**Intelligent End-To-End Traffic Congestion Trouble Shooting – using P4**

FINAL REPORT - Networks Project 236340

Spring 2020

**Staff : Itzik Ashkenazi, Alan Lo**

**Students : May Elbaz, Adi Sharon**

Contents

[**Introduction**  3](#_Toc48406994)

[Current Situation 3](#_Toc48406995)

[Our Mission 4](#_Toc48406996)

[**Timetable** 5](#_Toc48406997)

[**Background** 6](#_Toc48406998)

[Setup 6](#_Toc48406999)

[What will we do 7](#_Toc48407000)

[**Components** 8](#_Toc48407001)

[Physical Components 8](#_Toc48407002)

[P4 Language 9](#_Toc48407003)

[Docker containers 10](#_Toc48407004)

[P4 Build 10](#_Toc48407005)

[P4 Runtime 10](#_Toc48407006)

[P4 Code 10](#_Toc48407007)

[The ERSPAN Header 11](#_Toc48407008)

[P4 Table Entries 12](#_Toc48407009)

[IP Tables 13](#_Toc48407010)

[Traffic Generation 13](#_Toc48407011)

[Special configuration 14](#_Toc48407012)

[**Work flow** 15](#_Toc48407013)

[**Results** 17](#_Toc48407014)

[The GUI 17](#_Toc48407015)

[**Conclusions** 19](#_Toc48407016)

[Acquired Knowledge 19](#_Toc48407017)

[Challenges 20](#_Toc48407018)

[Future work 20](#_Toc48407019)

[References 21](#_Toc48407020)

# **Introduction** Networking tips: 5 networking mistakes we're all making - SEEK Career Advice

## Current Situation

**Networking** **hardware** today sits at the **basis of communication** in any data center, server room and office of all modern companies.

Troubleshooting this networking hardware is of the absolute essence, and when problems occur, they need to be **addressed quickly and efficiently**, as any network admin would tell you.

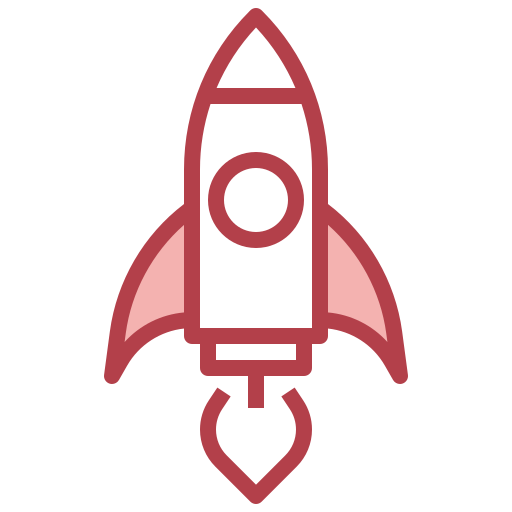
Data centers require **real-time & precise** feedback from their network equipment in order to make the troubleshooting process **simpler** and **more effective**.

The trouble today with the **standard switches** in data centers, is that they do not offer a lot of information about the state of the **traffic going through them,** Which makes it difficult to pinpoint an exact source of the network’s performance issues.

Mellanox has been developing a new line of highly capable switches, called Spectrum-2, supporting not only extremely heavy traffic loads, but also state of the art software support, bringing **hardware-calculated** metrics to the table and rendering the troubleshooting process simpler and more precise. It is a hybrid switch, incorporating all the convenience of a ready-made switching and routing protocols from its legacy capabilities, and all the agility of dynamically updating and reprograming without requiring an ONYX version update.

## 

## Our Mission

Every programmer’s best friend and worst enemy, is the debugger. It allows the user to troubleshoot the errors and malfunctions of a program. It’s a tool to see ‘what went wrong’ that is the most secure way to successfully achieve a solution.

Our mission is to create such a debugger, programmed to see the information flowing in the system and show the user where is the problem and who is at fault, using spectrum 2 switch’s capabilities and the P4 language strength and agility to provide the user all the necessary information to solve the network’s problem.

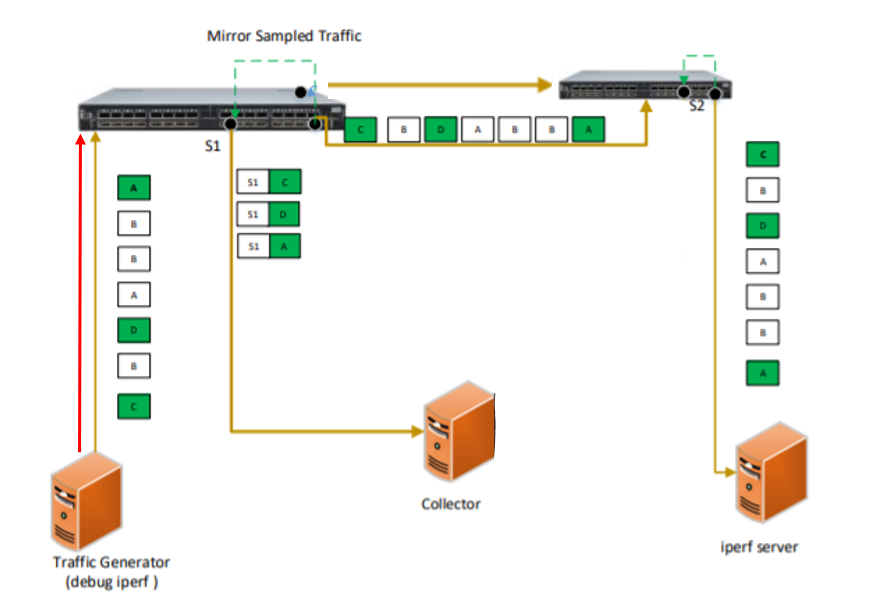
Our final goal – friendly user method to show the user the information analyzed- which port is congested, where is the congestion coming from, where is it headed, and more.

# **Timetable**

We set sail with a clear and goal-oriented schedule, which we were able to achieve.

# **Background**

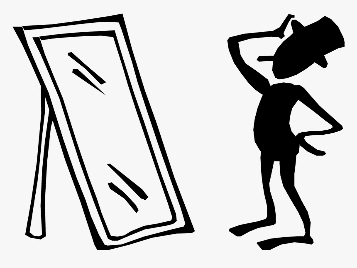
## Setup

We used the following architecture during our work on the project. Each component will be explained separately.

## What was done

We divided our mission into 5 stages of work, to achieve our final goal

* Sampling: We use the p4 programming language, to **sample** packets at a pre-defined rate, and this would be the first stage of the travelling packets going through our augmented switch. This way, we would not have to collect information through **every single packet** that goes through the switch, but only through a subset of them.



* Mirroring: We **mirror** those sampled packets on to a collecting unit. Mirroring is taking packets going from one port in the switch to another, and transmitting it as a whole, to another port.



* Tunneling: Finally, we could **wrap** those mirrored packets **with useful telemetric information**. Mellanox’s Spectrum-2 supports Generic Routing Encapsulation (GRE), a tunneling protocol developed by Cisco, that can encapsulate a wide variety of network layer protocols



* Analyze : Once holding all the necessary information and triggered by the user, experiencing congestion, the collector uses the mirrored information to analyze the source, destination, and other information arriving , separating wanted, user flows from congesting flows.



* Present: The information collected and analyzed by the collector is shown in a userfriendly GUI that presents the info, including the user’s 5-tuple, the congested port and the various flows that go through it

# **Components**

## Physical Components

**Mellanox SN3700 (Spectrum-2)**

The newer Mellanox switch model, supporting hardware-calculated telemetry, P4 language deployment, and more… It is essentially the core component of our project.

Our p4 program runs on this switch, updated dynamically using P4RT. The switch both forwards the packets to second switch and mirrors them the collector as well

**Mellanox SN2700 (Spectrum)**

An older Mellanox switch model, used to emulate a realistic network environment of multiple switches that are connected.

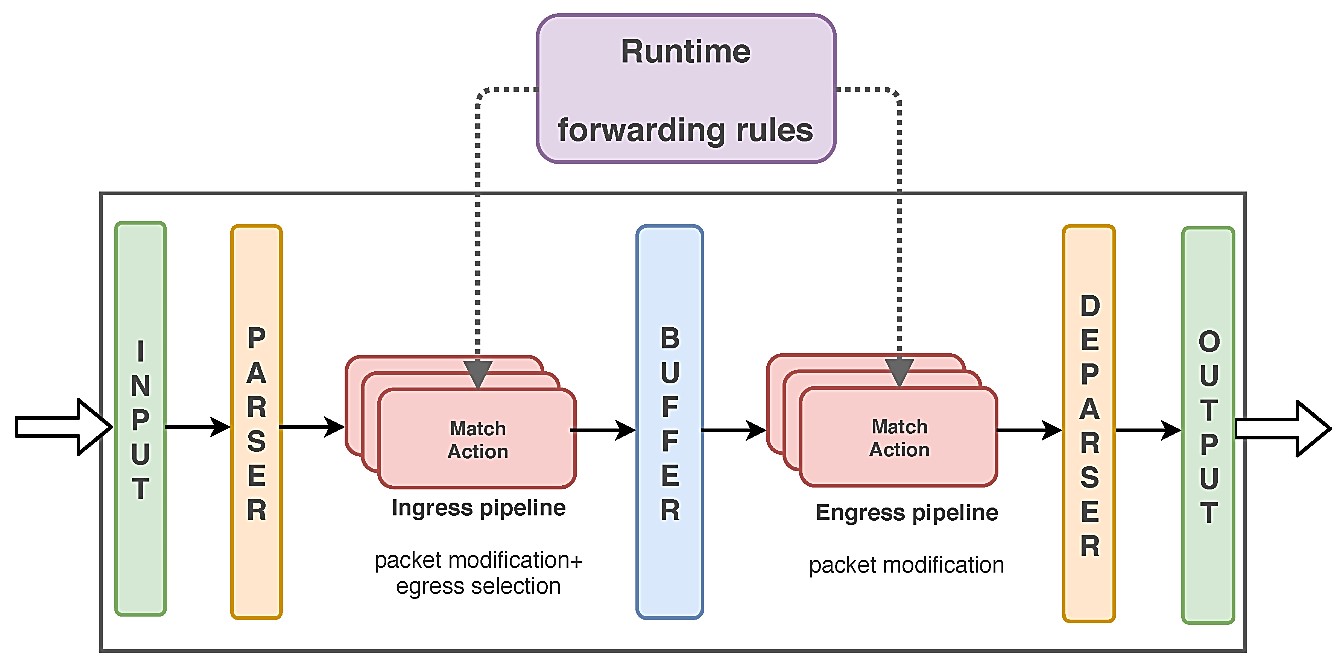
**Traffic Generator**

 A server with a highly capable 2-port Mellanox network interface. We used it to generate high-rate traffic that was directed to the SN3700 switch, in order to create a congestion over the port, and thr user’s desired flow. The application id triggered here to start sampling packets.

**Collector**

A receiving server that runs the GUI software, which in turn, receives the sampled packets, analyzes them and presents the information. Here we run a Docker container holding a P4RT shell environment. The collector sniffs the mirrored packets. It also establishes a connection to the switch using P4RT which allows it to add and modify table entries.

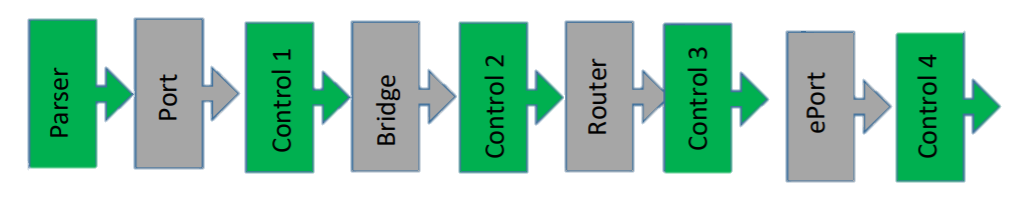
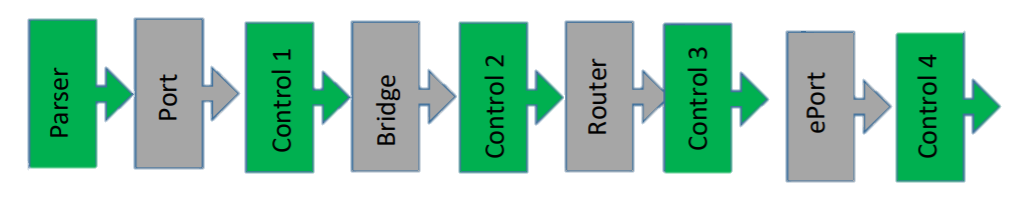
## P4 Language

P4 is a new, highly advanced open-source programming language that controls the packet forwarding planes in network devices, such as routers and switches. At any stage of the forwarding pipeline we can program the packet flow behavior.

The actions are defined through the primary component of P4 called **match-action tables**. Through these tables we can define and trigger actions when there is a match.

The P4 language is target-independent, meaning it can be compiled against different types of execution machines (called P4 targets). Each target must be provided with a backend compiler that maps the P4 source code into a target switch model.

P4 requires a compatible HW – a programmable switch or target, capable for P4 execution, such as spectrum SN3700



Our P4 capable switch includes programmable blocks. In our project, we used programmable block 5 (control 4) to insert our p4 code. Programmable block - egress port, has the ability to define chain of multiple match action tables. Supported actions are drop, egress mirror, packet modifications, counters and more.

### Docker containers

**Docker container** image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries **and** settings. We used 3 different docker containers running, for different uses.

First one on a VM, called p4 build, in our PCs for compiling and debugging our own p4 code.

Second docker on the spectrum-2 switch, called p4 RunTime, which receives the compiled output of our p4 code and loads it onto the data plane. It’s our connection to the p4 on the switch. There we also monitored our p4 program – the access list being added and removed; counters change as a match action occurred.

And lastly– a P4RT shell docker on our collector Linux machine. This docker required special configuration to allow it to create a P4RT friendly environment to run our GUI on, and also to access the host’s physical interfaces in order to sniff the packets mirrored from the switch. On this docker, we run our program.

### P4 Build

A heavy Linux-based OS environment used for the **development and compilation** of the P4 code. Installed on our personal machines, this environment provided us with a P4 compiler and other Spectrum-2 tools and scripts the helped us with the P4 development.

### P4 Runtime

A light Linux-like OS environment used for the **deployment** of the compiled P4 code. This would be installed on the switch, which supports docker containers. Once deployed, we get a CLI shell that supports multiple actions, allowing us to dynamically add, remove & modify table entries.

In order to make our program as dynamic as possible, we used an open source **P4Runtime-shell** in order to create new table entries that match the specific egress port that is congested.

### P4 Code

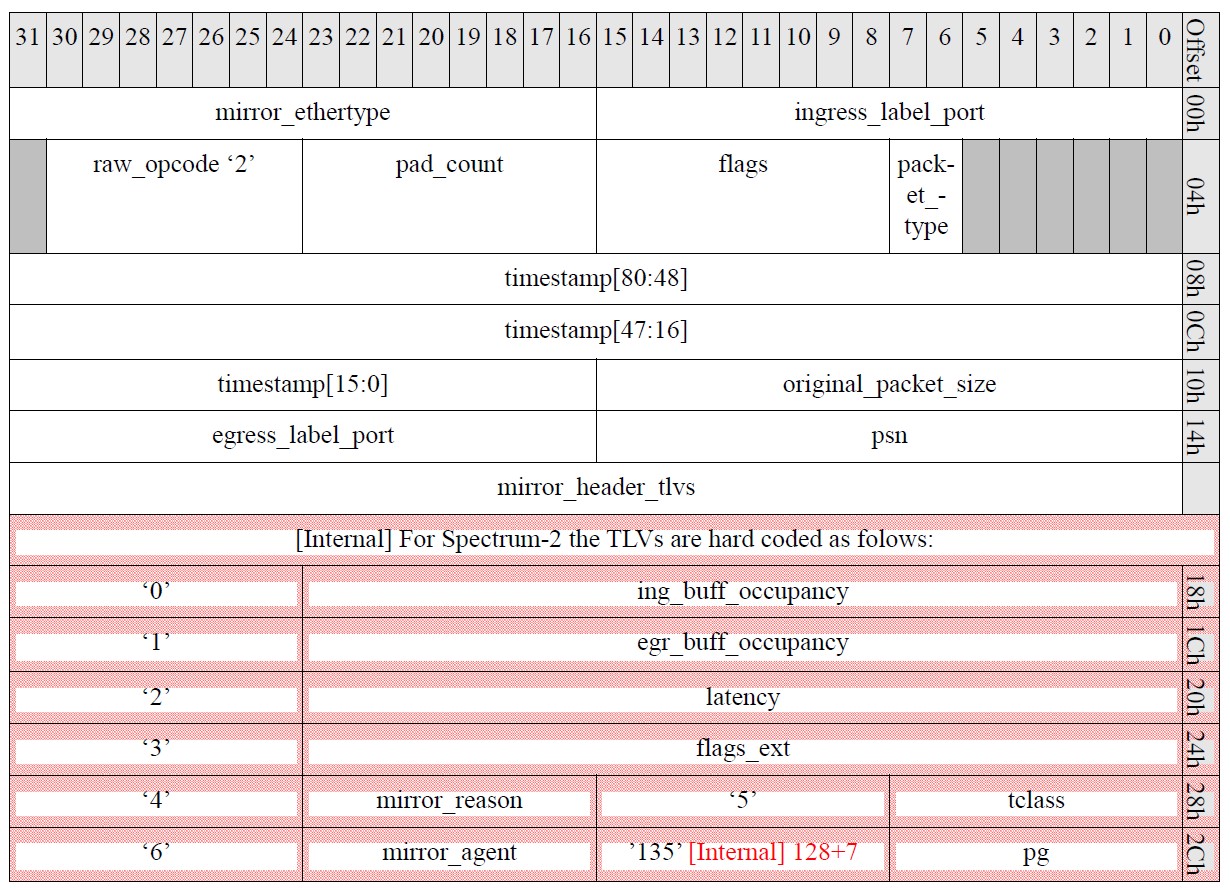
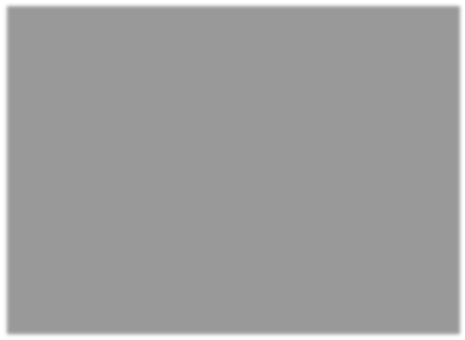
the **P4 code can add match-action tables to the data plane pipeline.**

One of p4’s main feature, is the support for **match-action tables.**

Match-action tables are mappings between agreed-on field values to a given set of action. That way we can look at certain fields in any packet that goes through the switch and determine whether or not to perform an action on it. For example, we can mirror a packet with additional information, by inspecting the DSCP, egress port and checksum.

### The ERSPAN Header

The ERSPAN header is Mellanox’s implementation of a **GRE header**. Generic Routing Encapsulation (GRE) is a tunneling protocol that can encapsulate a wide variety of network layer protocols inside point-to-point links over an IP network. The ERSPAN header adheres to the GRE protocol, allowing for telemetric values and other packet metadata to be transmitted with the packet. Below is a visual representation of the ERSPAN header.



**Figure 3 - Mellanox ERSPAN-2 header (marked in red are Spectrum-2 specific additions)**

For our final interface, we extracted the following metrics from the header:

* **egress\_label\_port** – to present the egress port of the mirrored traffic.
* **ingress\_lable\_port** – to present the ingress port of the mirrored traffic.
* **Source IP, Destination IP, Transport layer protocol, Transport layer source and destination ports** – parsed by the collector from the encapsulated packet, and used (as a part of the 5-tuple) to identify each individual flow, as it is later represented in the GUI.
* **egr\_buff\_occupancy** – to draw the egress buffer occupancy and determine if the port is congested.
* **TOS value** – parsed by the collector. This fields’ value is used to identify the users’ flow out of the congesting flows that go through the same egress port.

### P4 Table Entries

The P4 code is written and deployed, the setup is complete and functioning and traffic is flowing through the switch. Now all we need to do is to propagate the table with desired entries.

In order to achieve our mission, we used two distinct table entries:

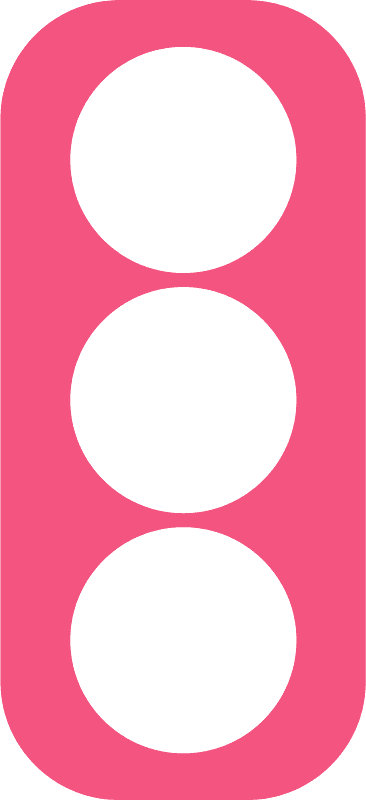
1. **DSCP mirroring**This constant table entry is hard-coded into our P4 program and is deployed (e.g. – an entry is added to the switches’ ACL) as soon as the GUI program is launched. This entry defines a mirror action that’s triggered by a specific match to a DSCP value coded into our P4 program, which means that only packets containing that specific value will be mirrored, encapsulated in the above mentioned ERSPAN header and sent to our collector to be analyzed and presented. We made sure that only the user’s traffic will contain the required DSCP value by using *IP tables*, as will be explained below.
2. **Egress port & checksum value mirroring**This table entry is dynamically added to the switch as a part of the user’s interaction with the GUI. It is dynamic in order to allow the program to match on a value that may vary – in our case, the users’ egress port. This entry matches on two values, in two different ways:
   1. **Egress port**  
      After our program has identified the users’ path through the switch (ingress and egress ports) and determined that a congestion is occurring on the switches’ egress, we now need to identify *who* is causing the congestion – meaning, who is going out the same port as our user. Therefore, we created a specific match on the egress port, and required it to match to the users’ egress port that we’ve identified from the ERSPAN information extracted from the reflected packets that have arrived because of the DSCP entry.
   2. **Checksum value**  
      In order to mirror a fraction of the user traffic, we’ve conditioned the mirroring of packets of that same egress by a sampling rate determined by the checksum value. Sampling rate is determined using the checksum field. This field is set as a ternary match, meaning we provide a mask and a value to match on, using the combination ‘MASK&&&TERNARY’. For example, in our program, the entry ‘0x7f00&&&0x7f00’ would match 7 bits randomly, providing a rate of 1:2^7=1:128 packets.

### IP Tables

Iptables is an extremely flexible firewall utility built for Linux operating systems. Iptables uses policy chains to allow, block or modify ingress/egress traffic.

Using this powerful tool on the traffic generator (as it also functions as our user computer), we created a python script that, upon receiving a specific application name, adds an iptables rule to mark each egress packet with DSCP value of 26, which is the same value the P4 entry matches on. The program is run by the user on his PC, and when the users signal it to do so, it can also remove the added iptables entries.

### Traffic Generation

Our goal here is to emulate a **real network environment** so that we can test the true functionality of our project. The SN3700 is a high rate switch (with each port being able to transmit and receive at a 100Gbit/s rate).

In order to be able to congest such ports, even with low rate shapers installed (~1Gbit/s, as is described below) we must be able to:

* Generate and send traffic at high rates
* Generate random traffic

o Required for having precise sampling at assigned rate

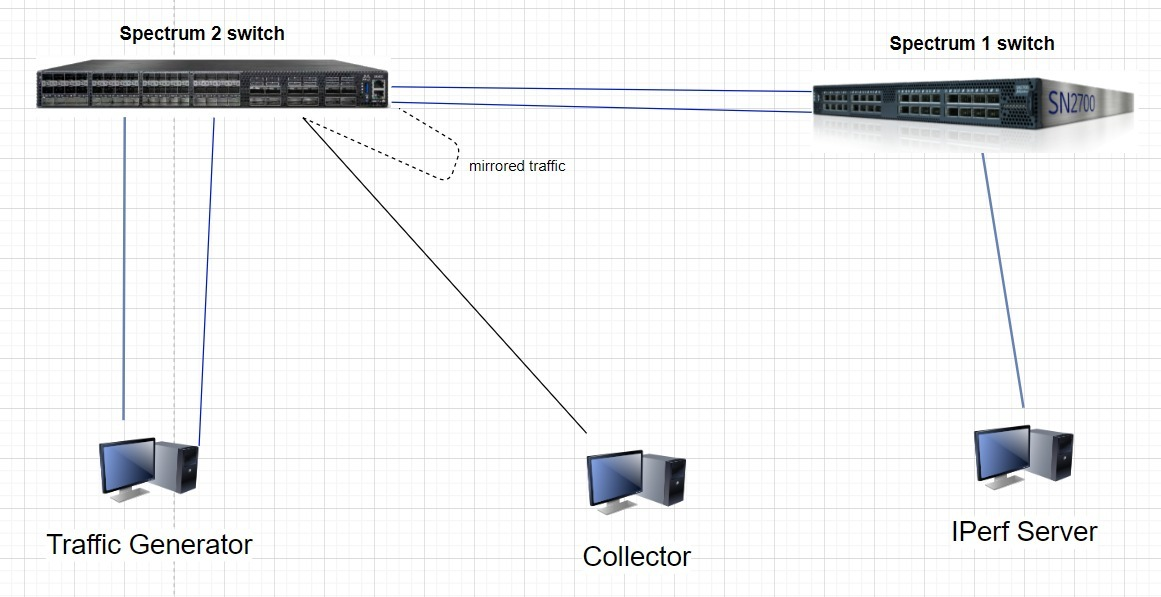
o Good for true real-environment emulation

Using basic applications or libraries to generate traffic (e.g. Scapy) we were able to utilize the sendfast function in order to congest the port that connects the switches. However, in order to simulate an actual network environment, as well as to create a traffic that is varied so it’ll be reflected by our checksum sampling, we also had to use a script that generated traffic from a pre-recoded pcap file. So, eventually, we used to two different scripts simultaneously – one a pcap file that emulated real traffic, and one to create synthetic, congestion traffic.

### Special configuration

In order to create a congesting event, we had both generated enough traffic (as described above) and configure our lab switch as follows:

* **Bandwidth limit** – in order to ensure congestion, we’ve limited the bandwidth on the port between the switches to ~1Gps.
* **VLAN manipulation** – to make sure that our users’ traffic goes out the port that we’ve limited, we used the advantage of the traffic generators’ dual port NIC, using one for the user’s traffic, and the other for the additional congesting traffic. We assign the switch port toward the user to vlan 4000, and the congesting flow to vlan 1202.  
  Between the switches, on the spectrum2 switch, we configured the limited user port to trunk both vlan 1202 and vlan 4000, and the second connection to allow only vlan 1202. On the spectrum1 switch, we configured the user’s connection to allow only vlan 4000, and the other connection to allow only vlan 1202, thus insuring a congesting event without creating a loop.



Eth 1/32 – Vlan 4000 + 1202

Eth 1/29 – Vlan 1202

Eth 1/3 –  
 Vlan 4000

Eth 1/25 – Vlan 1202

# **Work flow**

The event’s occurring managing our application is as following:

* A congestion event is occurring. We hard-coded our threshold for congestion to be 3000 on the buffer size.
* Host -side app (on the traffic generator) reports packet loss, causing the user to manually activate the debugger on the client side.
* The debugger starts marking the packets of the relevant app (1 of Xth)
* The marked packet (green squares) reaches the spectrum-2 switch, which matches it according to the DSCP value. Every matched packet is copied, encapsulated (by GRE with ERSPAN header, and additional metadata), and mirrored to the collector.
* The collector uses this information to identify the user’s 5-tuple.
* The collector dynamically adds a match-action using P4RT in order to match the congested traffic (by egress port).
* The collector uses the information to differentiate the user’s flow from the congesting flow
* The collector presents the information to the user (with a GUI)

# **Results**

### The GUI

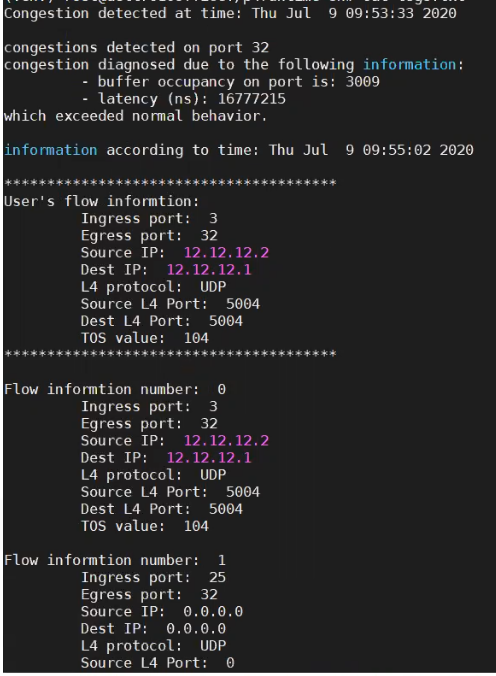
This is our final product. Once the congestion was spotted and packets were marked, the user can activate the GUI to see all the wanted information.

The GUI is written in python and gives the user the ability to get the relevant information

* Show user path: Displays the user’s 5-tuple (not the congesting flow)
* Analyze congestion: If the program identifies a congested port, this button uses P4Runtime to add a table entry that matches dynamically on the congested port specifically
* Export logs: creates a dated, ordered and clear summary of all the information shown by the GUI
* Separate flows: displays all the different flows that’ve been detected on the congested port.

GUI’S Log file:

**Our GUI log file, clearly dated and detailing the congestion detection cause, the users’ flow the identified congesting flows.**

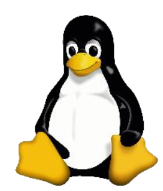


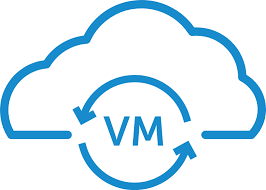
# **Conclusions**

## Acquired Knowledge

The project taught us plenty by both introducing new technologies and improving our knowledge in ones that we already know.















We’ve substantially improved our Linux skills, which was our main platform. We’ve taken our Python skills to the next level by writing scripts, and learning how to build our own GUI. We were introduced to technologies like P4, Docker, Spectrum, VMs and more. We also learnt how to work from afar, both from each other, from our contractures and the physical set up, during the Corona time.

The acquired knowledge will with high probability be beneficial to our careers wherever we go!

## Challenges

As every project, we have had a lot of challenges along the way. Some were expected, and some were not.

Going in we knew that p4 language is new and innovating, and this is one of the things that drew us to this project. Lack of online information and documentation made working with it challenging and educating. We appreciate Alan’s help and guidance on this matter.

Docker usage was also new to us. Combining different scripts, environments and HW components to work together dynamically was not always a simple task.

On the unexpected side, however, we handled a set-up change in the middle of the project because of the lack of the second switch due to corona virus influences. We had to do a lot of changes in our set up, requiring a unique asymmetric layer 2 configuration (Detailed under “Special configuration”), and as a result we also had to change code that was already compiled and deployed.

Furthermore - working from far with no physical access to the set up and the lab was also not always easy while working on a project of this type. Required a long distance assistance- mostly on Itzik’s expense.

## Future work

We’ve drafted several suggestions regarding which steps should be taken after this project:

.

Make the application self-triggered

Improve the project to work on a larger,

more sophisticated set up

Add a dynamic control over the congestion’s threshold

# References

<https://gitlab.cs.technion.ac.il/lccn/w2019-postcard-with-p4> <https://en.wikipedia.org/wiki/P4_(programming_language)><https://gitlab.cs.technion.ac.il/lccn/w2018-mirror-sampling-mellanox-p4><https://gitlab.cs.technion.ac.il/lccn/s2019-postcard-p4><https://pysimplegui.readthedocs.io/en/latest/><https://matplotlib.org/users/pyplot_tutorial.html>

<https://github.com/p4lang/tutorials>

<https://www.linuxtopia.org/Linux_Firewall_iptables/x4172.html>

<https://mediaonfire.com/blog/2013_11_01_dscp_tagging_with_iptables.html>

<https://gist.github.com/abhi-bit/cafd0edcf107ac2f66b9>  
<https://github.com/p4lang/p4runtime-shell>