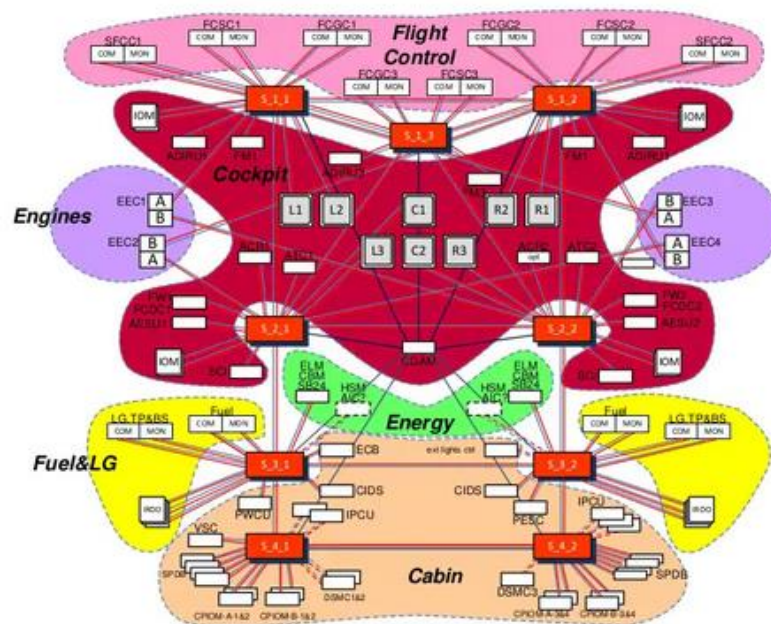


Technical Report

Implementation of a Real-Time Ethernet with Quality-of-Service mechanisms

AFDX network using P4 language



P4FDX

28/04/2022

EMS 2021/2022 - Integrated Team Project

CHAMPAIN Florian

GRENIER Celestin

ILLI Adil

ZEMZEM Mehdi

Table of content

| | |
|-----------------------------------|----|
| 1. Introduction | 4 |
| 2. Presentation | 4 |
| 2.1. Project scope | 4 |
| 2.2. Libraries presentation | 5 |
| 2.2.1. P4 Language..... | 5 |
| 2.2.2. DPDK..... | 5 |
| 2.2.3. T4P4S..... | 6 |
| 2.2.4. P4PI..... | 6 |
| 2.2.5. P4APP & Mininet | 7 |
| 2.3. Hardware presentation | 8 |
| 2.3.1. Raspberry PI3 & PI4 | 8 |
| 2.3.2. PC | 8 |
| 3. AFDX Implementation | 9 |
| 3.1. AFDX P4 switch | 9 |
| 3.1.1. AFDX specifications..... | 9 |
| 3.1.2. P4 implementation | 10 |
| 3.2. Raspberry (P4PI) | 12 |
| 3.2.1. Implementation..... | 12 |
| 3.2.2. Limitations | 16 |
| 3.3. PC (T4P4S) | 17 |
| 3.3.1. Implementation..... | 17 |
| 3.3.2. Limitations | 18 |
| 3.4. Global integration | 19 |
| 3.4.1. Network architecture | 19 |
| 3.4.2. Results and analysis..... | 20 |
| 3.5. Automation tools | 21 |
| 3.5.1. Topology Manager..... | 21 |
| 3.5.2. AFDX Packet generator..... | 22 |
| 3.5.3. Log analyzer | 22 |

| | | |
|--------|---|----|
| 4. | Quality-Of-Service implementation | 23 |
| 4.1. | Strict Priority Queuing | 23 |
| 4.1.1. | Algorithm presentation..... | 23 |
| 4.1.2. | P4 implementation | 23 |
| 4.1.3. | Simulation..... | 24 |
| 4.1.4. | Hardware limitation | 24 |
| 4.2. | Weighted Round Robin | 25 |
| 4.2.1. | Algorithm presentation..... | 25 |
| 4.2.2. | P4 implementation | 25 |
| 4.2.3. | Simulation..... | 27 |
| 4.2.4. | Hardware and Software Limitation and stuff..... | 28 |
| 5. | Conclusion | 29 |
| | References | 30 |

Table of figures

| | |
|--|----|
| Figure 1: P4 language decomposition..... | 5 |
| Figure 2: T4P4S flow chart..... | 6 |
| Figure 3: P4PI Reference Architecture | 7 |
| Figure 4: Simplified structure of AFDX network on the A380 | 9 |
| Figure 5: R. Pi4 32ms | 14 |
| Figure 6: R. Pi4 64ms | 14 |
| Figure 7: R. Pi4 128ms | 14 |
| Figure 8: R. Pi4 256ms | 14 |
| Figure 9: R. Pi3 32ms | 15 |
| Figure 10: R. Pi3 64ms | 15 |
| Figure 11: R. Pi3 128ms | 15 |
| Figure 12: R. Pi3 256ms | 15 |
| Figure 13: DPDK SSE4.2 instruction error | 18 |
| Figure 14: Initial Network Topology | 19 |
| Figure 15: Simplified Network Topology | 19 |
| Figure 16: End-to-end delay histogram curve..... | 20 |
| Figure 17: TopoManager.py tool flow chart..... | 21 |
| Figure 18: Analysis of end-to-end delay of SPQ..... | 24 |
| Figure 19: Ingress, first queue with room for VL2 is q1 (as q2 already has 2 messages) | 27 |
| Figure 20: Egress, does not decrement usage yet (a message is present in q1 below q2) | 27 |
| Figure 21: True WRR..... | 28 |
| Figure 22: Pseudo WRR..... | 28 |
| Figure 23: Delays with weight of 10 for VL 1 and 1 for every other, 1478 bytes packets | 28 |

1. Introduction

In recent decades, Real-time networking has been a major subject of discussion in the embedded systems world. For air borne avionics systems, Real-time communications are a requirement. One of the state-of-the-art standards of real-time networking is ARINC 664, also known as AFDX. It is the prominent form of communications due to Baud rate, latency and overall system weight.

The main aim of the project is the implementation of an AFDX-based Real-time Ethernet network with quality-of-service (QoS) mechanisms; this should be done through a code abstraction layer (i.e., P4 high level language).

In this project we mainly focus on implementing AFDX on real hardware. And, adding the QoS capabilities to the AFDX network which is not currently implemented in the industry. We then validate these capabilities through simulation and compare them against a reference benchmark.

2. Presentation

2.1. Project scope

The objectives initially set early in the project were the following:

- Implementation of P4 on Raspberry (using P4Pi) and on computer (using DPDK)
- Implementation of QoS using WRR and SPQ mechanisms

In a first time, launching a simple switch programmed in P4 in a Mininet simulation is a good introduction to the technology stack used. Another point is to port this P4 program to work on real hardware.

Next, the actual AFDX program can be expressed in P4 and tested in simulation and on real hardware. The same iterations can be done for the various QoS to be implemented.

2.2. Libraries presentation

2.2.1. P4 Language

P4 (Programming Protocol-independent Packet Processors) is a domain-specific programming language for network devices that specifies how data plane devices such as switches, routers, NICs and filters process packets.

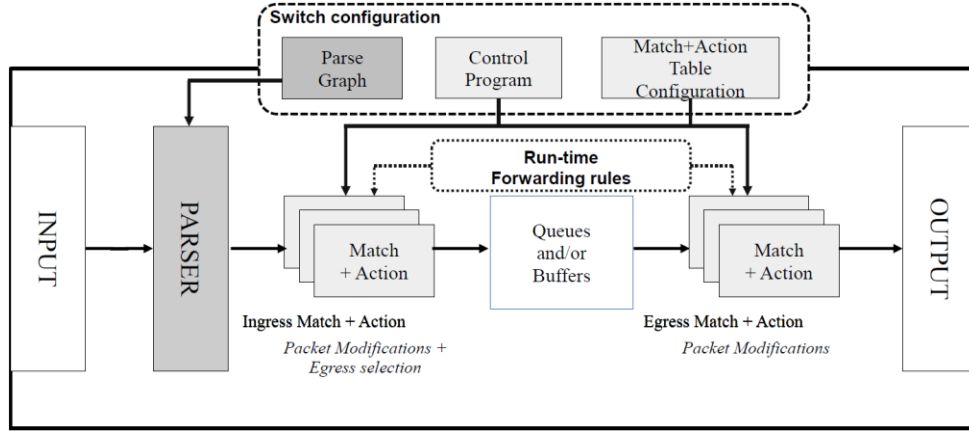


Figure 1: P4 language decomposition

It allows to describe by code the implementation of the parsing, ingress, egress, deparser and checksum computation / verification blocks.

P4 may be used on a wide range of devices, such as programmable network interface cards, FPGAs, software switches, and hardware ASICs. As a result, and being a domain-specific language, it is limited to structures that can be implemented efficiently across all these platforms.

2.2.2. DPDK

DPDK [Ref 9] or Data Plane Development Kit is a C development kit and framework dedicated for the development of high-speed networking applications.

DPDK was first developed by Intel and is now maintained by the Linux Foundation. It provides a set of libraries, an Environment Abstraction Layer (EAL). This allows applications developed in DPDK to bypass the kernel to access directly to the Network Card and other hardware resources. This offloading achieves higher computing efficiency and higher packet throughput than is possible using the interrupt-driven processing provided in the kernel.

In our project, DPDK is used indirectly. It is the backend for the T4P4S tool (will be presented later) and allows us to run P4 code on CPUs (PC or Raspberry Pi).

2.2.3. T4P4S

T4p4s is a multi-target compiler that converts P4 descriptions into high-performance switch programs. It is open source, it supports multiple targets (CPU, NPU, FPGA, etc.) and provides an easily retargetable P4 compiler that separates hardware dependent and independent parts.

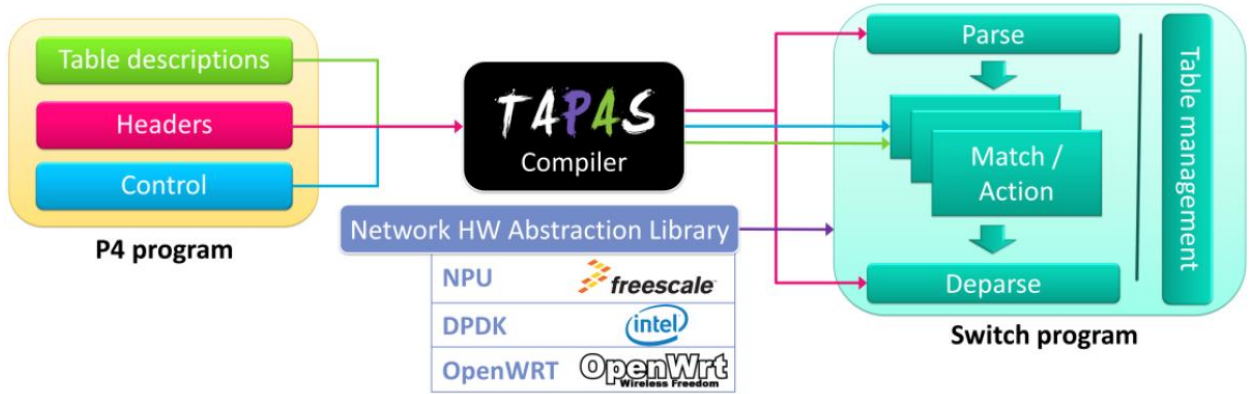


Figure 2: T4P4S flow chart

In this study, T4P4S has been used as an end-to-end tool to compile (using P4C) and launch P4 programs on Linux PC and Raspberry Pi.

2.2.4. P4PI

P4Pi [Ref 8] is a Raspberry Pi OS that allows you to construct and deploy P4 network data planes.

P4Pi is a Raspbian image that integrates all the tools needed to develop and deploy P4 programs using Raspberry Pi. It is developed by the P4 Education Group with the aim of providing a low cost, open-source platform for computer networks teaching and research.

P4Pi allocates two cores to T4P4S to execute the p4 switch and packet processing. The physical interfaces are connected to the T4P4S switch through virtual ethernet devices and bridges. It also provides a management interface through both SSH and WLAN AP.

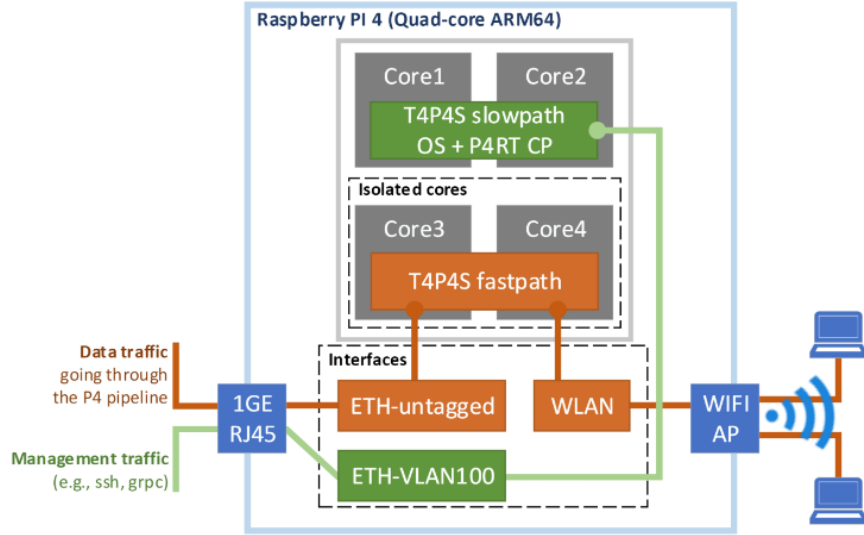


Figure 3: P4PI Reference Architecture

We use P4Pi to have a hardware implementation of the AFDX network on Raspberry Pi.

2.2.5. P4APP & Mininet

p4app is a utility, part of the P4 language ecosystem, which makes it possible to execute, debug, and test P4 programs. p4app leverages Docker to package and launch a Mininet environment. This environment is favorable to executing the result of P4 compilation in simulated switches, linking virtual nodes (or end systems).

p4app “applications” are bundled in folder or compressed archives usually ending with the “.p4app” filename extension. They consist in at least an entry point p4app.json which describes:

- The main P4 program to use (bundled with),
- The language used (e.g., p4-14 or p4-16),
- The network architecture to simulate.

Any other file part of the bundle is made available at runtime to the Mininet environment. Using Python scripts is a way of shipping a behavior for the nodes of the network, or other needed automations. The construction of the Mininet architecture is implemented this way, as lists of Mininet commands ran automatically.

The p4app utility is used to start the Docker environment. Once this is done, another way to interact with the components of the network is to issue commands that the p4app relays through Docker. This is the solution we used in the end to have the nodes send AFDX packets to each other.

2.3. Hardware presentation

2.3.1. Raspberry PI3 & PI4

In our project we used two versions of the Raspberry Pi platform: model 3B and model 4B.

The Raspberry Pi 4 Model B [Ref 10] was released in June 2019 and features a 1.5 GHz quad-core ARM Cortex-A72 processor, on-board 802.11ac Wi-Fi, Bluetooth 5, full gigabit Ethernet (throughput not limited), two USB 2.0 ports, two USB 3.0 ports, 2, 4 or 8 GB of RAM, and dual-monitor support via a pair of micro-HDMI (HDMI Type D) ports for up to 4K resolution. The Pi 4 is powered via a USB-C connector, allowing additional power to be delivered to downstream peripherals.

The Raspberry Pi 3 Model B [Ref 11] is the earliest model of the third-generation Raspberry Pi. It replaced Raspberry Pi 2 Model B in February 2016. It features a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU with 1GB RAM, a BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board and 100 Base Ethernet and 4 USB 2 ports.

2.3.2. PC

Several PC (from Intel Core 2 duo - 1GB Ram to Intel I7 - 8GB Ram) with multiple ethernet cards are available. The initial plan was to use:

- at least 1 PC as a 4 ports P4 switch
- Several PC as end systems to send and monitor packets on the network

But due to hardware / software compatibilities limitations, we were only able to set up a 2 ports switch (see §3.3.2 for more details).

Moreover, a PC with 5 ports has been used to directly emulate 5 independent end systems. This setup offers the advantage of facilitating the packet sending / monitor commands launching as well as making measurements easier due to the usage of a single clock.

3. AFDX Implementation

3.1. AFDX P4 switch

3.1.1. AFDX specifications

Avionics Full-Duplex Switched Ethernet (AFDX), described in the standard ARINC 664, is a data network for safety-critical applications that uses dedicated bandwidth while delivering deterministic quality of service, as patented by worldwide airplane manufacturer Airbus. Airbus has a worldwide trademark registration for AFDX.

The AFDX data network is built on Ethernet and uses COTS (commercial off-the-shelf) components. The AFDX data network is a specific implementation of ARINC Specification 664 Part 7, which outlines how commercial off-the-shelf networking components will be used for future generation Aircraft Data Networks (ADN).

The AFDX data network standard basic specifications are as follows [Ref 1 & Ref 2]:

- Full duplex as a type of access to medium.
- Interconnected topology.
- Baud rate of 100 Mb/s.
- Frame size between 64 and 1518 bytes.
- Static routing through the concept of VL (virtual links) a predefined link between an emitter end system and a set of receiver end systems.
- A respected value of the Bandwidth allocation gap (BAG) between generated packets of the same VL (on a given End System): 2ms, 4ms, 8ms, 16ms, 32ms, 64ms, ...

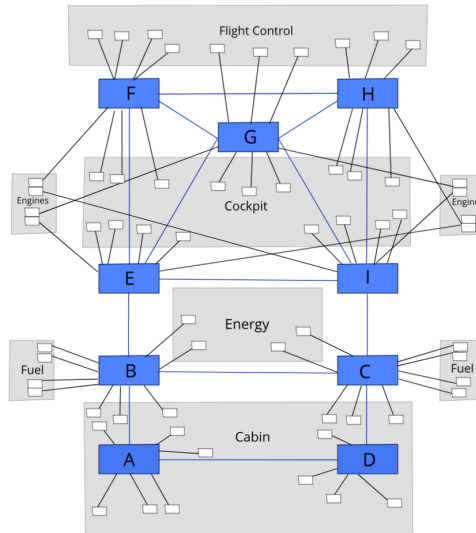


Figure 4: Simplified structure of AFDX network on the A380

In the Figure 4, we can see a simplified version of the AFDX network on the commercial aircraft A380. End systems (depicted as white squares) are the components that are producing and consuming the data, and switches (depicted as blue squares) are the components that are assuming the function of packets switching according to the AFDX standard.

3.1.2. P4 implementation

Under the scope of this project, the AFDX network, specifically the switching function, is to be implemented using the P4 language, previously described in §2.2.1. above. For this purpose, we will use the P4 version of the standard ethernet switch to which we will add the corresponding restrictions of the AFDX standard. This implementation can be divided into the following:

- AFDX Packet constants & header definition:

```
const bit<32> DST_CONST = 0x3000000;
const bit<24> SRC_CONST = 0x20000;

//Headers
typedef bit<24>    MacAddr_t;    // Variable part = 24bits
typedef bit<16>    VLid_t;      // 2Bytes

header afdx_t {
    bit<32>    dstConst;
    VLid_t    dstVL;
    bit<24>    srcMac_cst;
    MacAddr_t  srcMac_addr;
    bit<16>    etherType;
}
```

In this P4 code we defined 2 constants that will be used to check the validity of the received packet as AFDX packets. Also, we introduced the format of a standard AFDX packet header.

- Ingress implementation:

```
control MyIngress(inout headers hdr,inout metadata meta,inout standard_metadata_t
standard_metadata) {
    action Drop() {
        mark_to_drop(standard_metadata);
    }
    action Check_VL(bit<32> MaxLength, bit<16> MCastGrp) {
        meta.maxi_length = MaxLength;
        standard_metadata.mcast_grp = MCastGrp;
    }
}
```

```

table afdx_table {
    key = {
        standard_metadata.ingress_port : exact;
        hdr.afdx.dstVL : exact;
    }

    actions = {
        Check_VL;
        Drop;
        NoAction;
    }
    default_action = Drop();
}
apply{
    if ((hdr.afdx.dstConst == DST_CONST) && (hdr.afdx.srcMac_cst == SRC_CONST))
        afdx_table.apply();
    else
        mark_to_drop(standard_metadata);
}
}

```

In the “*apply*” function of the ingress control, we check the validity of the destination and source mac of the packet against the predefined constant, in case of match we call the *Check_VL* otherwise we drop the packet. The *Check_VL* function is called using the keys defined the *key* structure, which in this case are the ingress port and the VL id.

- Switching table definition:

```

table_add afdx_table Check_VL 1 1 => 1518 3
mc_node_create 3 3
mc_mgrp_create 3
mc_node_associate 3 0

```

This is an example of the switching table for our AFDX switch, in this case it is for VL with id 1 and is received on port 1, if this match takes place the *Check_VL* function will be called on the max length 1518 and cast group 3. The cast group is also defined as the group containing port 3 on which the packets verifying the conditions will be forwarded to.

Using P4APP we compiled this AFDX P4 switch on the target BMV2 (behavioral model v2), then generated some AFDX packets along with some other non AFDX packets. We then noticed that only the AFDX packets pass through the switch and are received exactly to their corresponding destination end system (host).

3.2. Raspberry (P4PI)

3.2.1. Implementation

The goal is to be able to send an AFDX frame from an End System to another ES through P4PI AFDX Switch. This is simply done through recompiling the AFDX program already used for the simulation on the P4Pi platform and configuring the switch correctly. All the used scripts and technical details are found in the p4pi folder of Ref 7.

3.2.1.1. *AFDX in P4PI*

After setting up the P4Pi platform correctly on the Raspberry Pi, we compile, configure and run the already developed AFDX.p4 program:

- 1/ Put AFDX.p4 program in /root/t4p4s/examples folder (default location)
- 2/ Add configuration options for your program in file /root/t4p4s/examples.cfg
- 3/ Run the program

Next, we need to populate the routing tables defined in the p4 program. Two methods are available:

- 1/ Configuring routing tables using the P4 runtime (P4RT) interface.
- 2/ Using static tables defined directly in the p4 source file.

3.2.1.2. *Test plan*

To validate the implementation, we define the following tests:

- 1/ Forward an AFDX traffic between two ports:
- 2/ Forward an AFDX traffic between several ports
- 3/ Timing test for Raspberry PI 4
- 4/ Timing test for Raspberry PI 3

3.2.1.3. Test results

We run the defined test scenarios to detect the potential problems and correct them.

3.2.1.3.1. First Test

After the setup and configuration steps, we ran the first test and were able to detect the following problems:

1/ We are not able to define and configure multicast groups, to solve this issue the AFDX.p4 program used on the Raspberry pi has been modified to use ports instead.

| Before | After |
|---|--|
| <pre> action Check_VL(bit<32> MaxLength, bit<16> MCastGrp) { meta.maxi_length = MaxLength; standard_metadata.mcast_grp = MCastGrp; } </pre> | <pre> action Check_VL(bit<32> MaxLength, bit<9> port) { meta.maxi_length = MaxLength; standard_metadata.egress_port = port; } </pre> |

2/ The Wi-Fi is not stable and is prone to disconnecting often during the tests. Since the Raspberry Pi has only one Ethernet port, the solution to this problem is to add a USB-to-Ethernet dongle converter and to reconfigure the switch to output on the new interface instead of the Wi-Fi. To do so, we must reconfigure the bridge that links the WLAN to the T4P4S switch:

- Disconnect the WLAN from the bridge
- Connect the USB-to-Ethernet to the bridge

3/ Configuring the routing tables through the P4RT shell is not always successful, the entries are sometimes not saved to the table or not recognized as correct. The use of static tables solved this issue.

4/ When adding entries to the routing tables (through both methods), we detected that some correct frames were dropped. This was finally linked to the endianness (MSB/LSB position) (example port 1 = 0x01000 not 0x0001) and solved by inputting the table entries correctly.

After fixing the previously mentioned issues, our AFDX on Raspberry Pi implementation can route traffic between two ports.

3.2.1.3.2. Second Test

With all these problems ironed out, we prepared the device under test for the 2nd test scenario. In this scenario, we will need to route AFDX flows between at least 3 different ports. Thus, we configure the P4Pi switch accordingly:

- 1/ Connect 3 Ethernet ports to the Raspberry Pi: 1 onboard and 2 USB dongles
- 2/ Create and connect bridges and virtual ethernet devices to the USB-to-Ethernet dongles
- 3/ Configure the T4P4S switch to use 3 ports by using specific options in file *opts-dpdk.cfg*

With this new configuration, we ran the test successfully.

3.2.1.3.3. Third Test

The aim of this test is to find out the minimum BAG that the Raspberry Pi model 4B is capable of handling correctly.

We ran this test 4 times with a BAG of 32ms, 64ms, 128ms and 256ms and we found the following results (time is in microseconds):

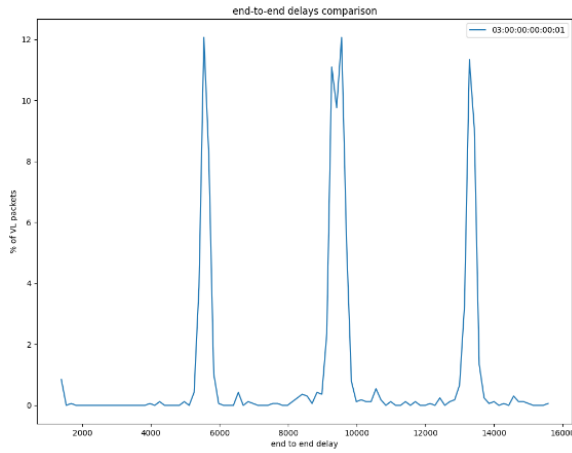


Figure 5: R. Pi4 32ms

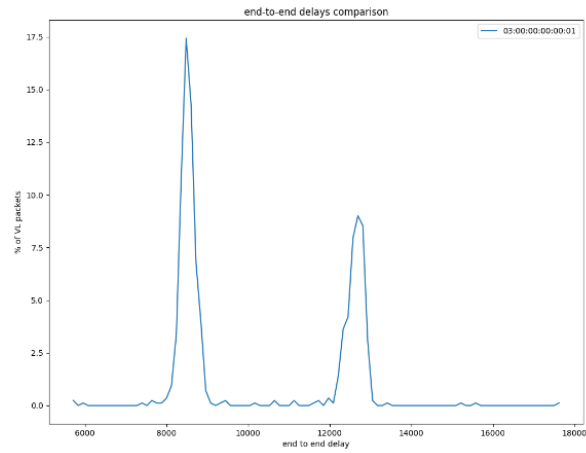


Figure 6: R. Pi4 64ms

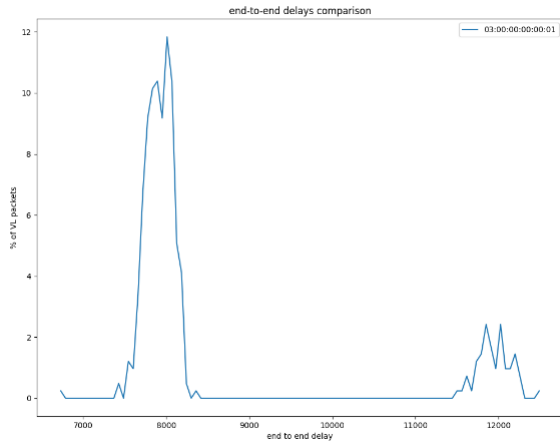


Figure 7: R. Pi4 128ms

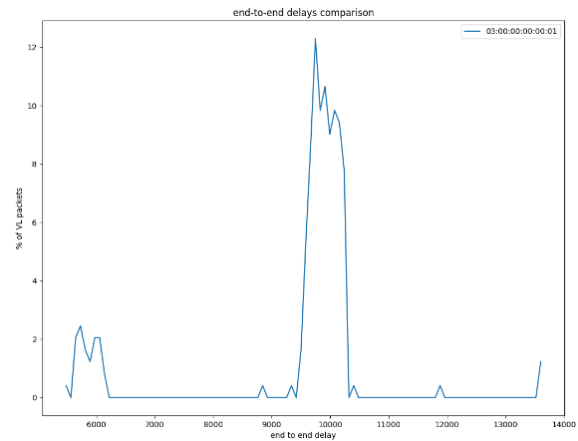


Figure 8: R. Pi4 256ms

These results prove that the Raspberry Pi model 4 can manage all the tested BAG values and we can safely assume that it will be able to handle stricter values such as 16ms or 8ms.

3.2.1.3.4. Fourth Test

The aim of this test is to find out the minimum BAG that the Raspberry Pi model 3B is capable of handling correctly.

We ran this test 4 times with a BAG of 32ms, 64ms, 128ms and 256ms and we found the following results (time is in microseconds):

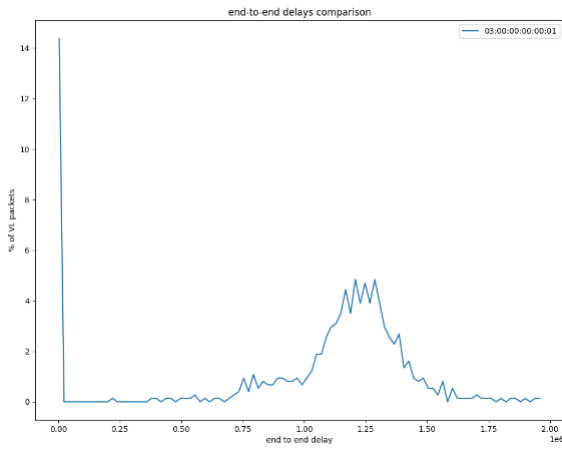


Figure 9: R. Pi3 32ms

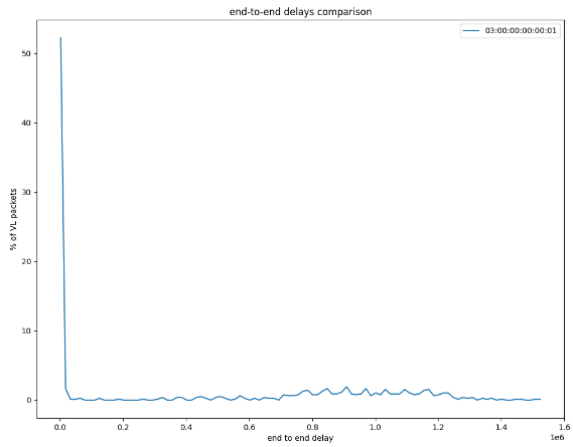


Figure 10: R. Pi3 64ms

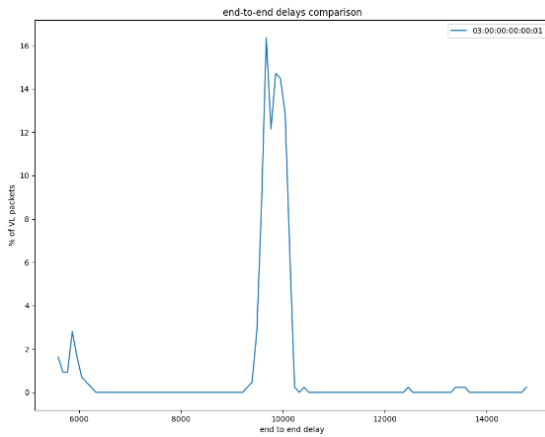


Figure 11: R. Pi3 128ms

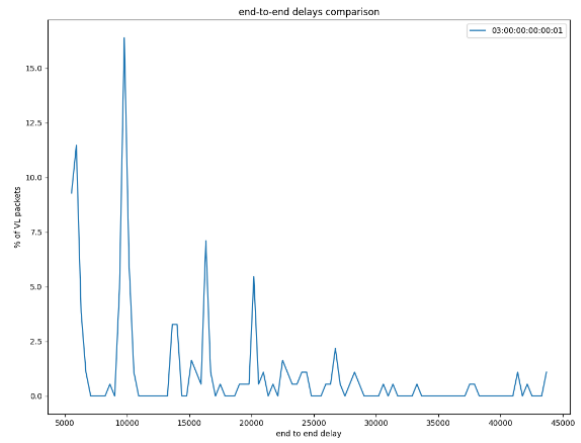


Figure 12: R. Pi3 256ms

These results prove that the Raspberry Pi model 3B cannot handle BAG values lower than 128ms as the tested lower values (64ms and 32ms) failed to reach their destination before the deadline.

3.2.2. Limitations

Through the previously described tests, we can expose the following limitations:

- 1/ It is impossible to have an AFDX flow with a BAG lower than 128ms routed through the Raspberry Pi model 3B due to memory limitations (1GB Ram only shared between the OS and the T4P4S switch, effectively reducing the used memory to 128MB).
- 2/ Unable to configure multicasting groups, thus AFDX on P4PI cannot implement VLs with multiple destinations.

3.3. PC (T4P4S)

3.3.1. Implementation

To avoid compatibility issues with DPDK, P4C and T4P4S, each PC potentially used as a P4 switch has been formatted with the latest LTS Ubuntu version (20.04 LTS).

The T4P4S script directly provided on GitHub compiles and installs all the tools needed for P4.

Once T4P4S is installed on the PC, the next major step is to bind the Ethernet Cards with the *vfio-pci* driver which is used by DPDK.

The *vfio* driver must be priorly loaded. Since no IOMMU was available on our configuration, we needed to refer to DPDK documentation [§5.2.1 of Ref 6]. Moreover, since the first method presented in the documentation, we had to use the alternative method.

Once the *vfio* driver enabled, the default driver can be unbonded from the Ethernet Card and the *vfio* driver bind.

All the operations can be implemented as a script:

```
# Enable vfio driver (alternative method, see §5.2.1 of DPDK documentation)
echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode

# Show the status of each ethernet card driver before binding
python3 dpdk-21.11/usertools/dpdk-devbind.py --status

# Unbind default driver
python3 dpdk-21.11/usertools/dpdk-devbind.py -u 0000:05:00.0
python3 dpdk-21.11/usertools/dpdk-devbind.py -u 0000:08:00.0

# Bind default driver
python3 dpdk-21.11/usertools/dpdk-devbind.py -b vfio-pci 0000:05:00.0
python3 dpdk-21.11/usertools/dpdk-devbind.py -b vfio-pci 0000:08:00.0

# Show the status of each ethernet card driver after binding
python3 dpdk-21.11/usertools/dpdk-devbind.py --status
```

Then, when the ethernet cards are bound for DPDK, the *afdx.p4* file must be in the example directory and the configuration line add to be added in the *examples.cfg* file:

```
afdx arch=dpdk hugepages=1024 model=v1model smem cores=1 ports=1x2
```

Finally, the t4p4s script can be launched with debug and verbose parameters to get the full log in the terminal:

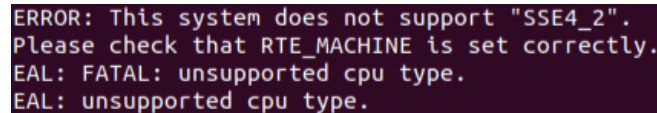
```
./t4p4s.sh :afdx.p4 dbg verbose
```

The switch is now operational and the tests have shown that the AFDX packets are correctly forwarded along VL, but all the other packets are dropped as expected.

3.3.2. Limitations

Some hardware issues have been encountered during the study:

- The PC first used was equipped with Core 2 Duo processors with only 1 GB ram memory with Lubuntu (Light Version of Ubuntu). Unfortunately, 1 GB memory was insufficient to compile DPDK.
- On second try, DPDK and T4P4S were successfully installed on a Core 2 Duo processors with only 4 Go ram memory with classic Ubuntu 20.04 LTS. However, it turned out that DPDK is only compatible with recent processors because of the SSE4.2 instruction (Intel Core i7 (“Nehalem”), Intel Atom (Silvermont core), AMD Bulldozer, AMD Jaguar, and later processors.)



```
ERROR: This system does not support "SSE4_2".
Please check that RTE_MACHINE is set correctly.
EAL: FATAL: unsupported cpu type.
EAL: unsupported cpu type.
```

Figure 13: DPDK SSE4.2 instruction error

- On the third and last try, T4P4S were successfully installed on an Intel I7 processor with 8 GB Ram. The PC was equipped with the motherboard Ethernet port, and 2 double port Ethernet PCI cards, totaling 5 Ethernet ports. Nevertheless, only one double port Ethernet card was recent enough to have a driver supported by DPDK (*Intel egb* driver). The other Ethernet card (*Intel e1000* driver) and the motherboard Ethernet port (*Intel e1000e* driver) are too ancient and are not supported by DPDK.

Unfortunately, obtaining a new compatible Ethernet card before the project's finish was not possible. As a result, with the available hardware, only a two-port switch could be implemented on a PC.

3.4. Global integration

3.4.1. Network architecture

At the beginning of the project, the topology that was originally envisioned was the following:

- one 4-ports PC switch with T4P4S
- one 2-ports Raspberry 3 switch with P4PI (the original ethernet port with 1 additional ethernet/USB dongle)
- one 3-ports Raspberry 4 switch with P4PI (the original ethernet port with 2 additional ethernet/USB dongles)
- one 5-ports PC to emulate 5 end-system hosts

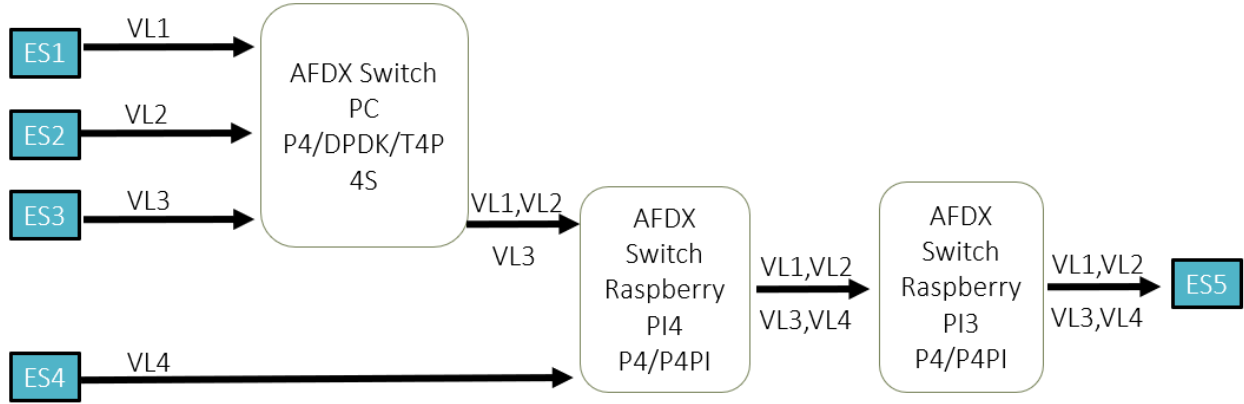


Figure 14: Initial Network Topology

However, since the PC switch has only 2 ethernet ports available (see §3.3.2), the topology had to be simplified with the removal of end-systems 2 and 3 and associated VL (VL2 and VL3).

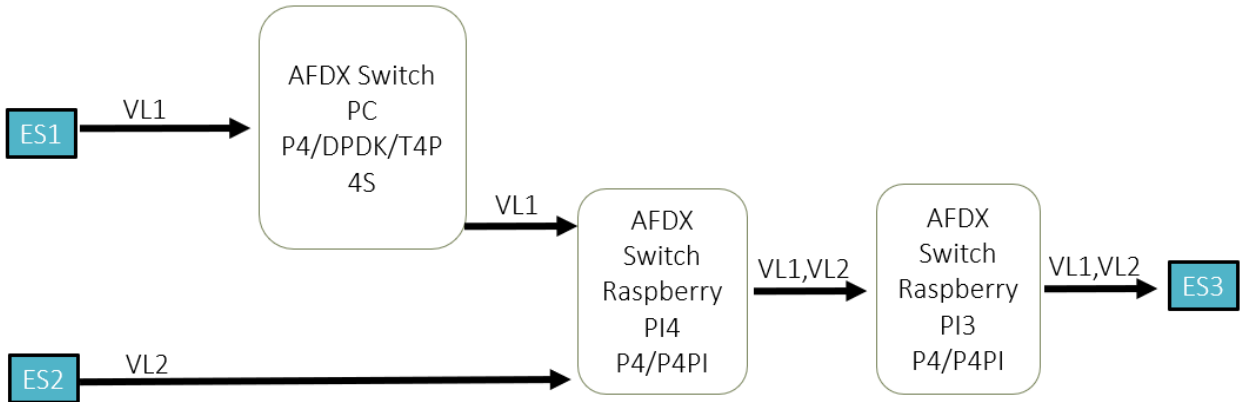


Figure 15: Simplified Network Topology

3.4.2. Results and analysis

This network architecture has been tested with a variety of BAG values and a variety of generated packet counts. Because the results appear to differ between tests when a small number of produced packets is used, the decision was made to use 10000 generated packets in the experiment. We also chose a BAG of 256ms due to the Raspberry PI3's limitations.

With these experiment parameters, we get the following end-to-end delay histogram curve:

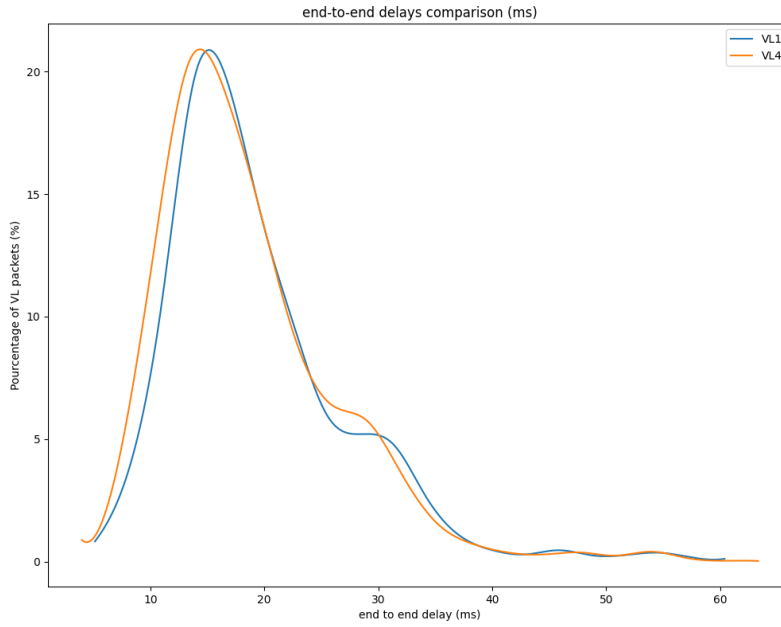


Figure 16: End-to-end delay histogram curve

The two VLs behave identically in the implemented architecture and with the restrictions of the PC as a DPDK-based AFDX switch, with a small additional average delay for VL1. This is because the PC switch operates as a forwarding component, and there is no preference for the treatment of packets of VL1 and VL2 on the Raspberry PI4. As a result, the end-to-end delay histogram has the same pattern.

3.5. Automation tools

3.5.1. Topology Manager

To manage complex network topologies, a tool *TopoManager.py* has been developed. The initial goal of this tool was to provide switch tables containing Virtual Links parameters for each switch. Then the tool has been completed to automatically generate VL check scripts (packet sender and monitor for each VL on each host).

Moreover, the tool has been configured to provide files for the 3 platforms used in this study:

- P4app on Mininet for simulation
- T4P4S for Linux PC with DPDK target
- P4PI for Raspberry 3 & 4 with DPDK target

This tool works with an input text file containing the links between hosts and switches and the complete path of each VL (with BAG and priority values). For PC and Raspberry, a port mapping must be also provided in input to make the correspondence between host and PC / ethernet interface name.

Finally, some basic checks have been performed on the input topology data (consistency between hosts/switches and VL paths, BAG value being a power of 2ms, host/port mapping).

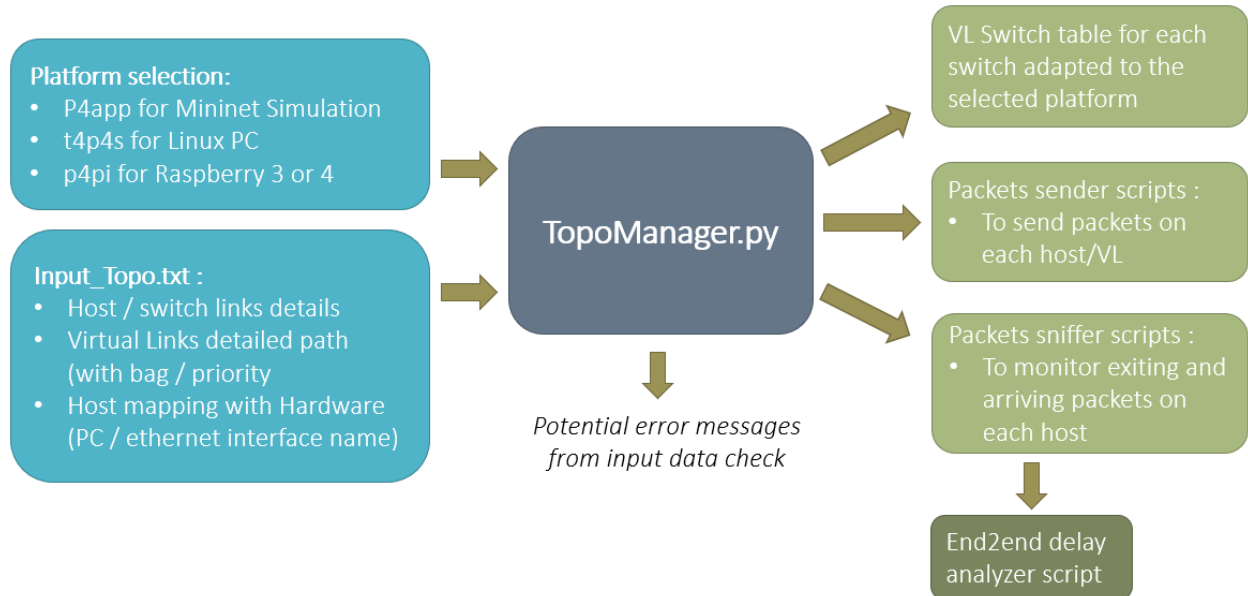


Figure 17: *TopoManager.py* tool flow chart

3.5.2. AFDX Packet generator

The AFDX packet generator is a tool developed in python, which simulates the behavior of end systems. It generates, in a periodic manner, a given number of successive packets for given VLs. And send the generated packets on a desired network interface according to the given BAG values. This tool uses the python library *scapy*, that gives us the ability to interact with native Linux tools such as *tcpdump*, *arp spoof*, *Nmap*, *tshark*...

Using this tool, we can generate in practice on a Linux station with Core 2 duo CPU and 1GB of RAM, up to 8 VLs each with 32ms BAG value. For higher values of BAG (64ms, 128ms, ...) we can generate more VLs that are respecting the emission schedule. But, for the purpose of this project, and with the limitations detected on the Raspberry PI3 board, these values would greatly suffice.

3.5.3. Log analyzer

Once launching a simulation on *Mininet* or a real experiment on the real implemented architecture, we save the transmitted and received packets of each host on *.pcap* files using *tcpdump* with the help of script created by the topology manager previously described. Then using a python script, we loop through all packets from emission and reception stations of each VL, and then we calculate the histogram of the end-to-end delays and plot the histogram for each VL in a graph. An example of the resulting figures is given in section 3.1.2.

All scripts, tools and example files used and developed for this project are available on GitHub [Ref 7].

4. Quality-Of-Service implementation

4.1. Strict Priority Queuing

4.1.1. Algorithm presentation

Strict priority queueing (SPQ) divides network packets into multiple categories: high priority, regular, and low priority, ensuring that priority traffic is always handled first. Packets are sorted into different FIFO queues based on their priority; a particular queue must be entirely empty before a lower priority queue is supplied. This approach has the benefit of ensuring delivery of high priority packets if their input does not exceed the network's transmission rate. The acknowledged potential downside is that a high amount of priority traffic will degrade normal traffic significantly [Ref 3].

4.1.2. P4 implementation

By default, the behavioral model “simple switch” which is a precompiled p4 target does not support multi-queueing and needs to be recompiled with some modifications that will activate this characteristic. For this purpose, we used the p4 provided Development Ubuntu virtual machine. And, then included queues in the *simple_switch* target and recompiled. After that, we activated two needed fields for this implementation on the *v1model*:

```
//Priority queueing
@alias("queueing_metadata.qid")          bit<5> qid;
@alias("intrinsic_metadata.priority")     bit<3> priority;
```

For our implementation we used 8 queues which is the default value for the number of queues in p4, with queue 0 being the highest priority and 7 being the lowest priority. We then affected to each VL a certain priority (integer between 0 and 7) in the switching table, which we then extract and save in the Ingress using the *check_VL* action (defined in §3.1.2), and then affected the value to metadata fields *qid* and *priority*:

```
action Check_VL(bit<32> MaxLength, bit<16> MCastGrp, bit<3> priority) {
    meta.maxi_length = MaxLength;
    standard_metadata.mcast_grp = MCastGrp;
    meta.priority = priority;
}

    ....
    ....
```



```

apply{
    if ((hdr.afdx.dstConst == DST_CONST) && (hdr.afdx.srcMac_cst == SRC_CONST)) {
        afdx_table.apply();
        standard_metadata.priority = meta.priority;
    }
    else
        mark_to_drop(standard_metadata);
}

```

4.1.3. Simulation

To simulate the SPQ, we implemented using *Mininet* a switch linked to 7 hosts, where 6 of them are emitting a single VL with a BAG of 64ms and a packet size of 64 bytes. VL1 has priority 0 (highest) while VL2 to VL6 all have priority 1 (second highest). And the switch forwards all the VLs to port 7, which is linked to port 7, according to the static priority queuing mechanism.

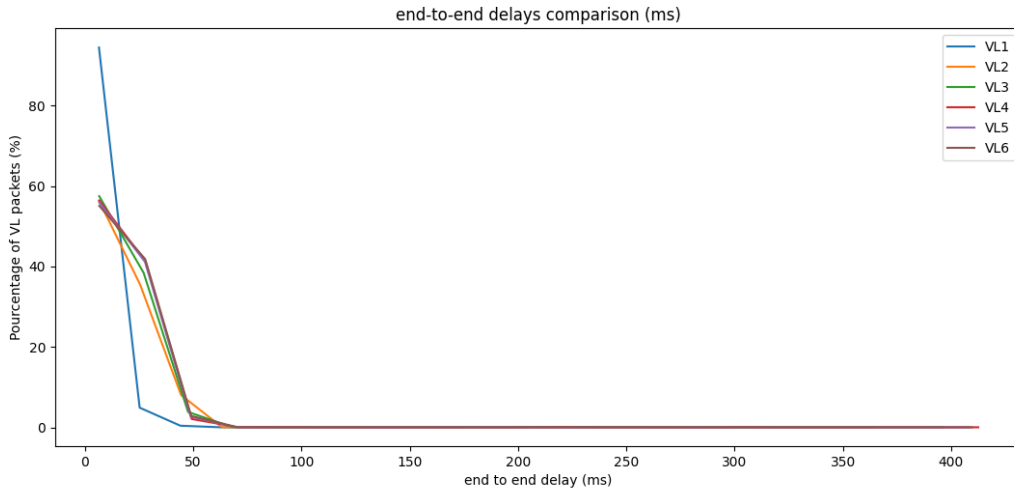


Figure 18: Analysis of end-to-end delay of SPQ

4.1.4. Hardware limitation

P4 queue management currently is only limited to BMV2 (behavioral model v2), which is a software only solution. And it is not readily available in the tools used to run P4 programs on hardware (T4P4S for PC and Raspberry Pi) [Ref 5]. Some workarounds can be found but no direct availability of queues management is present on any P4 hardware implementable target. For example, the P4C compiler could be altered to provide queue-capable back-ends that could be executed on hardware (PC or Raspberry).

4.2. Weighted Round Robin

4.2.1. Algorithm presentation

Weighted Round Robin (WRR) queueing is a round robin scheduling algorithm that approximates Generalized Processor Sharing (GPS) in a computationally efficient way.

In the GPS algorithm, packets are classified from each flow into different logical queues. GPS serves nonempty queues in turn and skips empty queues. It sends a set amount of data from each queue, so that in any finite time interval it visits all the logical queues at least once.

With the WRR approximation, every round each nonempty queue transmits up to a number of packets proportional to its set weight. If all packets are of uniform size, each class of traffic is provided a fraction of bandwidth linked to its assigned weight. When this property is not verified, as it is the case with ethernet packets, a good GPS approximation is harder in practice harder to achieve [Ref 4].

Deficit round robin is a later variation of WRR that achieves better GPS approximation without knowing the mean packet size of each connection in advance. It was studied but not implemented due to language limitations and time constraints.

4.2.2. P4 implementation

The implementation uses the queuing mechanism highlighted in paragraph §4.1.2. By design, the P4 language is limited in what operation can be performed on these queues and their usage. It is not possible to specify from which queue the next message should be sent.

Implementing a WRR scheduling mechanism as per definition is therefore not possible. We implemented an approximation of the WRR scheduling mechanism by abstracting over the 8 configured priority queues. The algorithm used is closely related to the SP-PIFO which implements Push-In First-Out queues with the same technical constraints.

It uses a table of integers to track the “usage” of each VL in each queue. If the usage, for a given VL, of a given priority queue becomes greater than the weight associated to the VL, it is forced to use a lesser priority queue.

```
// at ingress (pseudo-code)
apply {
    found_room = false;
    // test each queue in order of priority for usage,
    for (q = 7; -1 < q; q--) {
```

```

        // use first one with free space
        if (usage[vl_id][q] < vl_weight) {
            usage[vl_id][q] += 1;
            standard_metadata.priority = q;
            found_room = true;
        }
    }
    // if no room was found, defaults to the last queue
    // this is done so the order is preserved
    if (!found_room) {
        usage[vl_id][0] += 1;
        standard_metadata.priority = 0;
    }
}

// at egress (pseudo-code)
apply {
    // "below" priority-wise
    below = standard_metadata.priority - 1;
    // if the next queue "below" is not used,
    if (7 < below || 0 == usage[vl_id][below]) {
        // only then can the usage be decremented for the current queue
        usage[vl_id][standard_metadata.priority] -= 1;
        // if it reaches 0, reset usage everywhere
        if (0 == usage[vl_id][standard_metadata.priority]) {
            for (q = 7; -1 < q; q++) {
                usage[vl_id][q] = 0;
            }
        }
    }
}
}

```

At egress, the message being sent holds metadata indicating from which queue and for which VL it is. With this message sent, the usage of this “current queue” by this VL may be decreased under certain conditions. If the next queue “below” priority-wise still has unsent packets, these must be sent before any new packet is even added to the “current queue”. This otherwise would disrupt the order in which packets are transmitted. The solution to that is to decrease the usage for the “current queue” only if the next queue “below” is not used at all (notice the edge case when the “current queue” is the least priority queue). Therefore, if the usage on the “current queue” drops to 0, it is sure that there is no pending packet for the VL in question, and the usage can be fully reset.

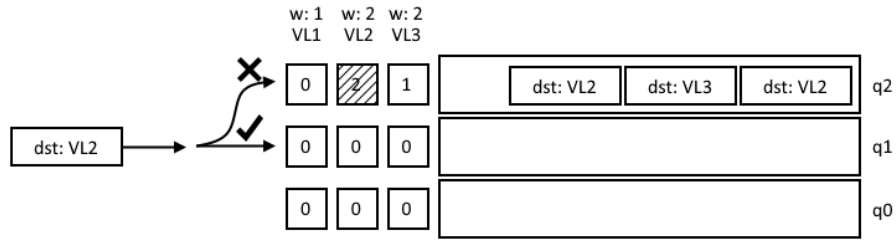


Figure 19: Ingress, first queue with room for VL2 is q1 (as q2 already has 2 messages)

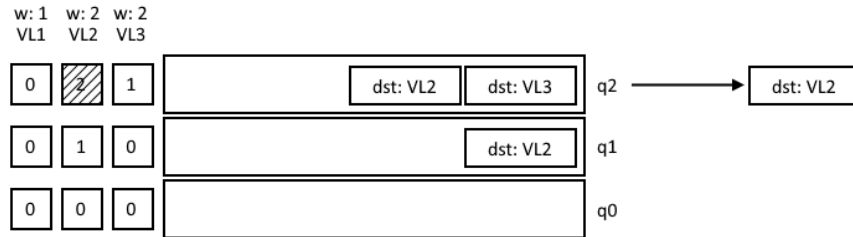


Figure 20: Egress, does not decrement usage yet (a message is present in q1 below q2)

4.2.3. Simulation

The first proof of concept was implemented in Python to compare against a proper WRR implementation. This simulation sends 1000 packets through a switch, over 3 different VLs, by bursts of 10 to 20 while letting time for only 14 to 26 packets go through between bursts. The weights are 3, 5 and 7 for each VL respectively.

The graphs below show the average number of packets for each VL in a sliding window over the resulting stream of packets as output by the switch.

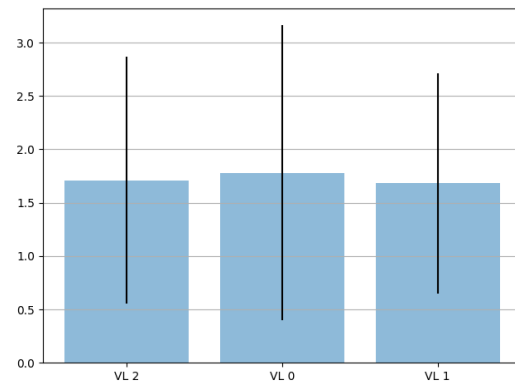
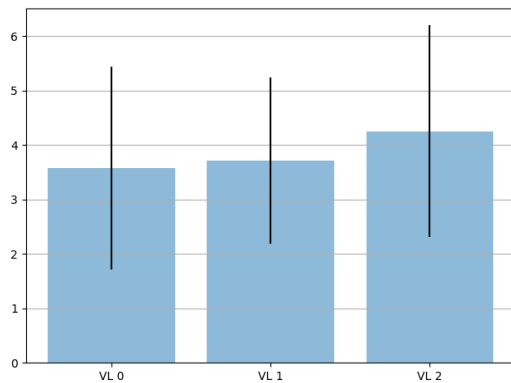


Figure 21: True WRR

Figure 22: Pseudo WRR

In the case of the true implementation of the WRR, we can see that multiple packets from the same VL tend to be more grouped, and VL 2 appears as having a higher average. This is what is expected given the weight set. On the other hand, the result of the approximation algorithm does not show this same trait.

Despite non-promising results, a Mininet simulation was performed to compare with SPQ.

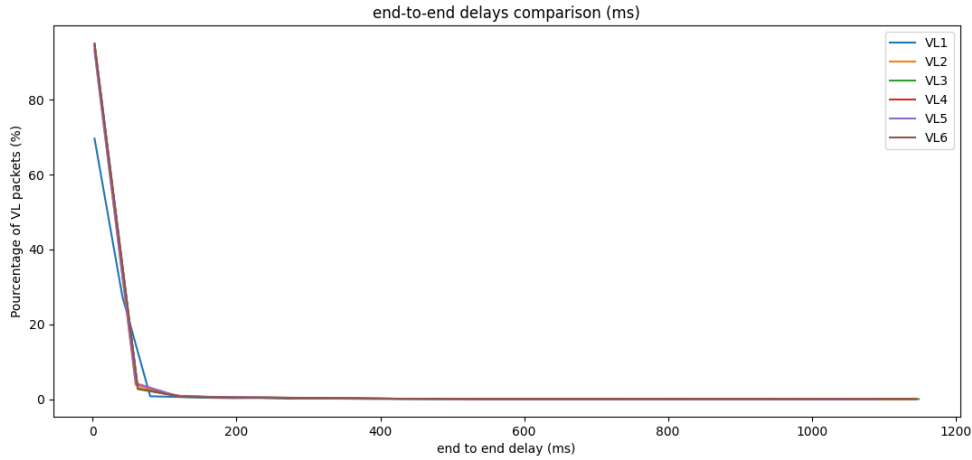


Figure 23: Delays with weight of 10 for VL 1 and 1 for every other, 1478 bytes packets

As shown in Figure 23, the resulting end-to-end delay for VL 1 stands out from the rest. However, the result looks counter-intuitive as the other VL seems to get a better end-to-end delay despite having a lower weight in the WRR. It was not clearly established why due to a lack of time and the difficulty to properly debug P4 programs.

4.2.4. Hardware and Software Limitation and stuff

This scheduling suffers from the same limitations as the SPQ. More queues would theoretically make for a more accurate approximation of the WRR.

Furthermore, this algorithm necessitates a table of register which size is the number of VL by the number of priority queue (it is implemented here as a list which size is the product of the two). It thus depends in size on the number of VL going through the switch. Because this number must be known at compile time, changing the topology of the network becomes less practical. In the case of multi-threaded switches, this structure needs to be accessed atomically (with `@atomic` in P4). This may have a negative drawback on specific hardware.

5. Conclusion

We were able to implement AFDX utilizing P4 and validate its behavior in simulation and on real hardware (PC and Raspberry Pi), as shown in this study. We were also able to use P4 to construct a static priority queuing mechanism for AFDX and simulate it against a reference benchmark with no QoS.

Nevertheless, this project highlighted serious P4 limitations related to the Quality of Service. It appears that it is probably not the ideal solution for the abstraction of switch queues management so far. However, future evolution of the language could have more QoS capabilities.

Furthermore, several leads can be pursued as a possible path forward. The initial step would be to implement QoS on hardware targets by altering the T4P4S backends. Then, further research into round robin algorithmic approximation is required in order to give a functional solution or determine the lack of one.

References

- Ref 1: Anaïs Finzi and Silviu S. Craciunas, “Breaking vs. solving: analysis and routing of real-time networks with cyclic dependencies using network calculus”, November 2019.
- Ref 2: Jean-Luc SharbarG, "AFDX (ARINC 664) Course materiel", ENSEEIHT, Janvier 2022.
- Ref 3: Robert Chang and Vahab Pournaghshband, “Simulating Strict Priority Queueing, Weighted Round Robin, and Weighted Fair Queueing with NS-3”, California State University, Northridge, 2017.
- Ref 4: Katevenis, M.; Sidgiropoulos, S.; Courcoubetis, C. (1991). "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip". *IEEE Journal on Selected Areas in Communications*. 9 (8): 1265–1279
- Ref 5: Sandor laki, Radostin stoyanov, David Kis, Robert Soulé, Peter Voros and Noa Zilberman, "P4Pi: P4 on Raspberry Pi for Networking education"
- Ref 6: https://doc.dpdk.org/guides/linux_gsg/linux_drivers.html#vfio-no-iommu-mode
- Ref 7: https://github.com/FlorianC31/AFDX_P4_QoS/
- Ref 8: "P4Pi: P4 on Raspberry Pi for Networking Education". Sándor Laki, Radostin Stoyanov, Dávid Kis, Robert Soulé, Péter Vörös and Noa Zilberman. *ACM SIGCOMM Computer Communication Review*, Volume 51, Number 3, July 2021
- Ref 9: <https://www.dpdk.org/>
- Ref 10: <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/>
- Ref 11: <https://www.raspberrypi.com/products/raspberry-pi-3-model-b/>