default: accept;
        }
    }
    state parse_telemetry{
      packet.extract(hdr.telemetry);
      transition select(hdr.telemetry.nextHeaderType){
        TYPE_IPV4: parse_ipv4;
        default: accept;
      }
    }
```

Listings 2: parser.p4

The packets are enqueued to their output port by the traffic manager and this queueing information is available only at the egress pipeline in which we set logic as shown in Listing.3 to point the switches to add the telemetry header in all the TCP packets with destination port 7777.

```
    control MyEgress(inout headers hdr,
                     inout metadata meta,
                     inout standard_metadata_t standard_metadata) {
    apply {
      if (hdr.tcp.isValid() && hdr.tcp.dstPort == 7777){
        if (hdr.telemetry.isValid()){
          if (hdr.telemetry.enq_qdepth < (bit<16>)standard_metadata.
   enq_qdepth){
            hdr.telemetry.enq_qdepth = (bit<16>)standard_metadata.
   enq_qdepth;
          }
        }else{
          hdr.telemetry.setValid();
          hdr.telemetry.enq_qdepth = (bit<16>)standard_metadata.enq_qdepth
   ;
          hdr.ethernet.etherType = TYPE_TELEMETRY;
          hdr.telemetry.nextHeaderType = TYPE_IPV4;
        }
      }
    }
}
```

Listings 3: int1.p4

4.2 Collision Detection and Load Balancing

4.2.1 Adding the Telemetry Header

In order to observe how the flows collide in a network with multiple alternative paths, we created a topology as shown in Fig.14 containing 6 switches and 8 hosts with 10 Mps bandwidth given to all links.

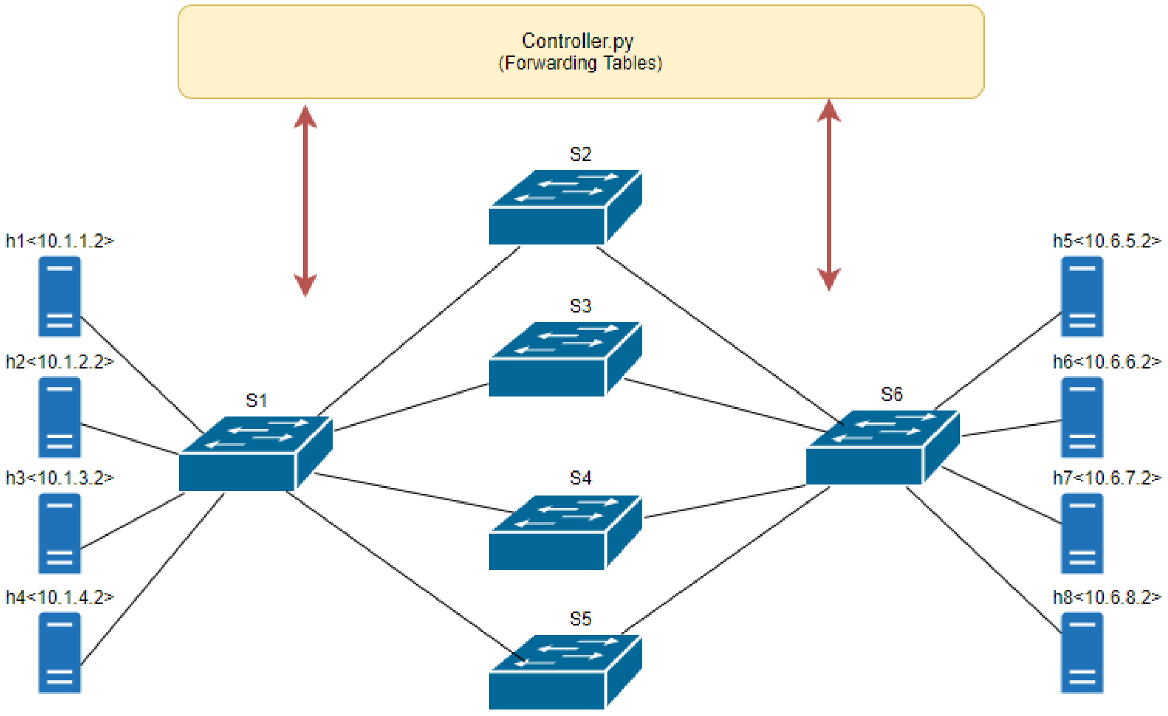


Figure 14: Topology

In section 5.1 we set the telemetry header when the packet entered the network addressed to a predefined destination port (port:7777) and it kept the telemetry header until the final destination. In this section, the switches will make use of the telemetry header from the esoteric network to detect the congestion and move the flows, load balance the network and then before the packets will be forwarded to the hosts we will have to remove the telemetry header (INT Sink). We need to specify which type of networking device (host or switch) is connected and to which switch port and for that we will use a table, the control plane and the topology. So we define a match action table in the ingress pipeline that will identify the output port for each packet. As we see in Listing.4 the table match exact to the egress_spec[5] and an action is being called to set the type of networking device (for example 1 for hosts and/or 2 for switches) the packet will be forwarded at.

```
    table egress_type {
        key = {
            standard_metadata.egress_spec: exact;
        }

        actions = {
            set_egress_type;
            NoAction;
        }
        size=64;
        default_action = NoAction;
    }

```

Listings 4: int.p4 Ingress Pipeline

After applying the table in the ingress control section we modify the egress code adding the logic: if the packet has no telemetry header and the next hop is a switch then we add the telemetry header, nextHeaderType, set the depth field and the Ethernet type to 0x7777. Then we check if the telemetry header is valid and the next hop is a host then we will have to remove the telemetry header and set the etherType to IPV4. If the next hop is a switch and the telemetry header is valid then we will update the enq_qdepth field in the packet to a bigger value. So when the TCP packets are entering to the network, the programmed switches will add the telemetry header and extract the telemetry headers before arriving to the host. In Table.2 we can see the switch port mapping from which we selected the Switch 6 and ports 5 and 4 which are connected

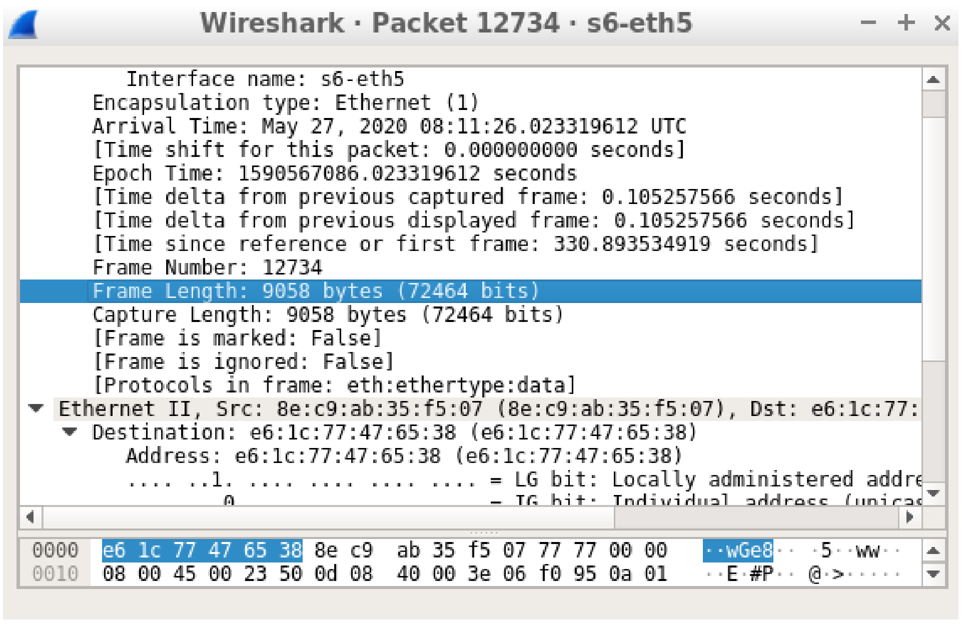
[5]The egress_spec is target specific. The behavioral (BMv2) target will specify how to write to this field to accomplish various tasks like forwarding to a specified port

s1:	1:h1 2:h2 3:h3 4:h4 5:s2 6:s3 7:s4 8:s5
s2:	1:s1 2:s6
s3:	1:s1 2:s6
s4:	1:s1 2:s6
s5:	1:s1 2:s6
s6:	1:h5 2:h6 3:h7 4:h8 5:s2 6:s3 7:s4 8:s5

Table 2: Switch Port Mapping

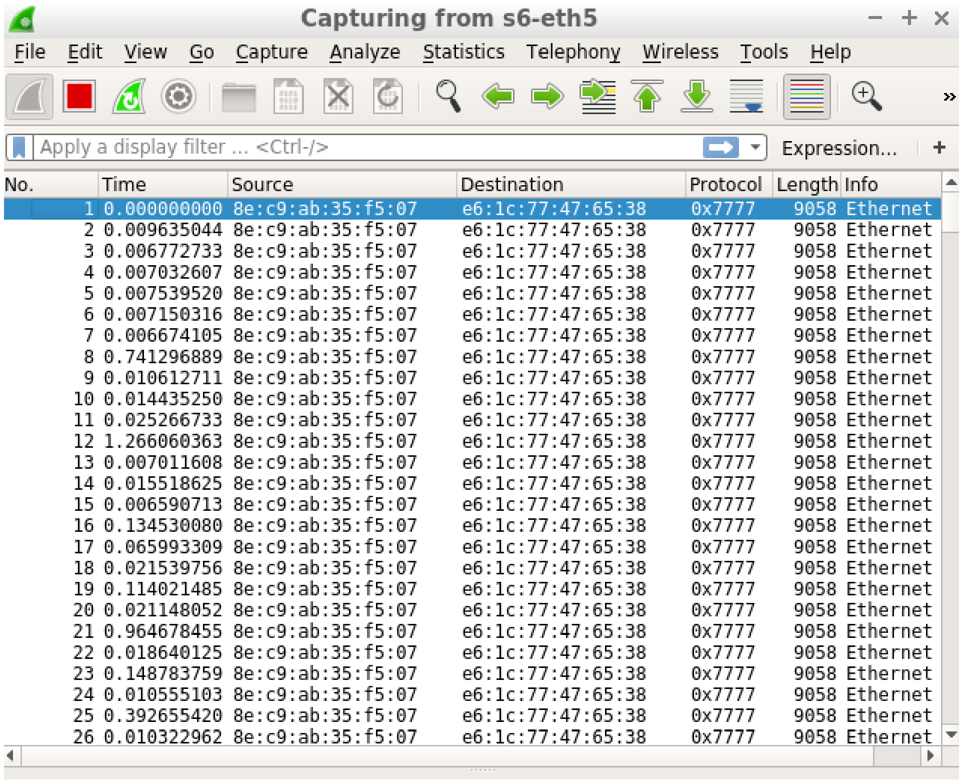
to the Switch 2 and host 8 respectively. So we have one internal network connection and one connection to the final host.

While sending TCP traffic on the network, we captured the packets with Wireshark to prove that the internal monitoring link ethernet header is *0x7777*. These extra four bytes (9854+(4)) we see in Fig.15 and Fig.16 are the actual telemetry bytes. Then, before the packet reach the destination host, switch 6 will extract the telemetry header as shown in Fig.17 and Fig.fig:w4 and deliver the TCP packet (9854 bytes).



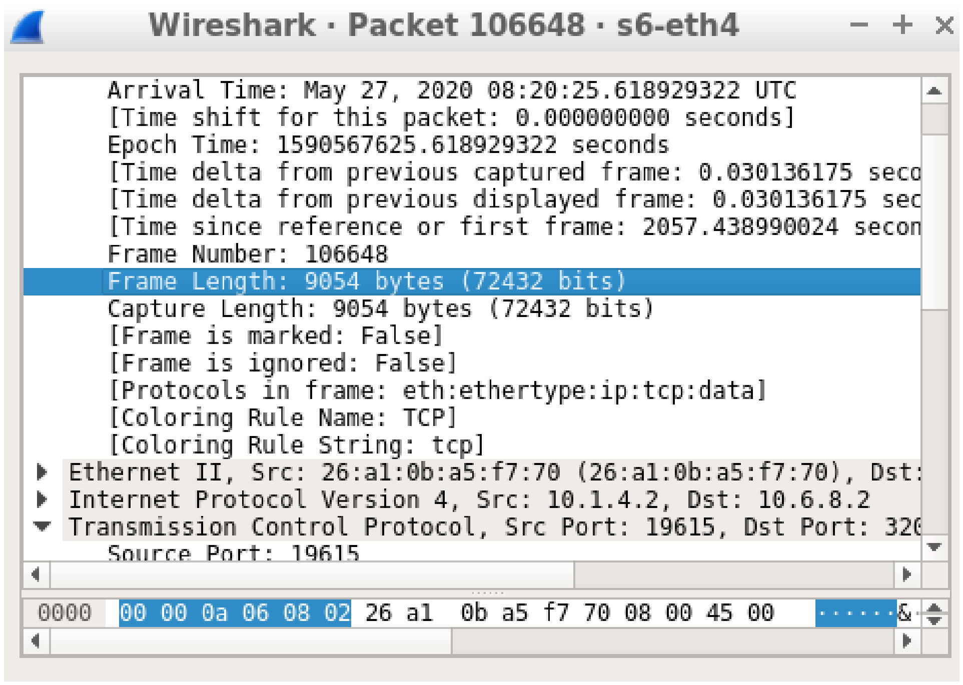
```
    Interface name: s6-eth5
  Encapsulation type: Ethernet (1)
  Arrival Time: May 27, 2020 08:11:26.023319612 UTC
  [Time shift for this packet: 0.000000000 seconds]
  Epoch Time: 1590567086.023319612 seconds
  [Time delta from previous captured frame: 0.105257566 seconds]
  [Time delta from previous displayed frame: 0.105257566 seconds]
  [Time since reference or first frame: 330.893534919 seconds]
  Frame Number: 12734
  Frame Length: 9058 bytes (72464 bits)
  Capture Length: 9058 bytes (72464 bits)
  [Frame is marked: False]
  [Frame is ignored: False]
  [Protocols in frame: eth:ethertype:data]
▼ Ethernet II, Src: 8e:c9:ab:35:f5:07 (8e:c9:ab:35:f5:07), Dst: e6:1c:77:
  ▼ Destination: e6:1c:77:47:65:38 (e6:1c:77:47:65:38)
      Address: e6:1c:77:47:65:38 (e6:1c:77:47:65:38)
      .... ..1. .... .... .... .... = LG bit: Locally administered addre

0000  e6 1c 77 47 65 38 8e c9  ab 35 f5 07 77 77 00 00   ··wGe8·· ·5··ww··
0010  08 00 45 00 23 50 0d 08  40 00 3e 06 f0 95 0a 01   ··E·#P·· @·>·····
```

Figure 15: Packet from Switch 6/Port 5


Capturing from s6-eth5
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

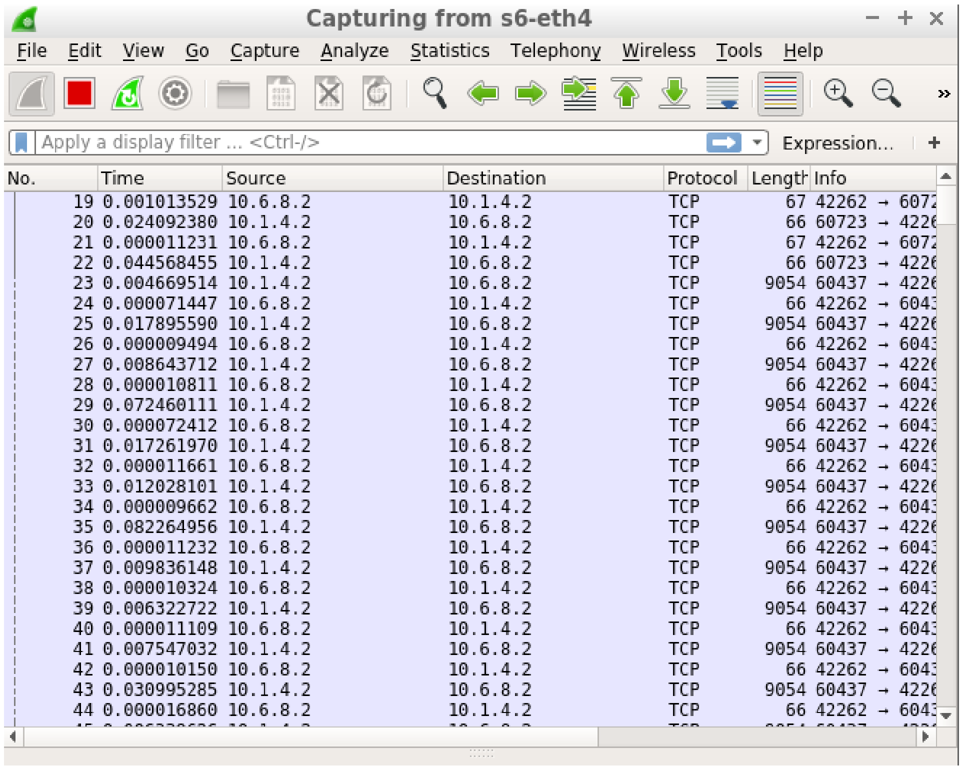
```
No.  Time        Source             Destination        Protocol Length Info
   1 0.000000000 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
   2 0.009635044 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
   3 0.006772733 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
   4 0.007032607 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
   5 0.007539520 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
   6 0.007150316 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
   7 0.006674105 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
   8 0.741296889 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
   9 0.010612711 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  10 0.014435250 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  11 0.025266733 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  12 1.266060363 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  13 0.007011608 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  14 0.015518625 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  15 0.006590713 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  16 0.134530080 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  17 0.065993309 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  18 0.021539756 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  19 0.114021485 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  20 0.021148052 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  21 0.964678455 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  22 0.018640125 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  23 0.148783759 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  24 0.010555103 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  25 0.392655420 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
  26 0.010322962 8e:c9:ab:35:f5:07  e6:1c:77:47:65:38  0x7777   9058 Ethernet
```

Figure 16: Packet-Analysis from Switch 6/Port 5



```
    Arrival Time: May 27, 2020 08:20:25.618929322 UTC
    [Time shift for this packet: 0.000000000 seconds]
    Epoch Time: 1590567625.618929322 seconds
    [Time delta from previous captured frame: 0.030136175 seco
    [Time delta from previous displayed frame: 0.030136175 sec
    [Time since reference or first frame: 2057.438990024 secon
    Frame Number: 106648
    Frame Length: 9054 bytes (72432 bits)
    Capture Length: 9054 bytes (72432 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ethertype:ip:tcp:data]
    [Coloring Rule Name: TCP]
    [Coloring Rule String: tcp]
▶ Ethernet II, Src: 26:a1:0b:a5:f7:70 (26:a1:0b:a5:f7:70), Dst:
▶ Internet Protocol Version 4, Src: 10.1.4.2, Dst: 10.6.8.2
▼ Transmission Control Protocol, Src Port: 19615, Dst Port: 32(
    Source Port: 19615

0000  00 00 0a 06 08 02 26 a1  0b a5 f7 70 08 00 45 00   ······&·
```

Figure 17: Packet from Switch 6/Port 5


Capturing from s6-eth4
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Expression...

```
No.   Time        Source      Destination  Protocol Length Info
   19 0.001013529 10.6.8.2    10.1.4.2     TCP        67 42262 → 6072
   20 0.024092380 10.1.4.2    10.6.8.2     TCP        66 60723 → 4226
   21 0.000011231 10.6.8.2    10.1.4.2     TCP        67 42262 → 6072
   22 0.044568455 10.1.4.2    10.6.8.2     TCP        66 60723 → 4226
   23 0.004669514 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   24 0.000071447 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   25 0.017895590 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   26 0.000009494 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   27 0.008643712 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   28 0.000010811 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   29 0.072460111 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   30 0.000072412 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   31 0.017261970 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   32 0.000011661 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   33 0.012028101 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   34 0.000009662 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   35 0.082264956 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   36 0.000011232 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   37 0.009836148 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   38 0.000010324 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   39 0.006322722 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   40 0.000011109 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   41 0.007547032 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   42 0.000010150 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
   43 0.030995285 10.1.4.2    10.6.8.2     TCP      9054 60437 → 4226
   44 0.000016860 10.6.8.2    10.1.4.2     TCP        66 42262 → 6043
```

Figure 18: Packet-Analysis from Switch 6/Port 5

4.2.2 Congestion Detector

Inside the egress block of the switches, after extracting the telemetry header now we need to apply the logic to detect congestions on the flows and when a congestion will be detected to send a notification packet to the ingress switch which is the first switch that adds the telemetry header on the packets. To create such packet notification and send it to the ingress switch we need to clone the packets that are triggering the congestion, modify and send them back to the switch. When the packet will be received, the switch will check the queue depth field if it is above the specified threshold[6], which should be between 20 to 60 packets in order to trigger the feedback message.

On the occasion of a congestion, the egress switch will get a storm of packets which will trigger the feedback notification mechanism. To avoid such burst of packets we

[6]By default the queues in networks are 64 packets long

will add a timeout per packet flow. So when the switch will send a feedback notification message for a flow, it will save the time stamp in a register, so the next time it will send again a feedback notification message, it will check the register for the time difference, for example, if the difference will be bigger than the specified (i.e.less than a second).

Congestions are usually created when multiple flows collide and all will trigger feedback notification messages so adding all the flows to another path is not optimal and may cause congestion to the new paths and leave the current without traffic, so to solve this issue we added a probability.

Now we need to send the notification packet to the ingress switch and notify that the specified flow is undergo a congestion. To create the new packet beside the one that the switch is forwarding, we need to clone the packet from egress to egress by adding an identifier that will inform the switch on which port to clone the packet. After cloning the packet, we recirculate it. That allows us to send the packet back to the ingress pipeline and be able to use the forwarding tables again to forward the packets. Then we modified the ethernet type in the parse so the switches to be able to identify the feedback notification messages (0x7778).

4.2.3 "Load Balancing"

The ingress switches will need to move the congested flows to new paths so every feedback notification message which should be dropped-when the switch is sending the packets to the host- we save in a register identifier a value for each flow. So when the switch will receive a congestion feedback notification message for a flow it will update the register value with a new identifier-for this case we used a random number. To get the register index we used a hashed function (5-tuple hash[42], source/destination IP, source/destination port, protocol) as a randomizer. In the end we drop the congestion feedback notification message.

5 Evaluation

Starting the topology in Mininet from Section 5.1 as shown in Fig.11, we used a python script and the tool Scapy[43] to create and send probe packets from host 1 to host 3. In the same time on host 3 we run a python script using Scapy to sniff the incoming packets at the ethernet port 0. Then we used the tool Iperf[44] to create TCP traffic with MTU size 9000 at random ports in range 1024 to 65000. In the beginning, the queue depth observed was 0 because we generated only ICMP (mouse) traffic, but when we run the python script and generated traffic flow with Iperf3, from h1 and h2 to h3 and h4 respectively, then the two flows started to collide because we have only one single path in the selected topology. The queue was filling up to 63 (the default queue size in BMv2 is 64 packets) and the latency was increasing dramatically. We can observe in Fig.19 the correlation between the latency and the queue depth.

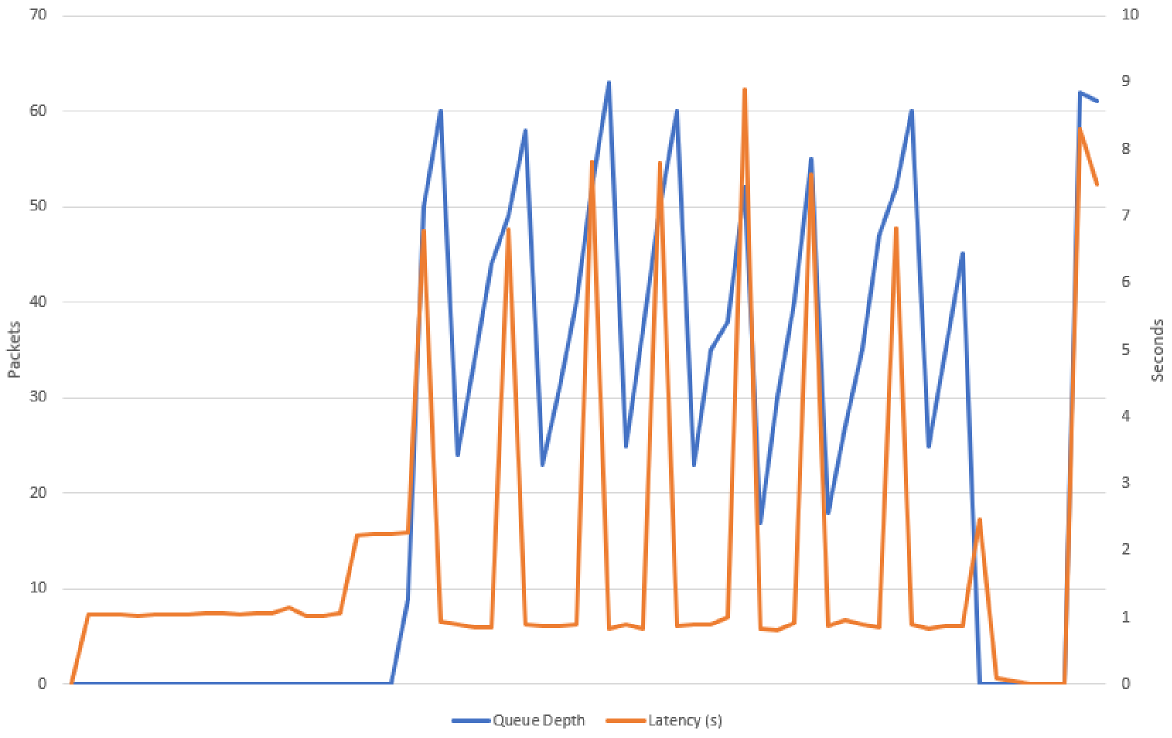


Figure 19: Queue depth and Latency

Following the example from Section 5.1 but with a more complex topology as shown in Fig.14, we used Iperf3 in order to generate TCP traffic with duration of 100 seconds, MTU size 9000, at random ports in range from 1024 to 65000 and from hosts 1-4 to hosts 5-8 respectively. First we sent traffic without applying Load Balancing in order to get throughput and latency measurements and then after applying Load Balancing, as described in Section 4, we generated and sent the same traffic to the network in order to compare between the two cases. Measuring the traffic in Switch 1 (s1:eth1, s1:eth2, s1:eth3 and s1:eth4 ports) and as you can see in Fig.20, Fig.21, Fig.22 and Fig23, we have a significant drop in throughput and instability in all of the four ports with an overall Average rate of 3.7 Mbits/sec and 93.18 ms latency [*in Table.3 we can see the Average results per host*] - to be reminded, all links are configured at 10 Mbits/sec.

h1:eth0 - s1:eth1	4.45 Mbits/sec
h2:eth0 - s1:eth2	3.22 Mbits/sec
h3:eth0 - s1:eth3	2.10 Mbits/sec
h4:eth0 - s1:eth4	5.00 Mbits/sec

Table 3: Switch1 eth1-eth4 ports Throughput

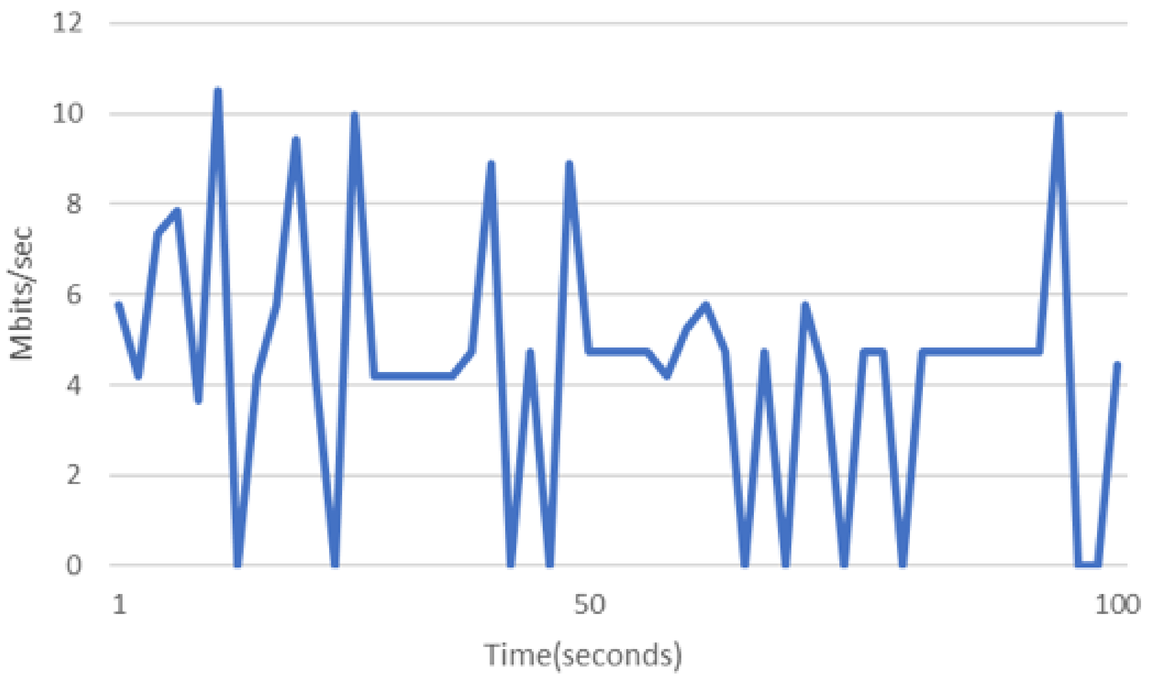


Figure 20: Switch:1 - Port:eth1

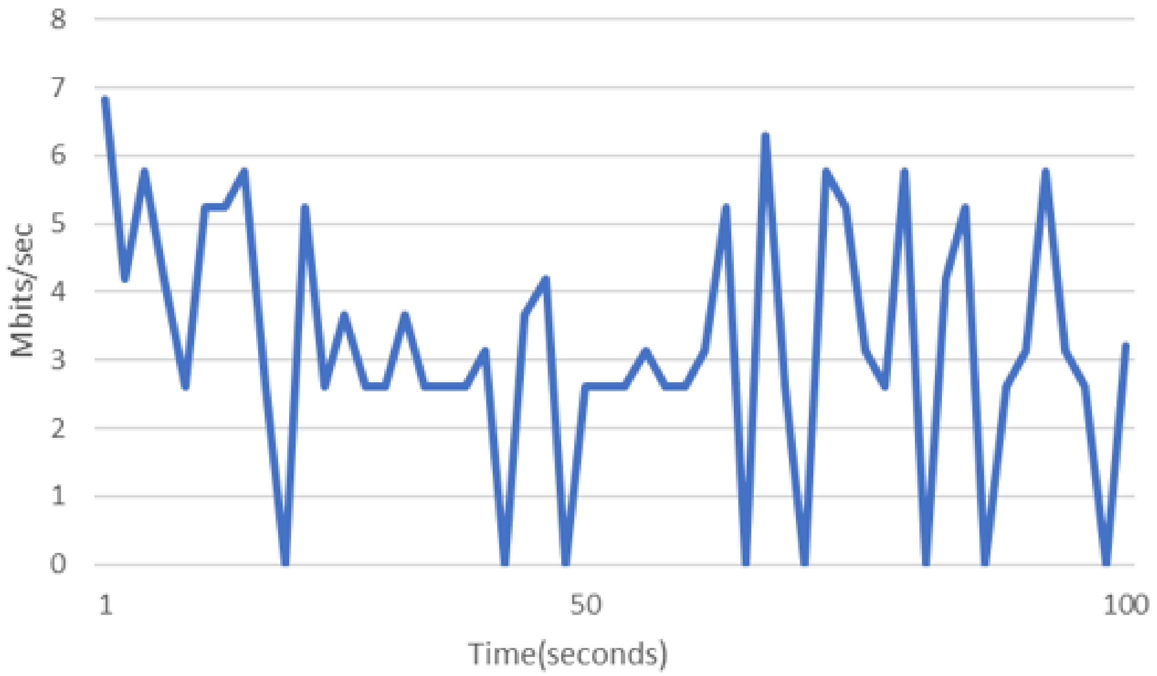


Figure 21: Switch:1 - Port:eth2

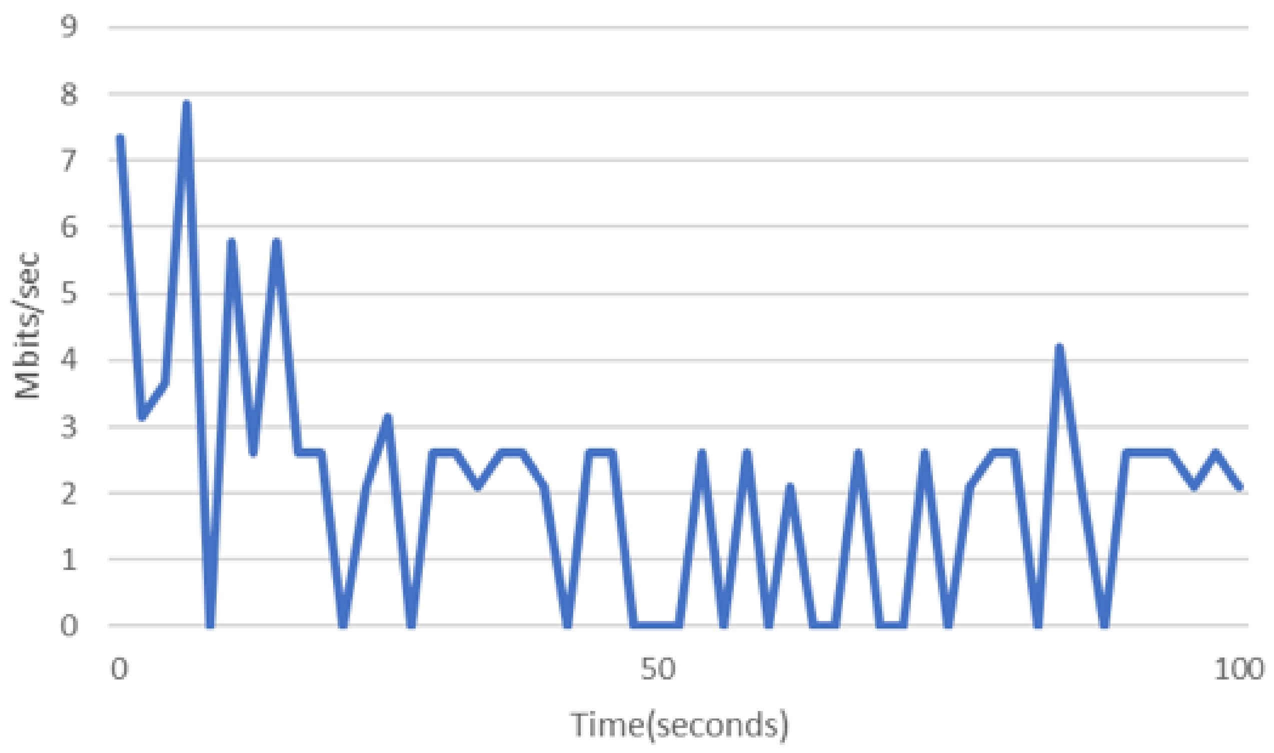


Figure 22: Switch:1 - Port:eth3

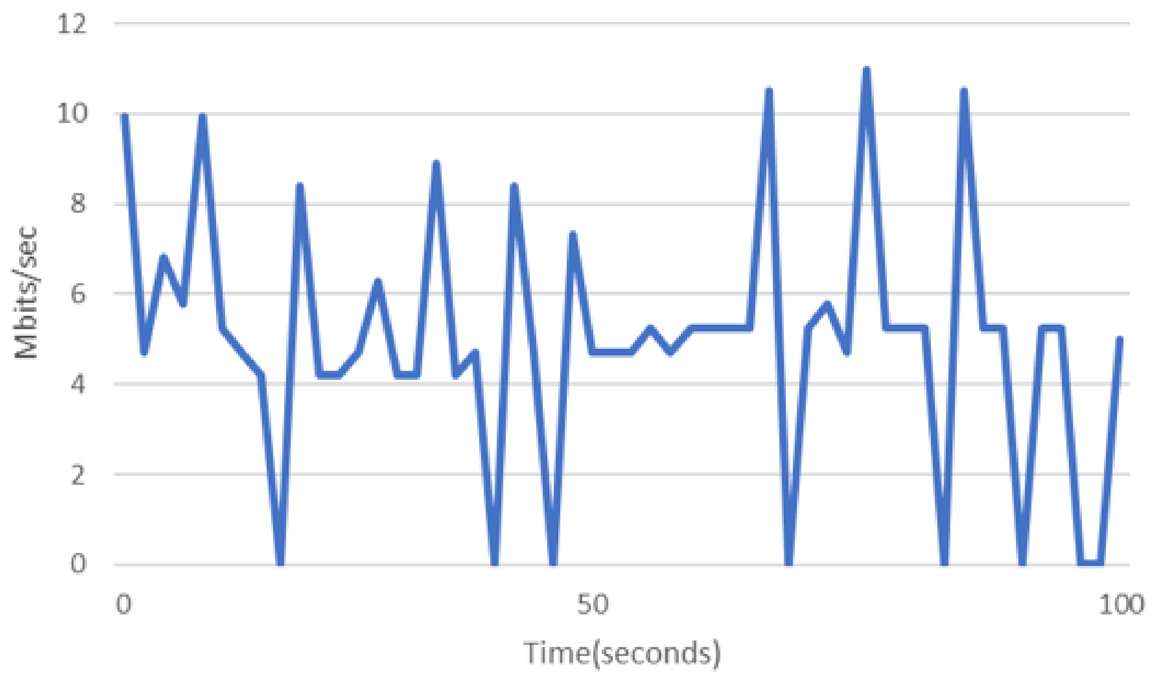


Figure 23: Switch:1 - Port:eth4

Taking advantage of the P4 Inband Network Telemetry, we were able to read the queue depth information as shown in Fig.24 and observed that the queue was constantly full, explaining the low throughput and the big latency. To solve this problem we applied Load Balancing with a threshold of 20 packets queue depth (as described in Section 4) and we managed to decrease the queue size significantly. The latency as shown in Fig.25 decreased from 93.18ms to 54.35ms in Average for all of the four ports, a reduction of 71.3%.

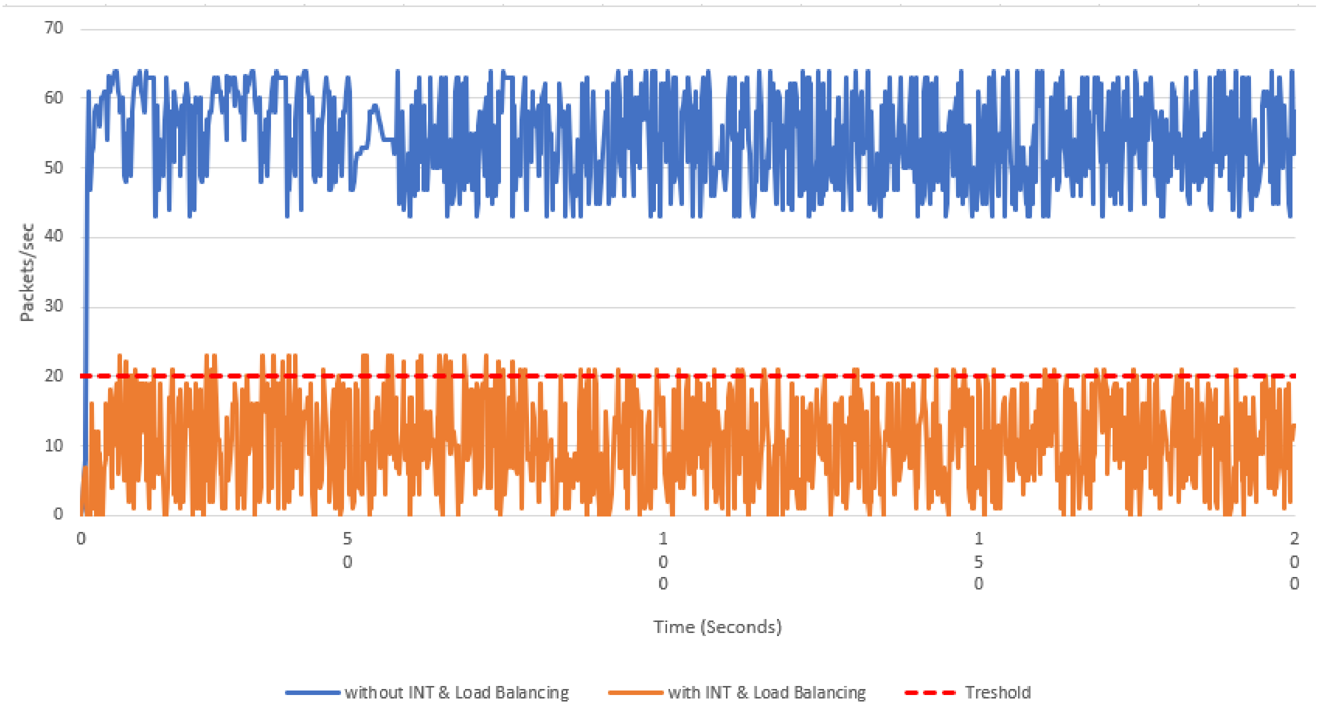


Figure 24: Queue depth and Latency - Comparison

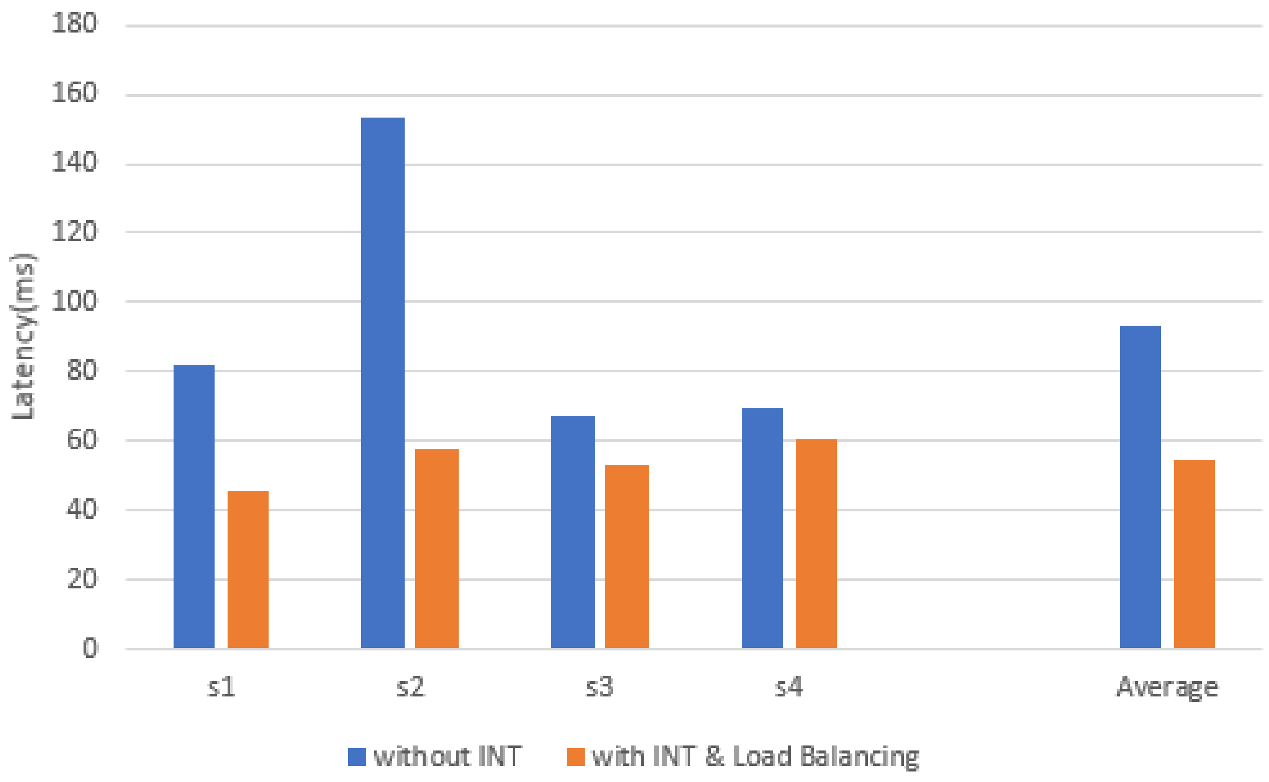


Figure 25: Latency(ms) with and without INT-LoadBalancing

By applying Load Balancing we observed also that the flow on each of the measured ports increased and had better stability as you can see in Fig.26, Fig.27, Fig.28 and Fig.29 with a throughput increase from 3.7 Mbits/sec to 9.19 Mbits/sec in Average for all of the four ports (*in Table.4 we can see the Average results per host*), an increase of 148%, as shown in Fig.30.

h1:eth1 - s1:eth1	9.12 Mbits/sec
h2:eth2 - s1:eth2	9.27 Mbits/sec
h3:eth3 - s1:eth3	9.11 Mbits/sec
h4:eth4 - s1:eth4	9.25 Mbits/sec

Table 4: Switch1 eth1-eth4 ports Throughput with Load Balancing

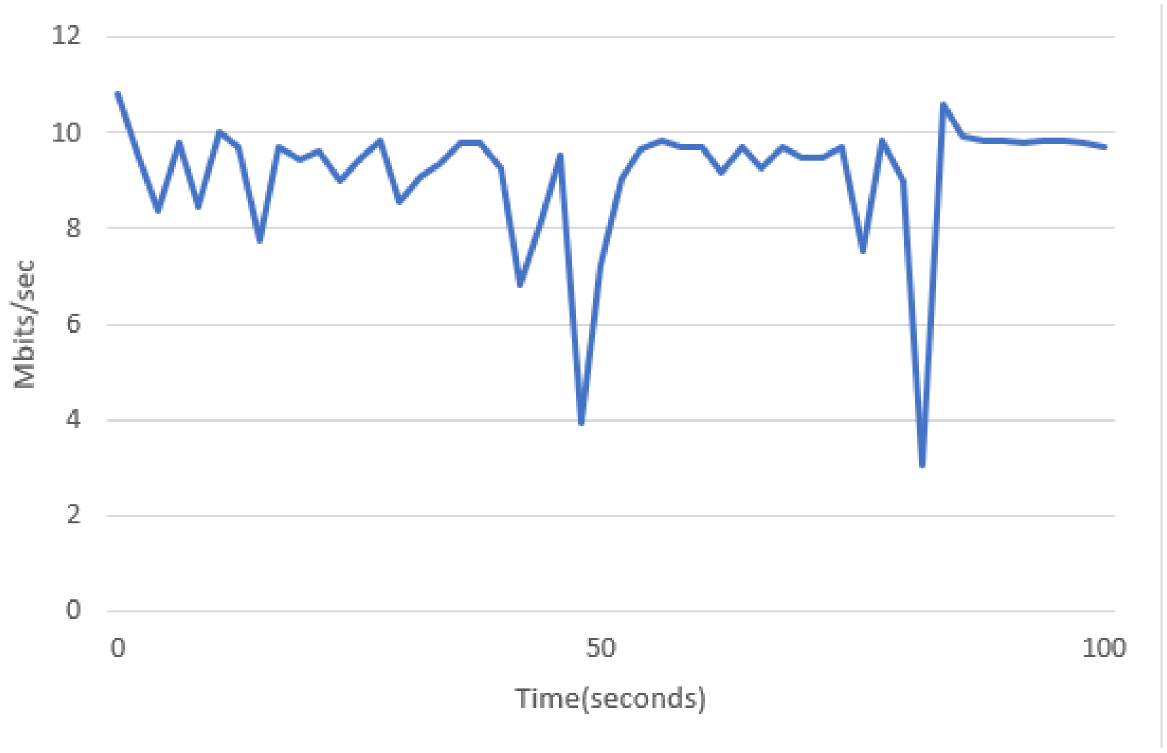


Figure 26: Switch:1 - Port: eth1

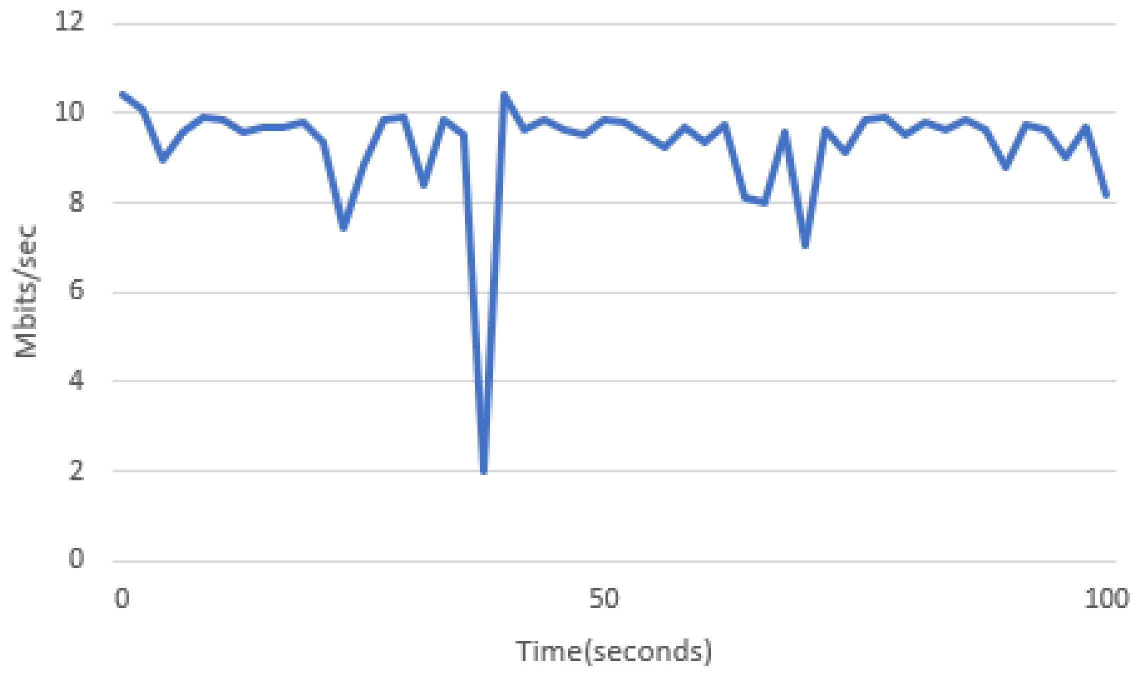


Figure 27: Switch:1 - Port: eth2

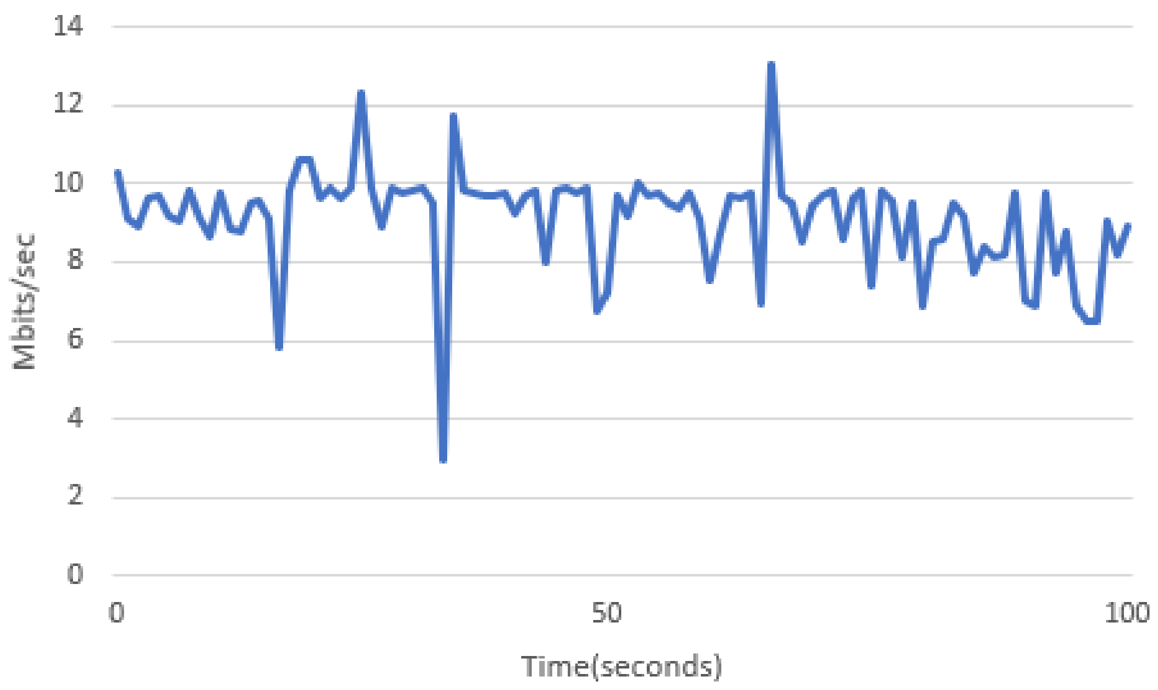


Figure 28: Switch:1 - Port: eth3

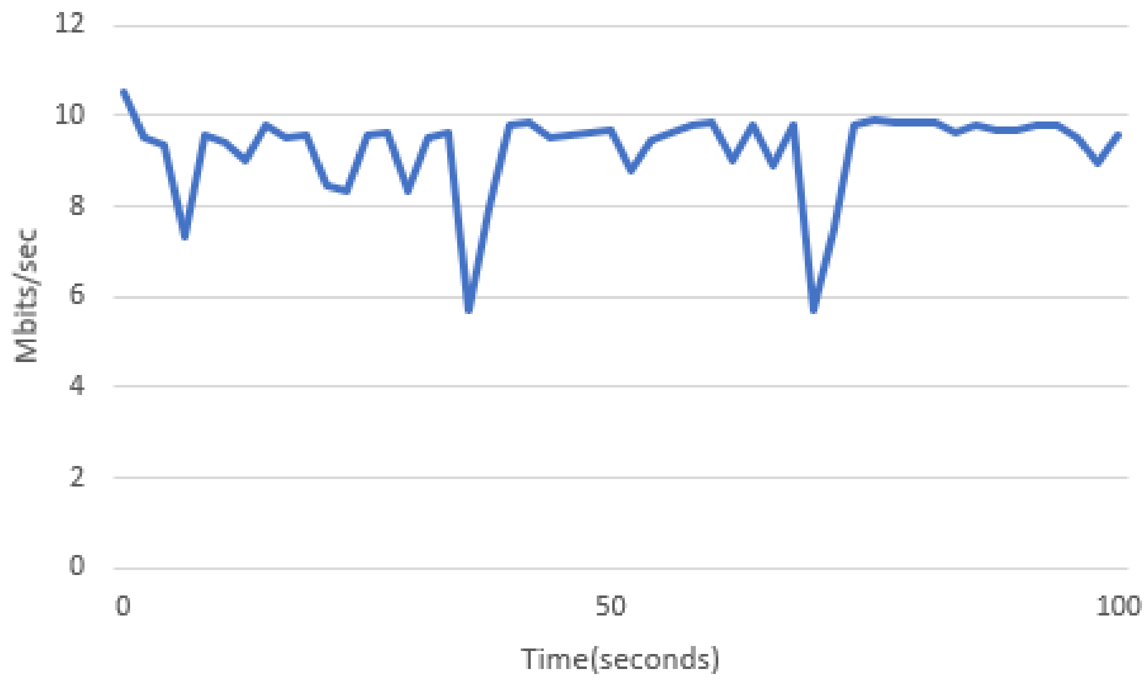


Figure 29: Switch:1 - Port: eth4

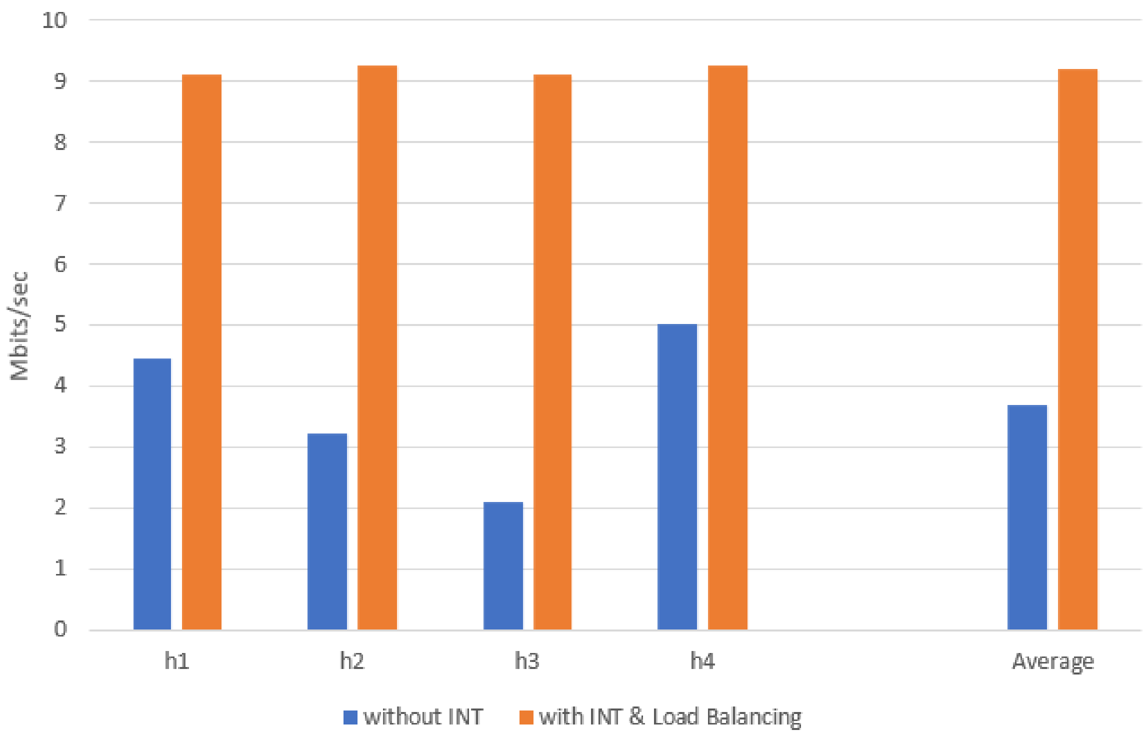


Figure 30: Switch1: Throughput Comparison

6 Conclusion and Future Work

Current monitoring methods and mechanisms are inadequate. There are not fast enough because they involve CPU and Control Planes in an environment where the network state changes rapidly. Also these monitoring methods do not provide end-to-end state making difficult to correlate per element state with the actual path of a flow[41].

This thesis presented P4, a programming protocol-independent packet processors language and proposed an implementation of In-band Network Telemetry in P4 which identifies congestion in the network by monitoring the queue occupancy in the data plane and in real time (as possible) according to the network state.

Emulating in Mininet two different topologies, a simple with 3 switches in line with 4 hosts and a more complex with 6 switches and 8 hosts (both networks with 10 Mbits/sec bandwidth), we implemented In-band Network Telemetry using $P4_{16}$ in order to collect real-time network information from the packets and measure the queue occupancy, network throughput and latency.

In the first case, when we generated traffic we observed that the queue occupancy is correlated with the latency. Every time the queue was reaching the limit (63 packets queue depth is the default for BMv2) the latency was at the highest level, reaching up to 8 seconds.

Taking one step further and using P4 and INT information in the second case after generating traffic in the network, we were able to monitor and observe not only that the latency was very high but also that the throughput of the hosts was very low and by using the telemetry header from the esoteric network we were able to detect the congestion.

To solve the observed issue we used Load Balancing, so when a congestion will be identified using a threshold triggering mechanism, a clone packet will be created and sent to the ingress switch and the notified switch will change the flow to a new path using a 5-tuple simple algorithm.

The results as shown in Section 5, even with this simple algorithm, showed that the flows were split in all paths increasing the throughput significantly by avoiding the congestion, from 3.7 Mbps/sec to $\sim$ 10 Mbps/sec, while in the same time the latency reduced from 94.2 ms to 54.2 ms.

In-Band Network Telemetry using P4 can provide real-time network state directly in the data plane opening new possibilities of enhanced monitoring and troubleshooting, able to adapt to any encapsulation format, any state that is required to be collected and any feature, protocol (current or future).

Further work is still required with the evaluation of offloading and Load Balancing in order to fully understand how different factors may affect the performance of the network. In the future we will conduct experiments using more complex algorithms for realistic applications using hardware P4 programmable networking devices and In-Network Telemetry.

References

[1] "P4lang/p4runtime," p4language. [Online]. Available: https://github.com/p4lang/p4runtime

[2] "P4lang/p4-applications," p4language. [Online]. Available: https://github.com/p4lang/p4-applications

[3] SDN Narmox Spear. [Online]. Available: http://demo.spear.narmox.com/app/?apiurl=demo#!/mininet

[4] W. L. da Costa Cordeiro, J. A. Marques, and L. P. Gaspary, "Data Plane Programmability Beyond OpenFlow: Opportunities and Challenges for Network and Service Operations and Management," vol. 25, no. 4, pp. 784–818. [Online]. Available: http://link.springer.com/10.1007/s10922-017-9423-2

[5] Q. He, X. Wang, and M. Huang, "OpenFlow-based low-overhead and high-accuracy SDN measurement framework," vol. 29, no. 2, p. e3263, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ett.3263. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3263

[6] B. Li, J. Springer, G. Bebis, and M. Hadi Gunes, "A survey of network flow applications," vol. 36, no. 2, pp. 567–581. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1084804512002676

[7] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic Matrix Estimator for OpenFlow Networks," in *Passive and Active Measurement*, ser. Lecture Notes in Computer Science, A. Krishnamurthy and B. Plattner, Eds. Springer Berlin Heidelberg, vol. 6032, pp. 201–210. [Online]. Available: http://link.springer.com/10.1007/978-3-642-12334-4_21

[8] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker, "Software-defined internet architecture: Decoupling architecture from infrastructure," in *Proceedings of the 11th ACM Workshop on Hot*

Topics in Networks - HotNets-XI. ACM Press, pp. 43–48. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2390231.2390239

[9] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey." [Online]. Available: http://arxiv.org/abs/1406.0440

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," vol. 38, no. 2, pp. 69–74. [Online]. Available: https://doi.org/10.1145/1355734.1355746

[11] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," vol. 44, no. 2, pp. 87–98. [Online]. Available: https://dl.acm.org/doi/10.1145/2602204.2602219

[12] Open Networking Foundation is an operator led consortium leveraging SDN, NFV and Cloud technologies to transform operator networks and business models. [Online]. Available: https://www.opennetworking.org/

[13] H. Soni, "Towards network softwarization: A modular approach for network control delegation," p. 135.

[14] Clarifying the differences between P4 and OpenFlow. [Online]. Available: https://p4.org/p4/clarifying-the-differences-between-p4-and-openflow.html

[15] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "Programming Protocol-Independent Packet Processors." [Online]. Available: http://arxiv.org/abs/1312.1719

[16] P4lang/p4-spec. [Online]. Available: https://github.com/p4lang/p4-spec

[17] P4˜16˜ Portable Switch Architecture (PSA). [Online]. Available: https://p4.org/p4-spec/docs/PSA-v1.1.0.html

[18] "P4lang/tutorials," p4language. [Online]. Available: https://github.com/p4lang/tutorials

[19] "P4lang/p4c," p4language. [Online]. Available: https://github.com/p4lang/p4c

[20] "P4lang/behavioral-model," p4language. [Online]. Available: https://github.com/p4lang/behavioral-model

[21] P4lang/behavioral-model. [Online]. Available: https://github.com/p4lang/behavioral-model

[22] P4lang/p4-applications. [Online]. Available: https://github.com/p4lang/p4-applications

[23] S.-Y. Wang, Y.-R. Chen, J.-Y. Li, H.-W. Hu, J.-A. Tsai, and Y.-B. Lin, "A Bandwidth-Efficient INT System for Tracking the Rules Matched by the Packets of a Flow," in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9013581/

[24] T. Mori, R. Kawahara, S. Naito, and S. Goto, "On the characteristics of Internet traffic variability: Spikes and elephants," in *2004 International Symposium on Applications and the Internet. Proceedings.*, pp. 99–106.

[25] P4 Language and Related Specifications. [Online]. Available: https://p4.org/specs/

[26] H. Gredler, S. Bhandari, daniel. bernierbell.ca, S. Youell, F. Brockners, T. Mizrahi, J. Lemon, remybarefootnetworks. com, D. Mozes, C. Pignataro, P. Lapukhov, and J. Leddy. Data Fields for In-situ OAM. [Online]. Available: https://tools.ietf.org/html/draft-ietf-ippm-ioam-data-09

[27] J. Kumar, A. Ghanwani, D. Cai, J. Lemon, H. OU, R. Manur, H. Holbrook, Y. Li, and S. Anubolu. Inband Flow Analyzer. [Online]. Available: https://tools.ietf.org/html/draft-kumar-ippm-ifa-01

[28] M. Bjorklund <mbj@tail f.com>. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). [Online]. Available: https://tools.ietf.org/html/rfc6020

[29] P4lang/papers. [Online]. Available: https://github.com/p4lang/papers

[30] J. Geng, J. Yan, Y. Ren, and Y. Zhang, "Design and Implementation of Network Monitoring and Scheduling Architecture Based on P4," in *Proceedings of the 2nd International Conference on Computer Science and Application Engineering - CSAE '18*. ACM Press, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3207677.3278059

[31] B. Turkovic, F. Kuipers, N. van Adrichem, and K. Langendoen, "Fast network congestion detection and avoidance using P4," in *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies - NEAT '18*. ACM Press, pp. 45–51. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3229574.3229581

[32] J. Hyun, N. Van Tu, and J. W.-K. Hong, "Towards knowledge-defined networking using in-band network telemetry," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/8406169/

[33] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-Directed Hardware Design for Network Performance Monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, pp. 85–98. [Online]. Available: https://dl.acm.org/doi/10.1145/3098822.3098829

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

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

[36] M. Dimolianis, A. Pavlidis, and V. Maglaris, "A Multi-Feature DDoS Detection Schema on P4 Network Hardware," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pp. 1–6.

[37] I. Pelle, F. Paolucci, B. Sonkoly, and F. Cugini, "Telemetry-Driven Optical 5G Serverless Architecture for Latency-Sensitive Edge Computing," in *2020 Optical Fiber Communications Conference and Exhibition (OFC)*, pp. 1–3.

[38] Z. Zhou, E. O. Zaballa, M. S. Berger, and Y. Yan, "Detection of Fog Network Data Telemetry Using Data Plane Programming," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*, ser. OpenAccess Series in Informatics (OASIcs), A. Cervin and Y. Yang, Eds., vol. 80. Schloss Dagstuhl{Leibniz-Zentrum fuer Informatik, pp. 12:1–12:11. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12006

[39] Nload(1): Displays current network usage - Linux man page. [Online]. Available: https://linux.die.net/man/1/nload

[40] Wireshark · Go Deep. [Online]. Available: https://www.wireshark.org/

[41] Nsg-ethz/p4-learning. [Online]. Available: https://github.com/nsg-ethz/p4-learning

[42] A. U. Khan, D. M. Chawhan, and D. Y. Suryawanshi, "Design of High Performance Packet Classification Architecture for Communication Networks," vol. 9, no. 4, p. 8.

[43] P. B. a. t. S. community. Scapy. [Online]. Available: https://secdev.github.io/

[44] iPerf - The TCP, UDP and SCTP network bandwidth measurement tool. [Online]. Available: https://iperf.fr/