Πρόγραμμα Μεταπτυχιακών Σπουδών
Τμήμα Εφαρμοσμένης Πληροφορικής
ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ

Implementation of Network Telemetry System With P4 Programming Language

CHRISTOS DEMERTZIS

Supervising Professor
Dr. Papadimitriou Panagiotis

Agenda

- Objectives
- Programmable Networks
  - Software Defined Networking (SDN)
  - OpenFlow (OF)
- P4
  - P4$_{16}$ Architecture
  - P4$_{16}$ Language
- In-band Network Telemetry
  - Modes
  - INT Headers
- Experiments Overview
- Experiment 1 – Queue Depth
- Experiment 2 – Congestion discovery & Load Balancing
- Results
- Conclusion

Objectives

- Implementation of a network telemetry system using P4 programming language
- Present the new capabilities of emerging programmable networking devices to encapsulate processing metadata information.

- What is P4?
- What is In-band Network Telemetry ?
- What benefits we will get from INT or simple why to use it?

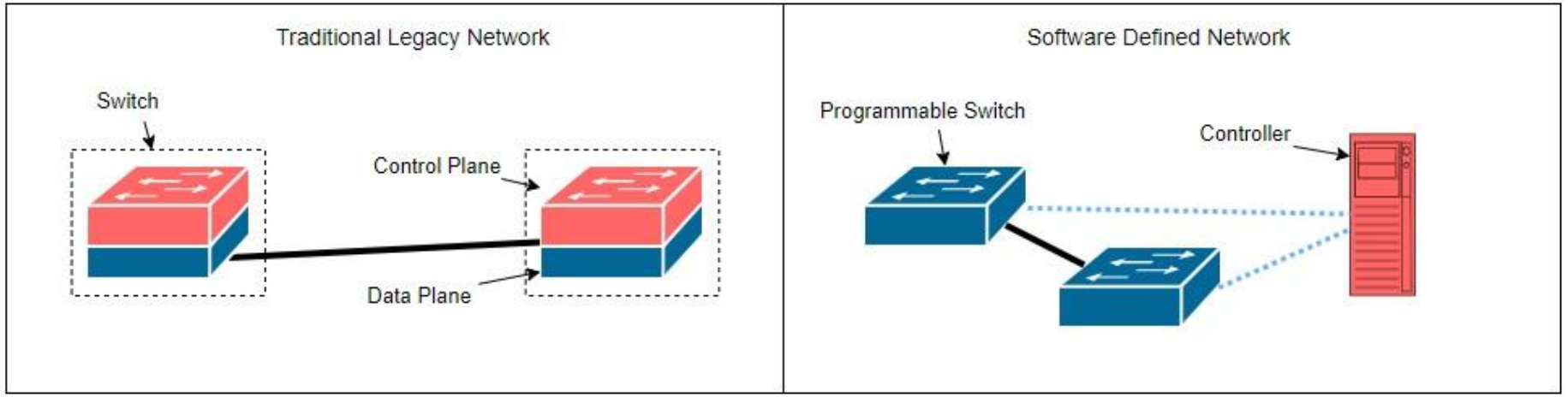
Programmable Networks

- Software Defined Networking (SDN)
- OpenFlow (OF)

Software Defined Networking (SDN)

- SDN is a new approach to networking
- Virtualize the network
- 2 Simple concepts
  - Decoupling the forwarding hardware from the control decisions (the control plane from the data plane)
  - Provides an open API for direct access to the data plane.

- Advantages
  - Networks can spun up and down dynamically
  - Networks can be fine tuned for specific application use cases
  - Security policies can be installed on each individual network.
  - Simpler Management
  - Less dependence on vendors and standards
  - Cheaper equipment

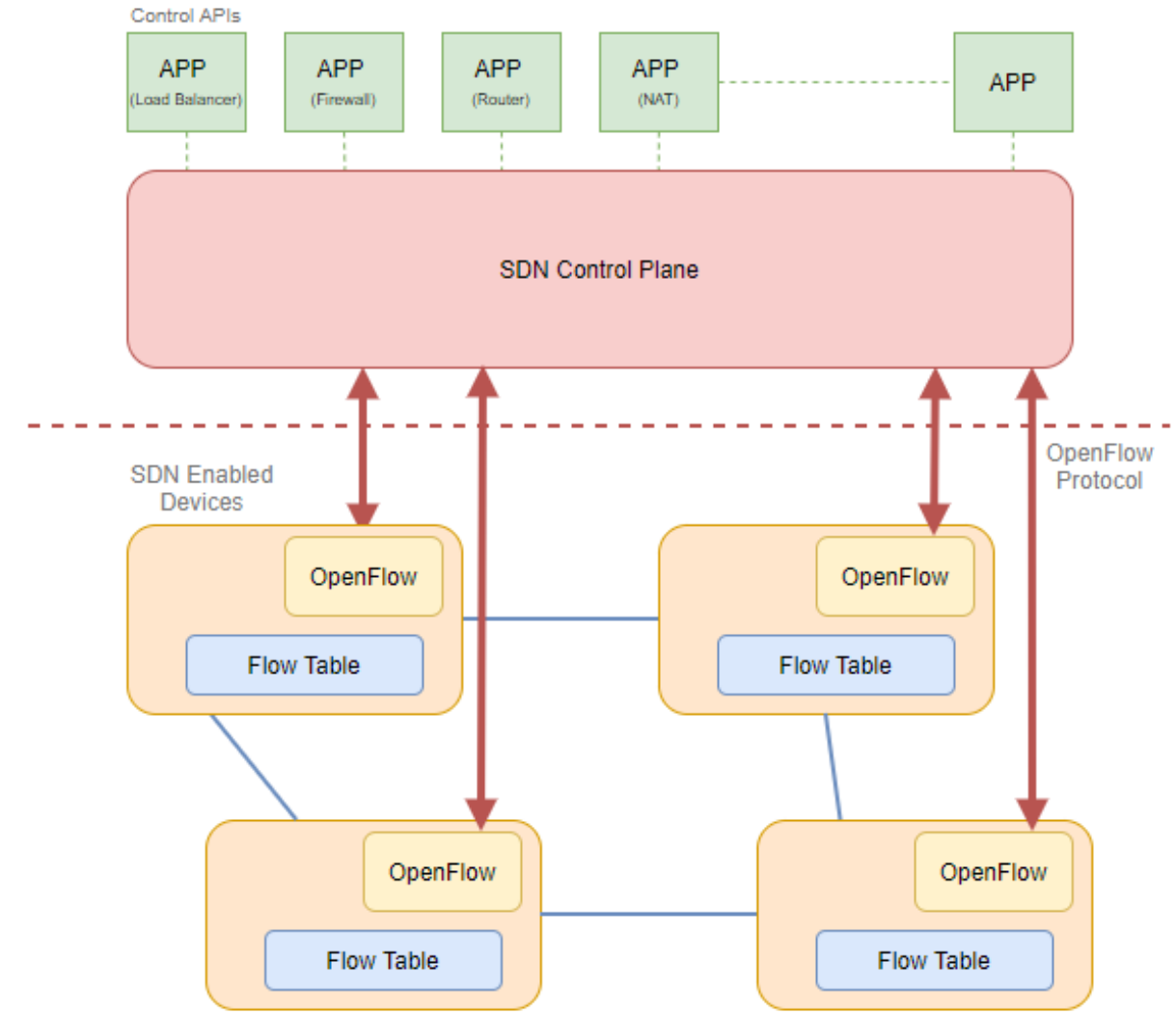


OpenFlow (OF)

- Most popular Southbound API
- Defines the way the SDN controller interacts with the Data plane
- It is a protocol that extracts the control of a switch to a centralized server
- Switches and controllers communicate to each other using the OF protocol.
- Can emulate various kinds of boxes (firewall, router, switch, etc.)

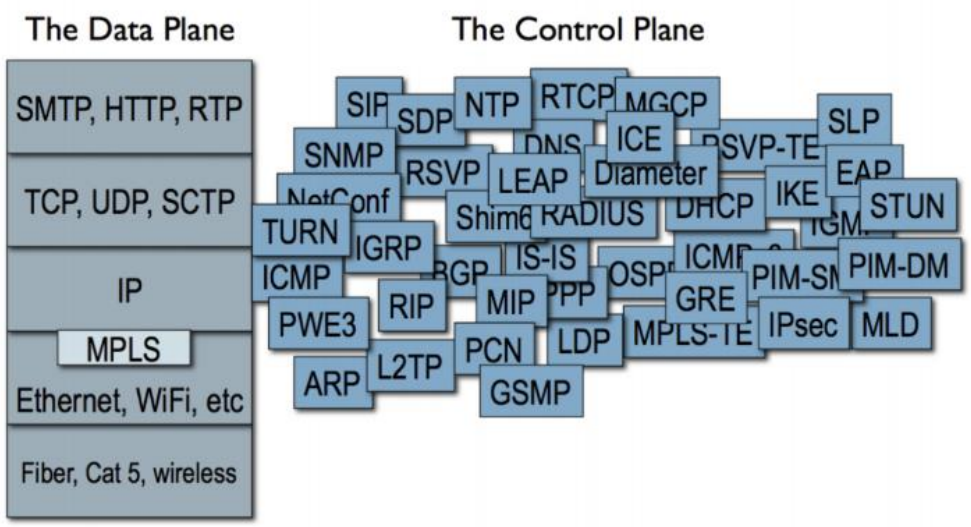
OpenFlow Applications (examples)

- DoS Attack Detection
- Network Virtualization
- Server Load Balancing
- Dynamic Access Control



OpenFlow (OF)

- OF expects the switches to have a fixed behavior (not programmable switch)
- Behavior of these networking devices cannot be changed (fixed)
- Protocol too complex – supporting complicated parsers and pipelines
- Specification complexity – extra features (from 4 header types to >50 today)
- Limited interoperability between Vendors.



Source Mark Handley. Re-thinking the control architecture of the internet. Keynote talk. REARCH. December 2009.

P4 *Programming Protocol-Independent Packet Processors*

Available: http://arxiv.org/abs/1312.1719

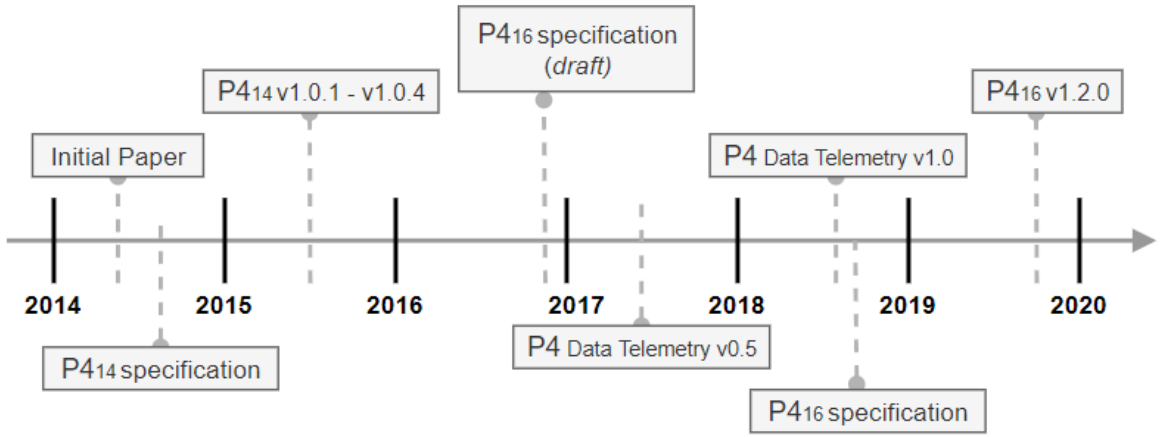
- What is it?

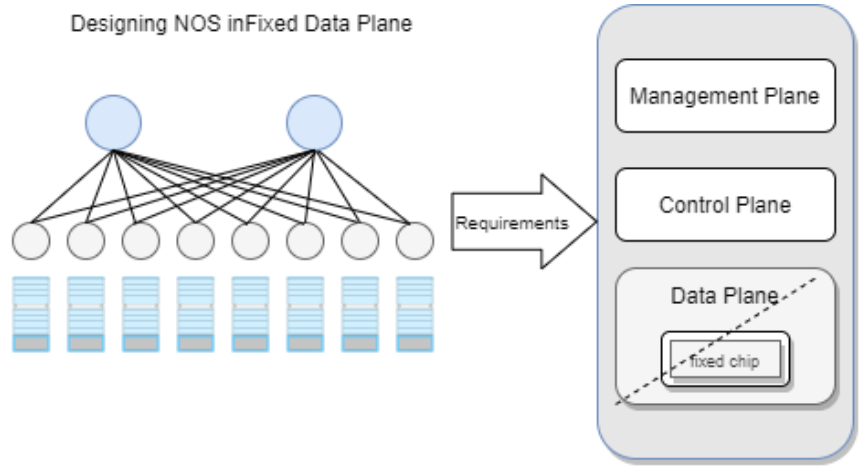
it is a high-level language expressing how the packets are being processed by the data plane of a programmable component such as software or hardware switch, router, network interface card or network appliance.

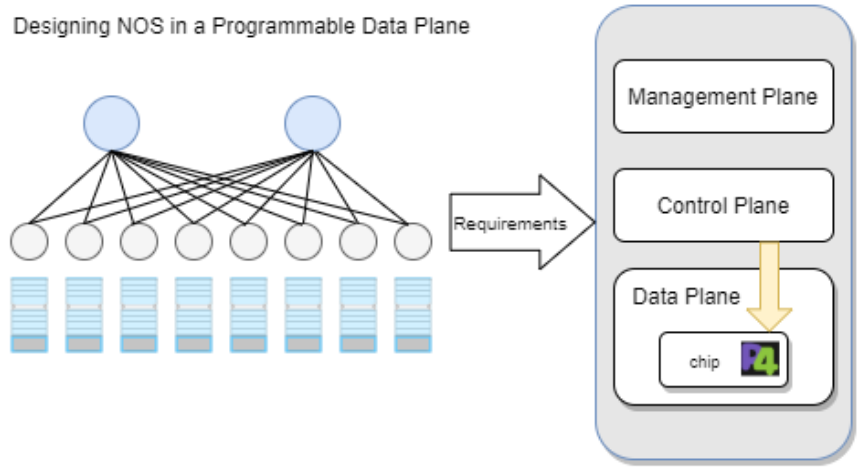
- Main goals:
  - Reconfigurability
  - Protocol Independence
  - Target Independence

- Benefits:
  - New Features (add new protocols)
  - Reduce Complexity
  - Efficient use of resources
  - Greater Visibility (INT)

"Think like a programmer rather than protocols"







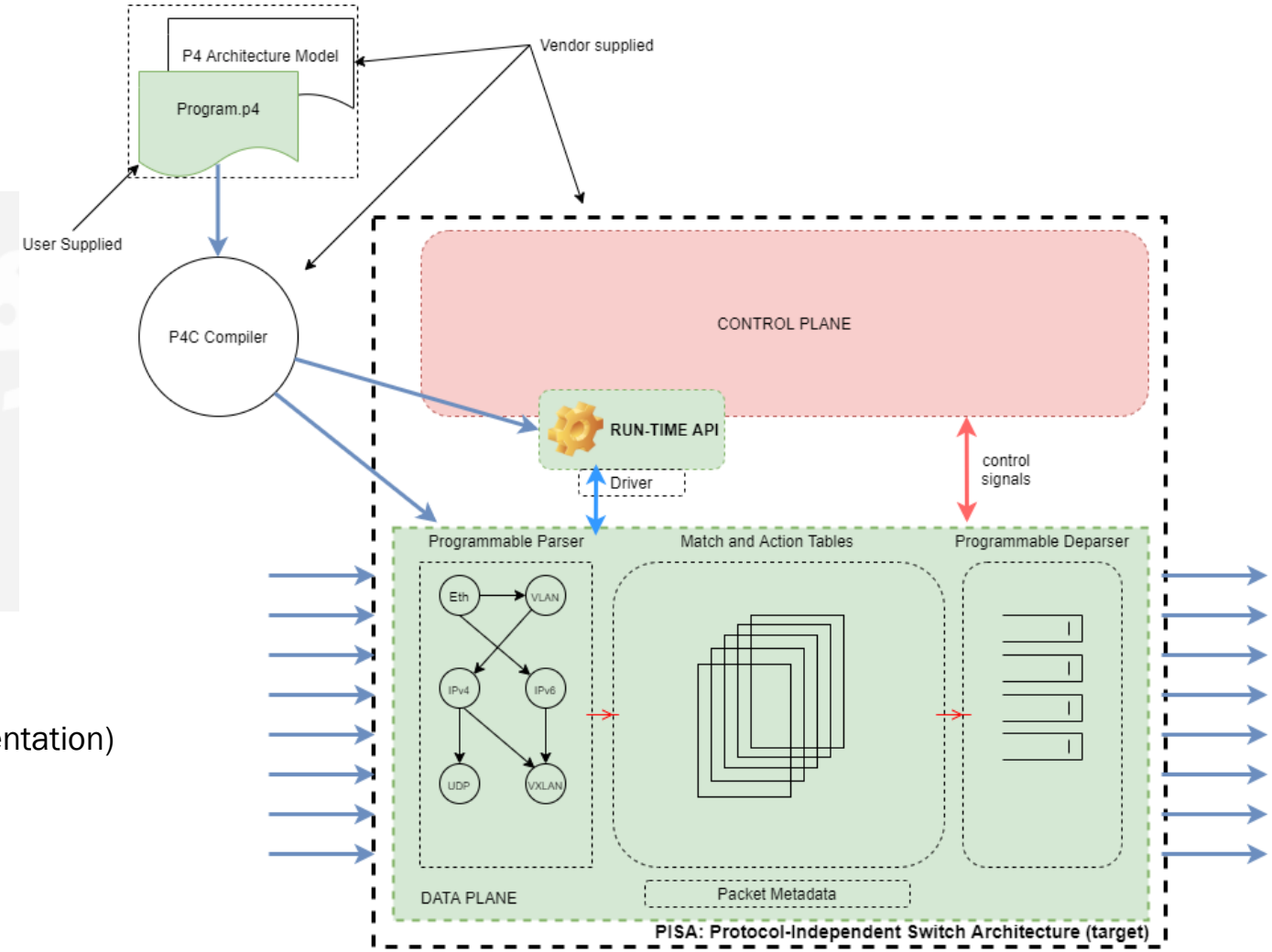
P4$_{16}$ Architecture

P4 is a domain-specific language which describes how a PISA architecture should process the packets (https//p4.org)

- *P4 Target – a model of a specific hardware implementation*
- *P4 Architecture – an API to program a target*

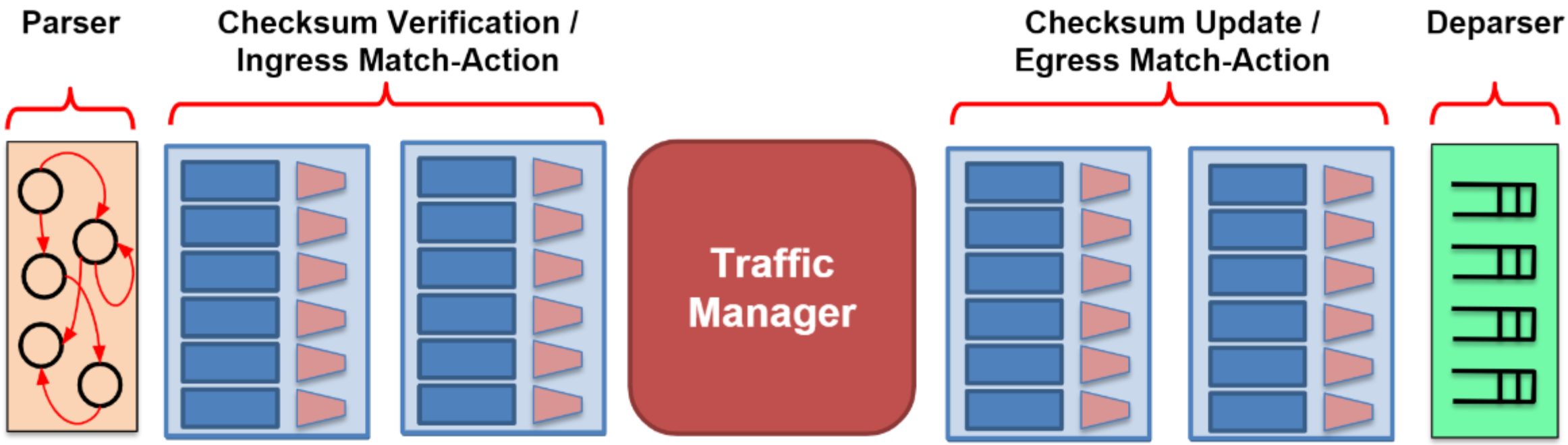
Key Elements:
- Code – P4 program (User Supplied)
- Architecture Model (Vendor Supplied)
- Compiler (Vendor Supplied)
- Target:
  - Control Plane (User Supplied)
  - Data Plane (Vendor Supplied)

P4 Concepts
- Pipeline:
  - Parser (converts packet data into metadata – Parsed Representation)
  - Match-Action Tables (Operate on metadata)
  - Deparser (Converts metadata back to serialized packet)
  - Metadata Bus (Carries the information within the pipeline)



P4$_{16}$ Architecture (V1 Model)

Implemented on top of Bmv2's *simple_switch* target



Source: https://github.com/p4lang/tutorials/blob/master/P4_tutorial.pdf

P4$_{16}$ *Language*

Language Elements:

- Parsers: Machine State, bitfield extraction
- Controls: Tables, Actions, control flow statements
- Expressions: Basic operations and operators
- Data Types: Bitstrings, headers, structures, arrays
- *Architecture Description: Programmable blocks and their interfaces*
- *Extern Libraries: Support for specialized components*

```
#include <core.p4>
#include <v1model.p4>
```
Libraries

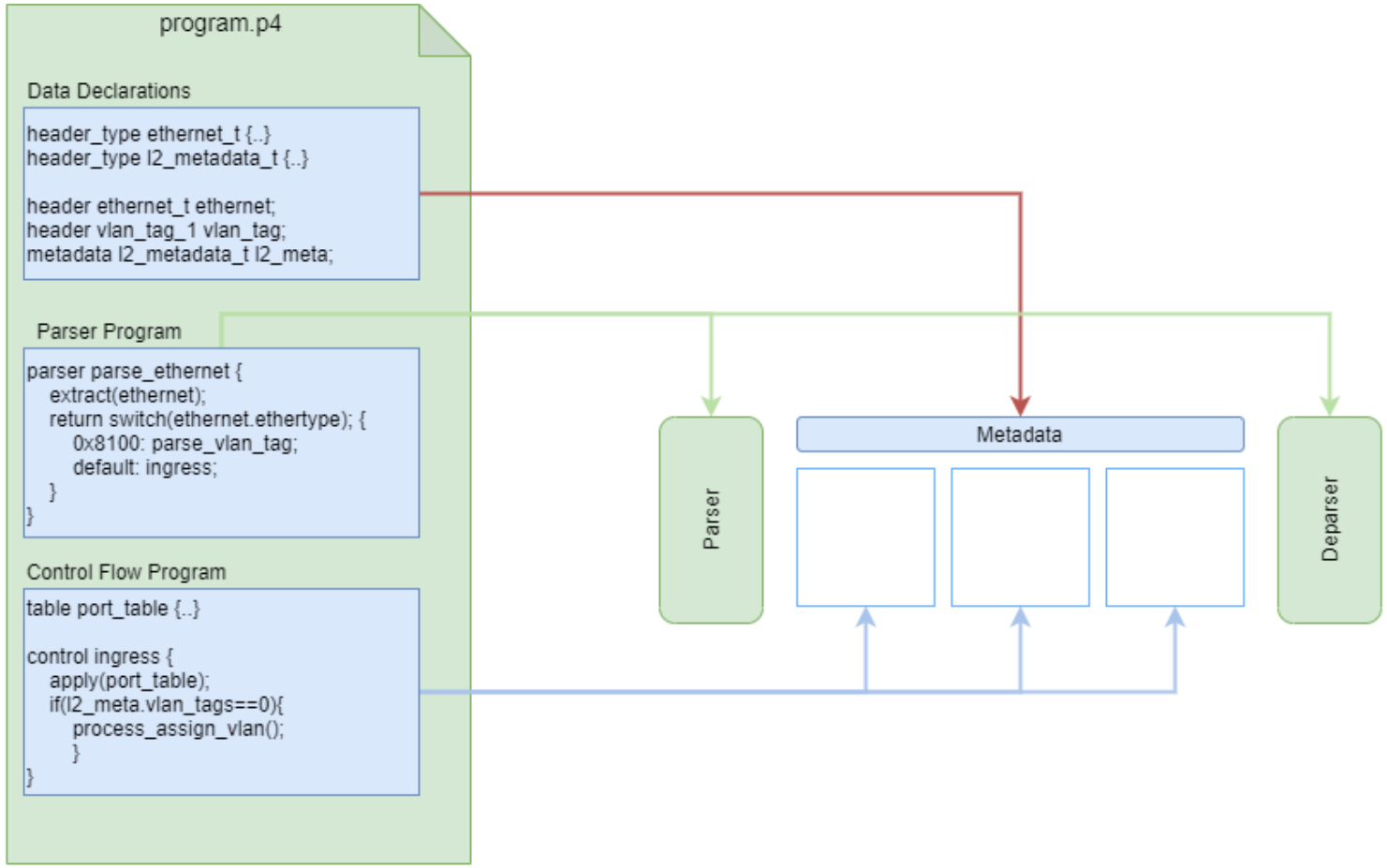
```
const bit<16> TYPE_IPV4 = 0x800;
typedef bit<32> ip4Addr_t;
header ipv4_t {…}
struct headers {…}
```
Declarations

```
parser MyParser(…) {
    state start {…}
    state parse_ethernet {…}
    state parse_ipv4 {…}
}
```
Parse packet headers

```
control MyIngress(…) {

    action ipv4_forward(…) {…}

    table ipv4_lpm {…}

    apply {
        if (…) {…}
    }
}
```
Control flow to modify packet

```
control MyDeparser(…) {…}
```
Assemble modified packet

```
V1Switch(
 MyParser(),
 MyVerifyChecksum(),
 MyIngress(),
 MyEgress(),
 MyComputeChecksum(),
 MyDeparser()
) main;
```
"main()"



Source: https://github.com/nsg-ethz/p4-learning/blob/master/slides/02_p4_env.pdf

In-band Network Telemetry (INT)

Available: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf

Today's network monitoring is :

- Expensive and Inefficient (Ping, SNMP, Traceroute, etc.,)
- Microburst can't be captured (100s ns – 10s μs)
- No visibility equals No Control

- What is it?

it is mechanism for collecting and reporting real time network state (INT metadata) directly in the data plane (in-band). The control plane is used only for decision making on what information the data plane will collect and for which flows.

Providing answers to the operators, for example:

1. which path did the packet take?
2. Which rules did the packet follows?
3. How long did it queue at each networking device?
4. Who did the packet share the queue with, who is aggressor flow?

By:

- Instrumenting metadata into the packet
- Without changing anything in the application layer.

What information we can add in the packet:

- Switch Id
- Ingress Information
  - Ingress Port
  - Ingress Timestamp
- Egress Information
  - Egress Port
  - Egress Timestamp
  - Hop Latency
  - Egress Port TX Link Utilization
  - Queue Occupancy
  - Buffer Occupancy

In-band Network Telemetry (INT) Modes

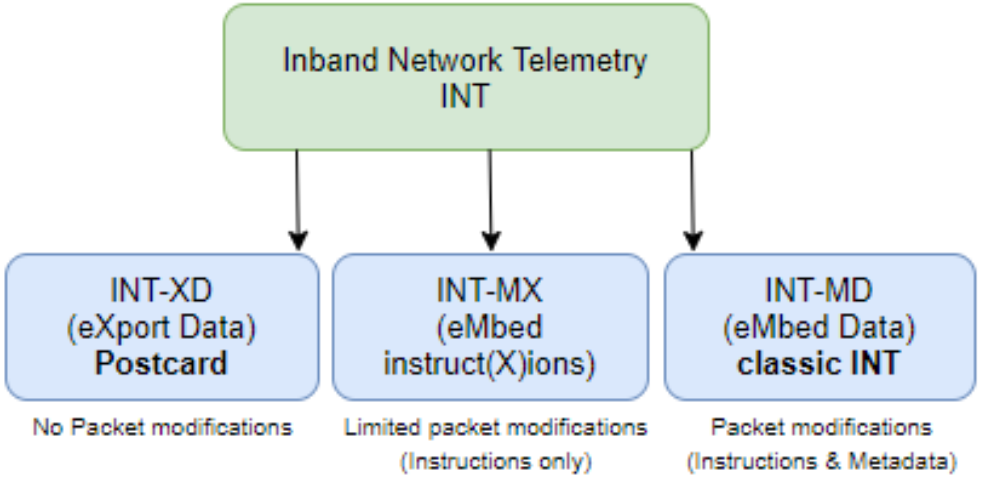
INT Modes:

- INT-XD Postcard (No Packet Modifications)
- INT-MD Classic INT (Packet Modifications)
- *INT-MX*

Key Terms:
– **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 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.

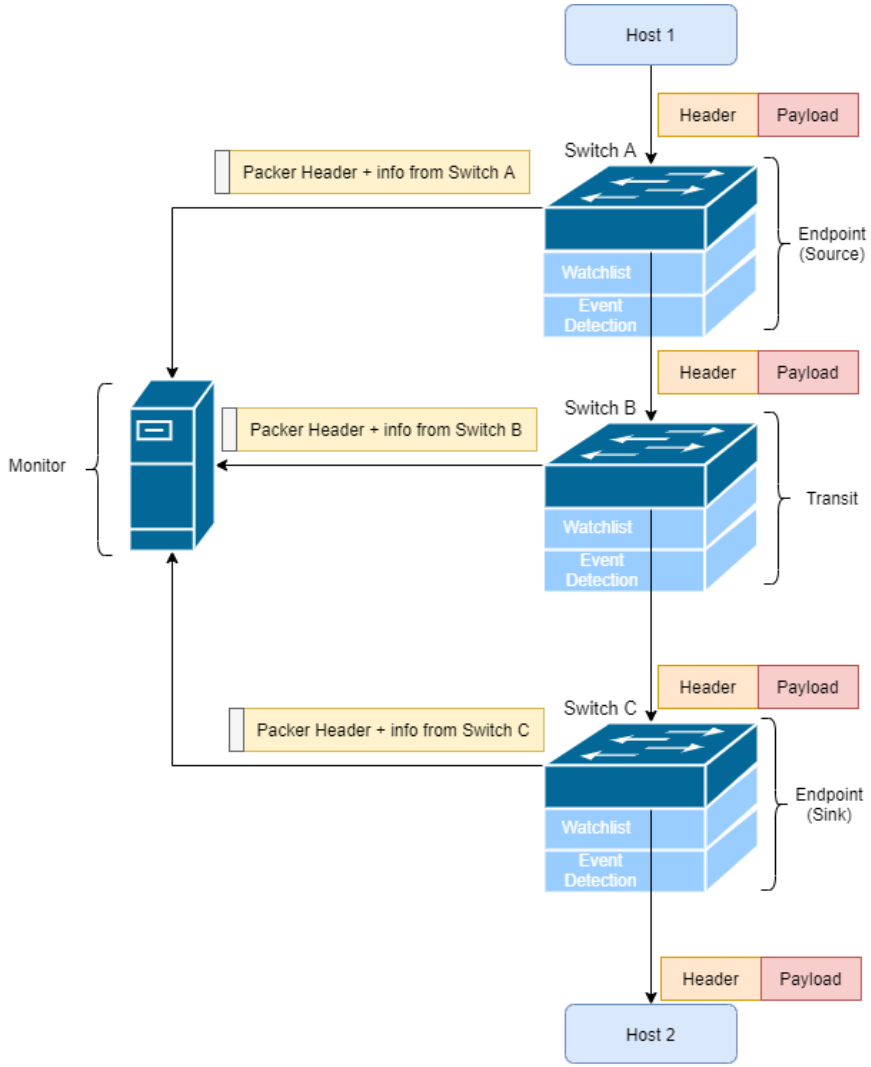


In-band Network Telemetry (INT) Modes / INT-XD, INT-MX

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* to the telemetry monitoring system without any modification of the packets. 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, 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.



- *Flow Watchlist: A data plane table that matches on packet headers and inserts or applies INT instructions on each matched flow. A flow is a set of packets having the same values on the selected header field.

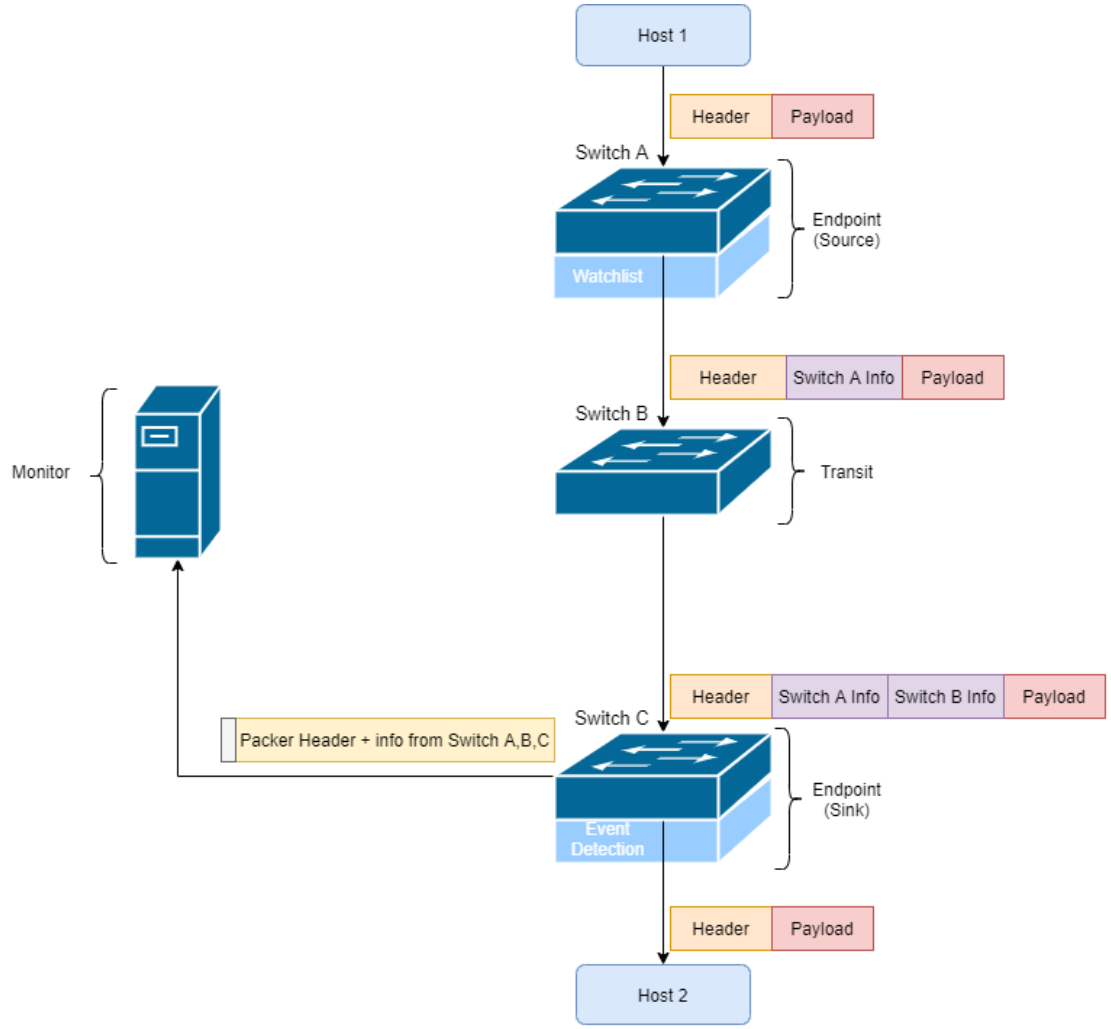
In-band Network Telemetry (INT) Modes / In-band (In-situ) Mode

INT-MD 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
- 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.

When the packet enters in the INT Source networking device (Switch A), 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 the telemetry information from all the INT networking devices.



Event Detector (monitor every packet but repot only what matters)

- Generate reports upon
  - Flow initiation & termination
  - Path or latency changes
  - Special field values
- Change detectors are reset periodically (e.g., once every sec)

In-band Network Telemetry (INT) / Headers

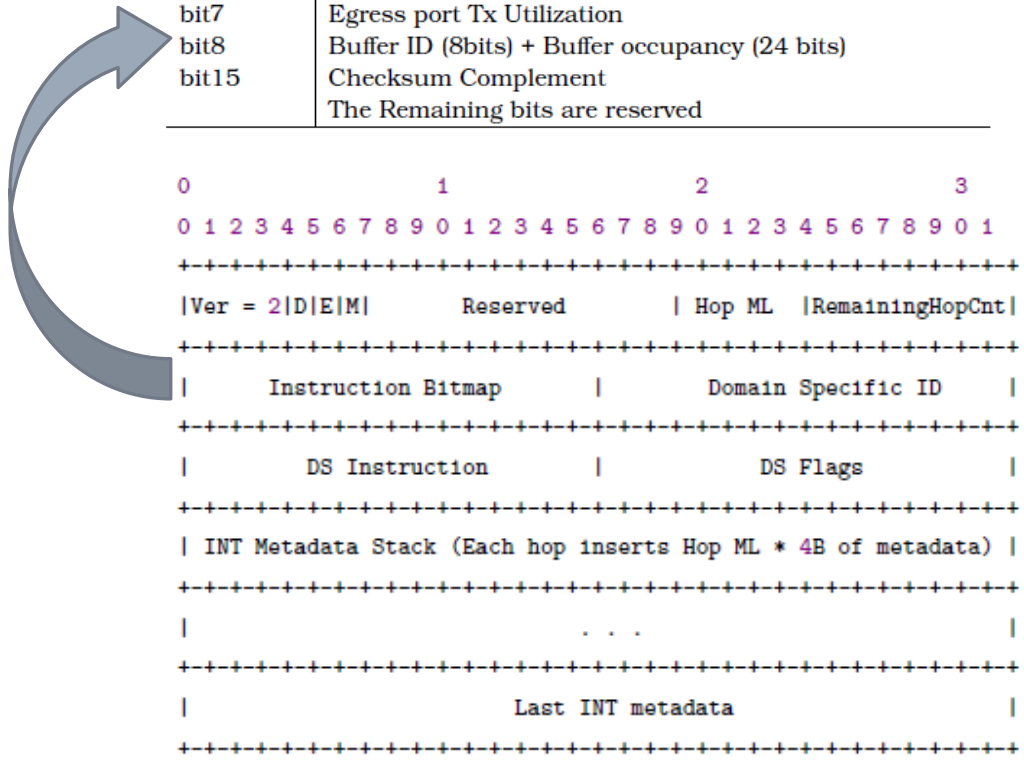
Three types of INT Headers exist: MD-type, MX-type and Destination-type.:
– *MD-type: This type of INT Header must be processed by intermediate devices*
– **Destination-type**: This type of header is consumed by the INT Sink networking device while the intermediate (INT Transit) networking devices ignore such type.
- **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.

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 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

The intermediated transit networking devices can set the following fields:
E, M, Remaining Hop Count, Domain specific fields

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



Experiments - Overview

Implement P4 In-band Network Telemetry which allows us to read queueing information, like the number of the packets waiting to be transmitted , the queue occupancy and use this information to identify congestion 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 switch (in our case this will be the switch before the destination host) will detect a packet which suffers from congestion, 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.

- Virtual Machine (4096MB, CPU:3, OS: Ubuntu 16.04 LTS)
- Mininet
- Wireshark
- nload
- Iperf3

Experiments - Experiment 1 (Queue Occupancy)

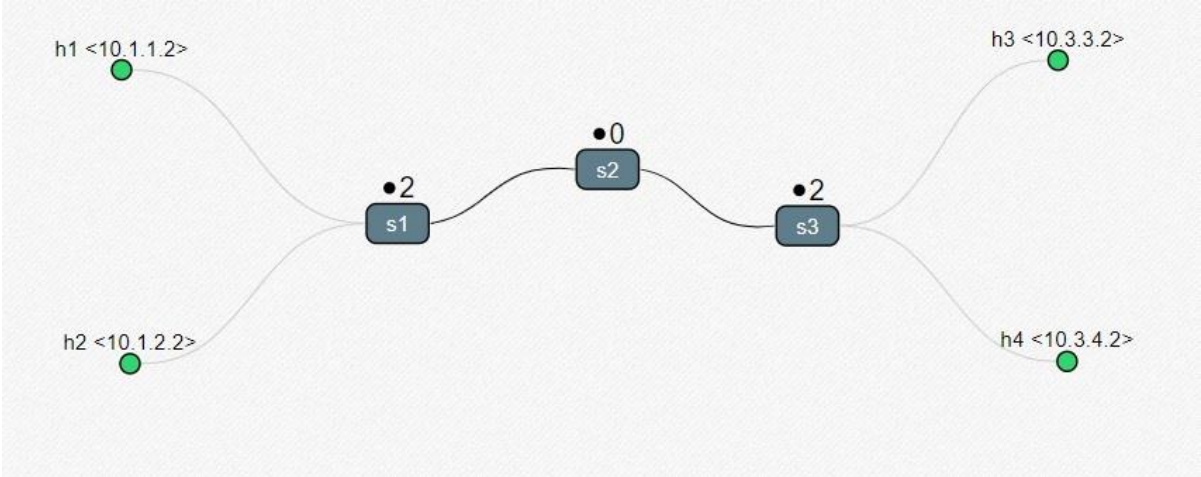
- Objective: Read Queue Information

- Set the telemetry header when the packet enters the network with a specified destination port (7777) and keep the telemetry header until the destination to show how the queue depth changes when we make flows to collide.

```
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;
      }
    }
  }
}
```

Switches are adding the telemetry header to all the TCP packets with destination port 7777. This telemetry header will carry the worst queue depth found across the path.

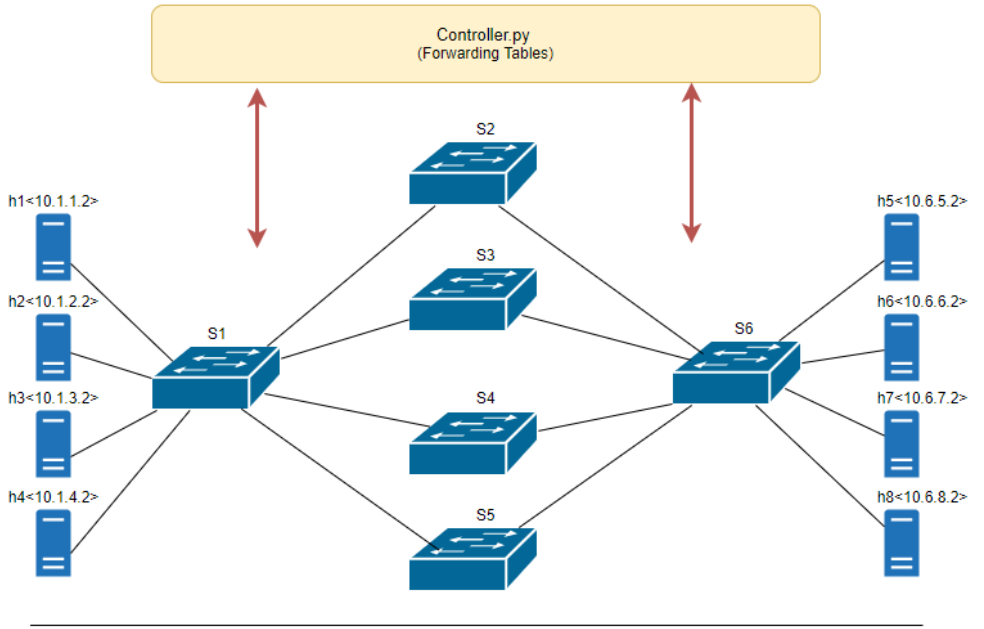
- Linear topology with 3 switches connected in series and 2 hosts connected to each extreme.
- Bandwidth: 10 Mbps/sec



Experiments - Experiment 2 (Collision Detection & Load Balancing)

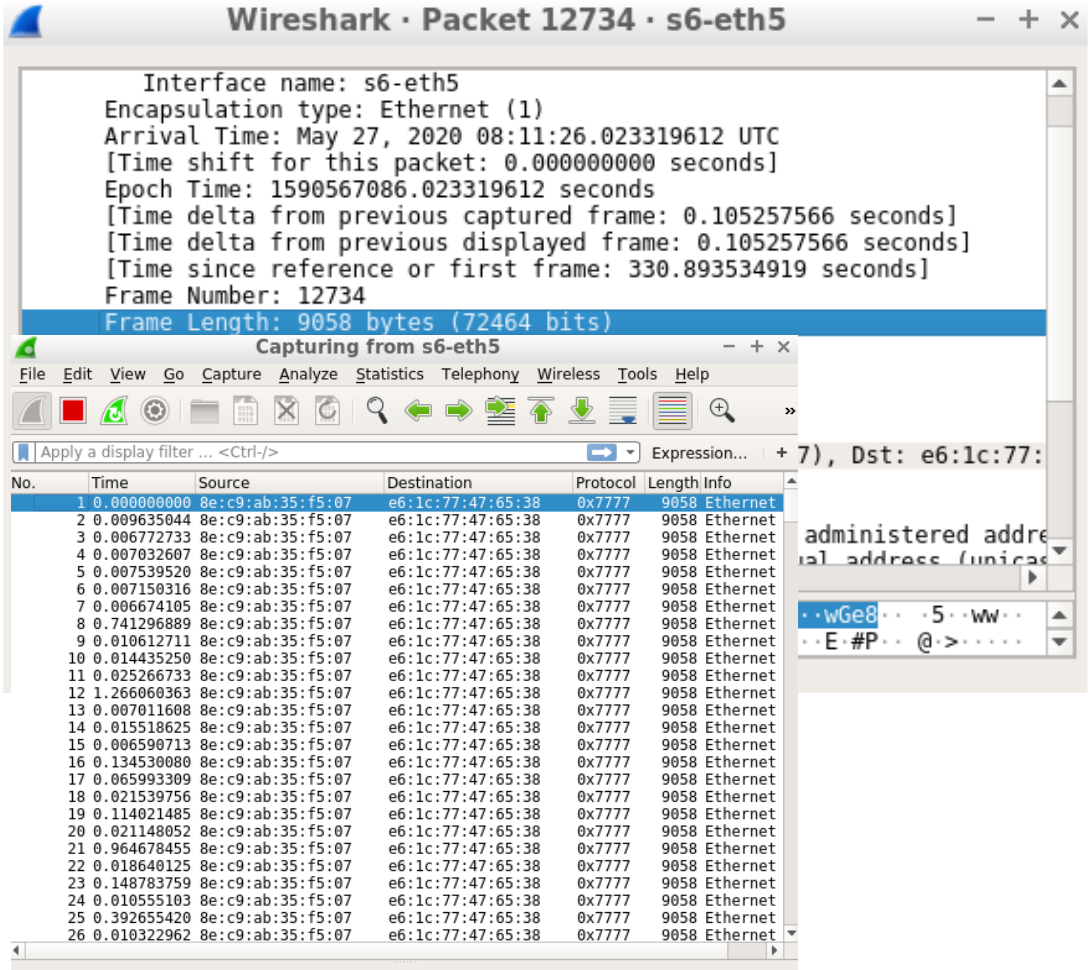
- Objective: Use the telemetry header to detect congestion and change the flow.
- The switches will make use of the telemetry header from the esoteric network to detect the congestion and move the flows, load balance the network and then before the packets will be forwarded to the hosts, we will have to remove the telemetry header (INT Sink= S6).
  - Specify which type of nodes (host or switch) is connected to each of the switch's port
  - Define a match+action table in the ingress pipeline, identify output port for each packet
  - Modify the egress, add logic :
    - When the TCP packets are entering the network, the switches will add the telemetry header and extract the telemetry header before arriving to the host
  - Apply the logic to detect congestions and when detected, send notification to the ingress:
    - Clone the packets that are triggering congestion
    - Modify the cloned packets and send back:
      - Check the queue depth if it is above the specified threshold (20 packets)
      - If yes, trigger the feedback message.
    - Add timeout per packet flow to avoid burst of packets
    - Add probability to avoid overflow of the new flow
    - Clone the packet – recirculate it.
    - Modify the Ether.type in the parser so the switches to identify the feedback msg.
  - Load Balancing: Ingress switches move the congested flows to new paths:
    - Notification packets which should be dropped (meaning the switch is sending the packets to the host) we save in a register identifier an ID value for each flow.
      Upon a congestion notification for a given flow, we update the register value with a new id (using a random number)

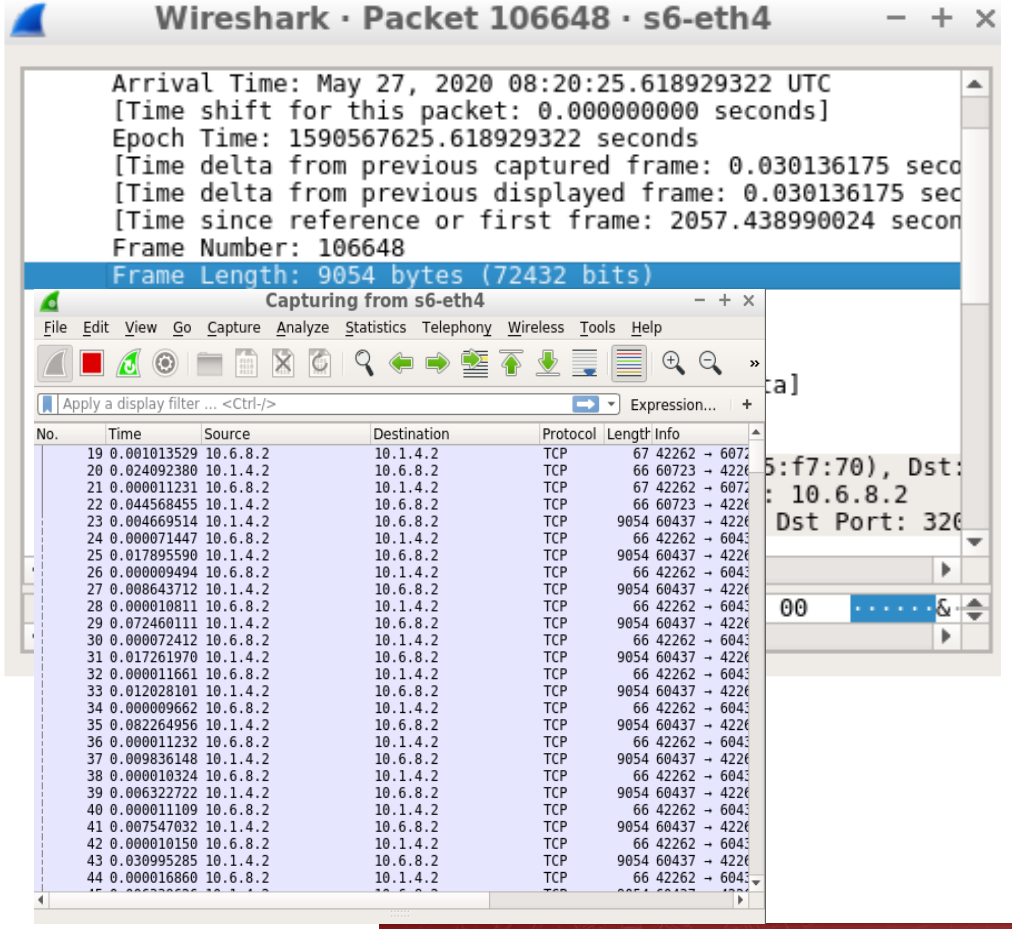
- Linear topology with 6 switches and 8 hosts
- Bandwidth: 10 Mbps/sec



Experiments - Experiment 2 (Collision Detection & Load Balancing)

- 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= S6).

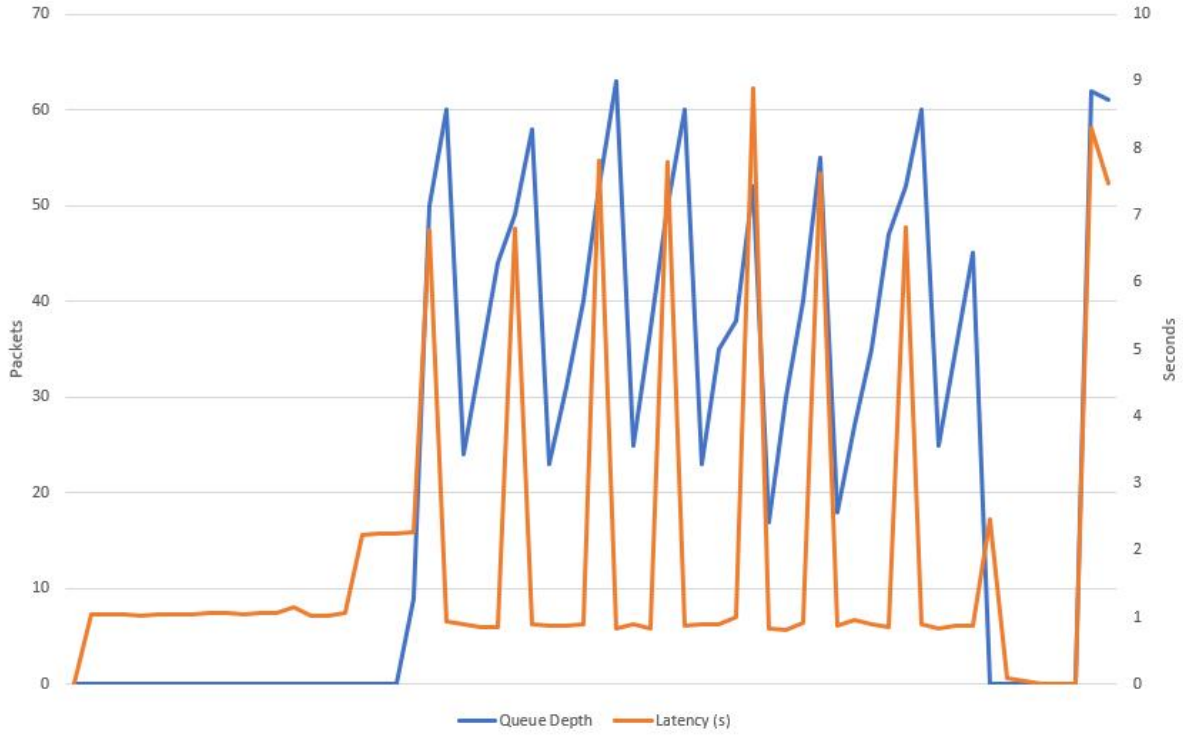




Results

- Experiment 1

We used a python script and the tool Scapy 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 Iperf3 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 is 64 packets) and the latency was increasing dramatically. We can observe the correlation between the latency and the queue depth.



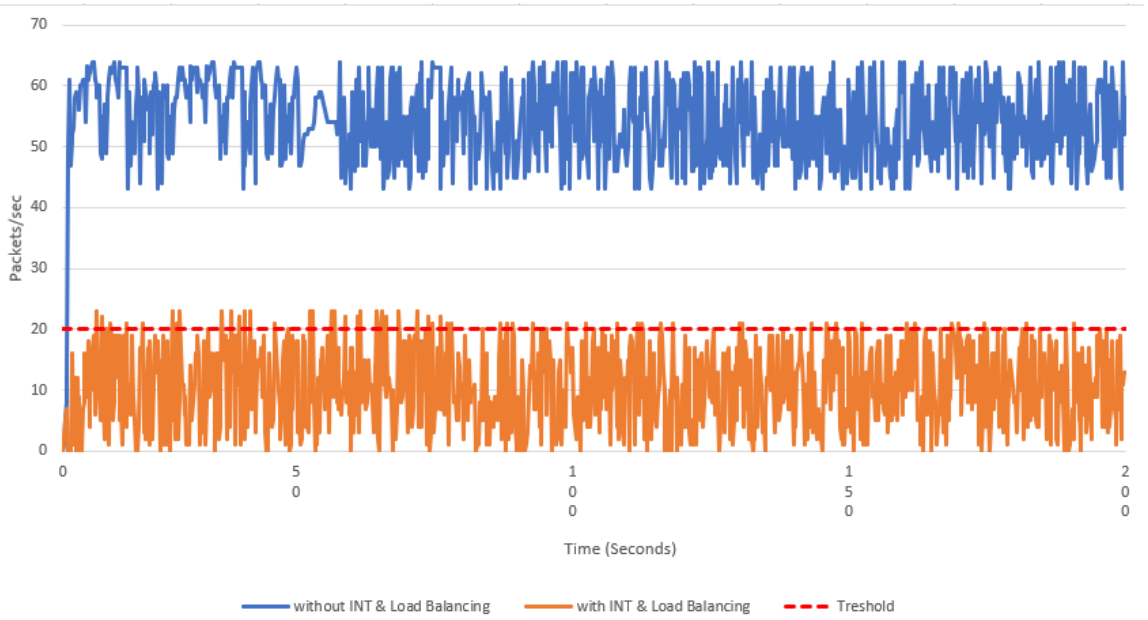
Results

- Experiment 2

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 (threshold to 20 packets), we generated and sent the same traffic to the network in order to compare between the two cases. By splitting the flows in all paths we increased the throughput significantly by avoiding congestions, from 3.7 Mbps/sec to ~10 Mbps/sec while in the same time the latency reduced from 94.2ms to 54.2ms.

Before

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



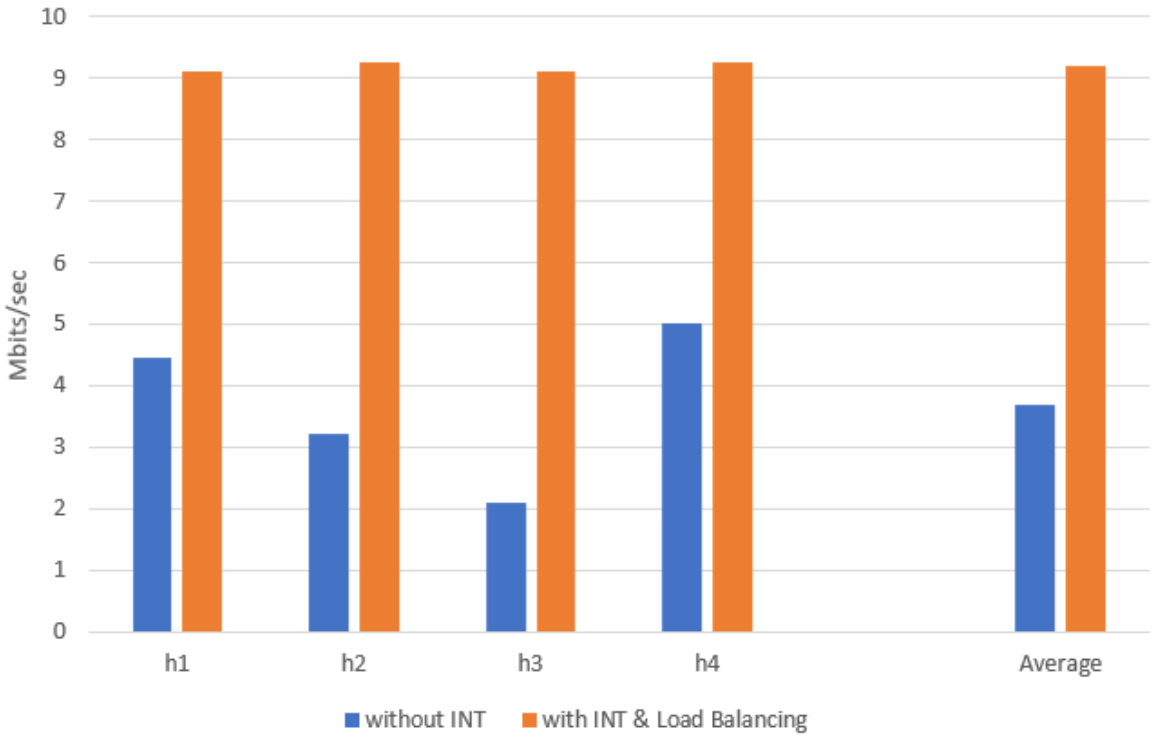
After

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

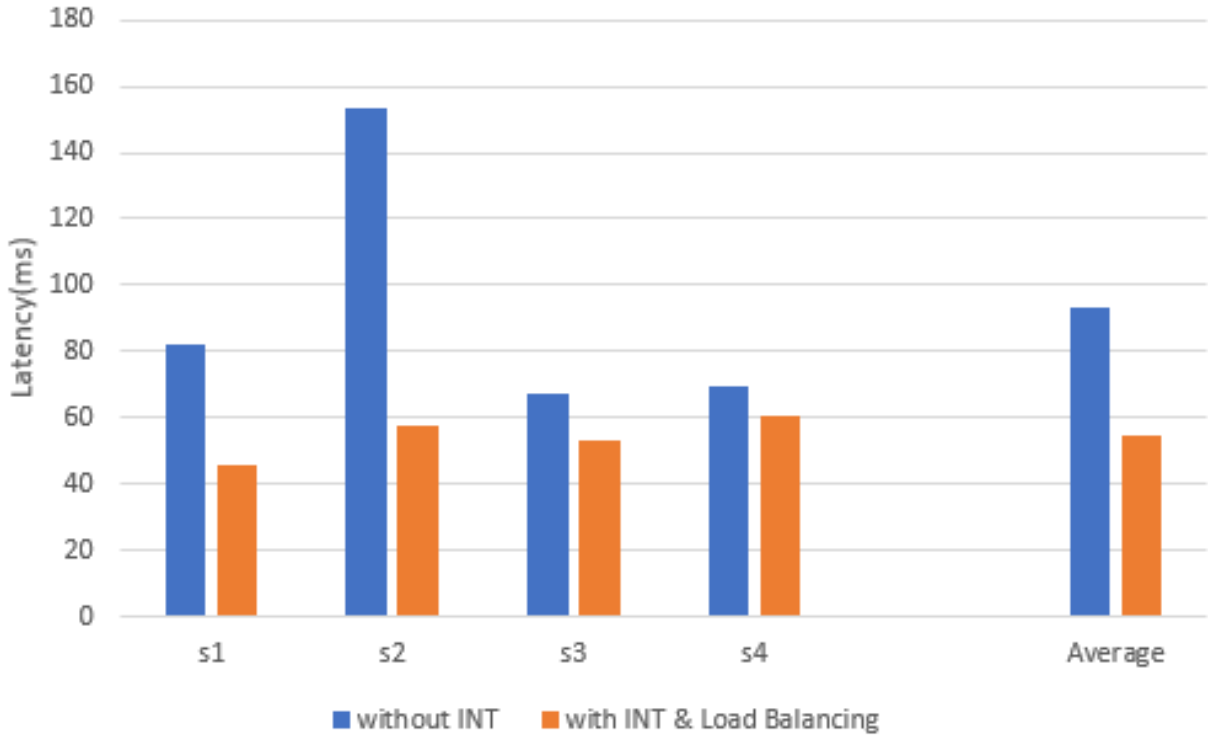
Results

- Experiment 2

Throughput



Latency



Conclusion & Future Work

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.

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).

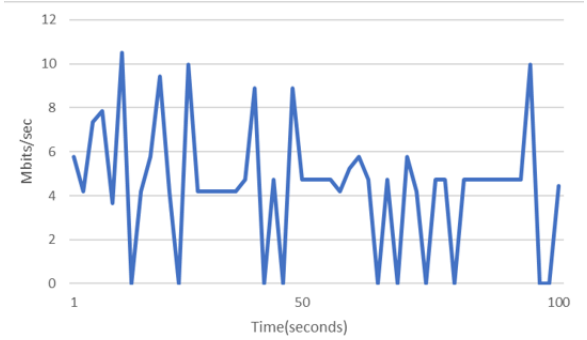
Further work is still required with the evaluation of Load Balancing in order to fully understand how different factors may affect the performance of the network.
In the future we will conduct experiments using more complex algorithms for realistic applications using hardware P4 programmable networking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

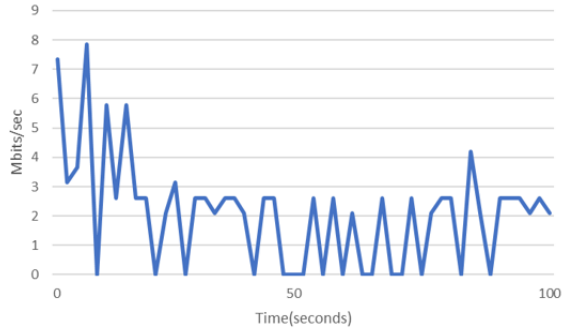
Thank you !

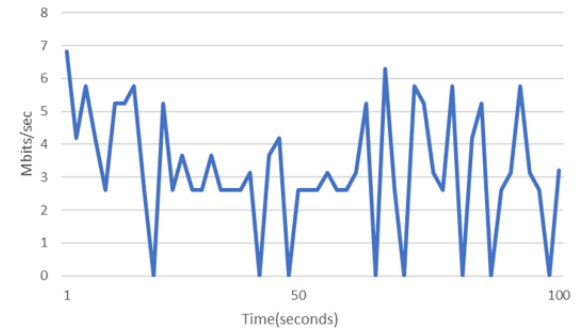
Results (extra)

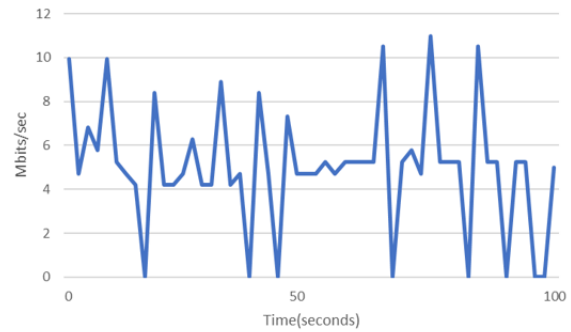
- Experiment 2

Before









After