An Emulation Model For Embedded Networks used in Avionics

Ramón Gómez Moya
Bid Payload and Telecom Satellite System
Thales Alenia Space
Toulouse, France
ramon.gomez@umh.es

Pablo Corral
Communications Engineering Department
Universidad Miguel Hernandez
Elche, Spain
pcorral@umh.es

Saulo M. Froes
Mechatronics Post Graduation Program
Federal University of Bahia
Salvador, Brazil
saulo.froes@ibge.gov.br

Antonio Cezar de Castro Lima
Electrical Engineering Department
Federal University of Bahia
Salvador, Brazil
acdcl@ufba.br

Israel Eduardo de Barros Filho
Postgraduate Program in Electrical and Computer Engineering, Federal University of Rio Grande do Norte,
Natal, Brazil
israelfilho_1@hotmail.com

Guillermo De Scals
Applied Physics Department
Universidad Miguel Hernandez
Elche, Spain
gdescals@umh.es

Abstract— ***The objective of this document is to show how to create an emulation of an AFDX network with commercial switches. AFDX, is a standardized deterministic network which can be found inside big size aircrafts such as the Airbus A380 and the A350. This network is formed by End Systems and AFDX switches. The End systems are devices that send or receive data into or from the network. On the other hand, the AFDX switches are the devices which are responsible of the commutation of all the traffic. Throughout this paper we are going to map AFDX network features, using Cisco switches as AFDX switches and Linux computers as End Systems.***

Keywords—AFDX, ARINC664, VLAN, CSMA/CD, QoS.

I. INTRODUCTION

The ARINC standard (Aeronautical Radio, Inc) 664 specification consists of the definition of how data networks within airplanes should work and how they are defined [1]. This consists of 7 parts: first (global concepts and overview), second (link layer specification and physical Ethernet), third (protocols and internet-based services), fourth (address structure and numbers based on internet), fifth (characteristics of network domain and functional elements), sixth (network management specification), seventh (deterministic networks) and octave (services of higher layers). In this paper we will work in depth with part 7 of this standard, the deterministic AFDX (Avionics Full-Duplex Ethernet Switching) networks [2].

One of the predecessors of ARINC 664 part 7 was the ARINC 429 formulated in the late 1970s which can still be found in some active and retired aircraft series [3]. This was one of the first standards ever made in avionics. This standard had as basic unit; the word, and two different coexisted in these networks: data words and message control words. As we can guess from the control words, Link control packets existed and they were responsible of informing, confirming… messages "data received OK", "data received not OK", "synchronization lost". that means that bidirectional information exchange was required. However, ARINC 429 defined a unidirectional and simplex bus so a station could be attached to multiple buses and operate as either sender or recipient. Because of this, a severe challenge was assumed when interlinking so even when having few stations, the configuration could present important complications and also this affected the weight of the aircraft [4].

The objective of this paper is to emulate, through commercial switches, the operation of this type of network in such a way that we obtain results of network parameters similar to those obtained in AFDX networks. That way, we will have a much simpler network to build, because we do not need to change AFDX, which is not very common, but we will also have a low-cost network. This not only implies reduction of costs for companies, it also implies more facility for research laboratories when developing technology and even possibility of accessing these devices to new students in universities, future professionals in the sector.

This paper is organized as follows: Section II describes an overview of previous studies. Section III shows an emulation of AFDX networks. Section IV summarizes how is the traffic generation and in Section V analyses the results. Finally, Section VI exposes the conclusions.

II. PREVIOUS STUDIES

AFDX networks are called deterministic because they are static networks, at all times we know what the origin and destination of the packages are. For example: wing sensors of the A380 airplane sending data to a central control system. In this case always the sender will be the sensor and the receiver the central system. For this reason, the whole network is configured in a static way, so, in every moment we know where the data goes. The AFDX networks have two types of elements: End Systems (ES) and AFDX Switches that would correspond in a LAN with the computers and with the switches respectively as we can see in Fig. 1. Obviously being data networks embedded in aircraft, it is necessary to ensure low delays, high availability in the devices, static configurations of the MAC tables, no configuration protocols such as ARP, STP or any type of proprietary protocols.

Another important notion in AFDX networks is the meaning of Virtual Link (VL) which is basically the connection between an ES and one or several ES through a shared physical medium (the same cable can be used by one or several ES, obviously, sharing the bandwidth). This presents a wide improvement over the standard used in the past, ARINC 429, which required using a physical cable for each connection [5].

On the other hand, in the same way that VLs are defined, the generation of traffic in them is not at all random. Each VL defines a Band Allocation Gap (BAG) that takes values between 1 and 128 ms, always being a power of 2. This parameter is the one that indicates the minimum time between two frames for which we can transmit.

As we can well imagine, transmitting data as quickly as possible (minimizing the delays of switching, propagation, ...) is crucial in AFDX networks. For this reason, the ES and AFDX Switches have strong technological restrictions in order to meet these requirements, which also means that their price is quite high.

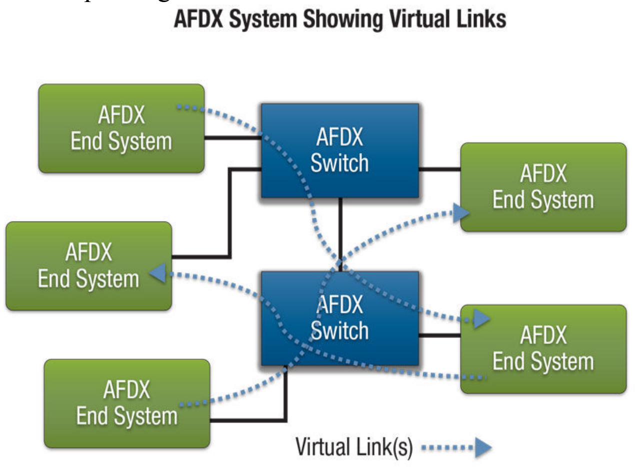


Fig. 1. AFDX Virtual Link Example.

It is true that the definition of VL in LAN type networks does not exist. However, we have the notion of VLAN (Virtual Local Area Network) that is quite close. This means that the advantage of VLs is that although they share a physical environment for practical purposes, it is as if each origin were really connected to their destinations independently.

We can achieve this same effect using the VLAN concept in a LAN. Specifically, the IEEE 802.1Q protocol that adds 4 bytes to the ethernet frame and that basically allows us to: tag which VLAN the pack belongs to (12 bits that allow us to define from VLAN 1 to VLAN 4095) and priority (3 bits, takes values from 0 to 7) also known as IEEE 802.1p Class of Service (CoS). In this way, defining the link as a trunk (allows all VLANs or VLANs that we indicate, since it supports access lists also in case, we need to use them) can have the same effect as we have with VLs [6].

In the AFDX networks the same ES may be the same VL, however this is not so usual in a LAN, because normally a computer is within a single VLAN, unless it has several network cards. However, thanks to the Linux tools it is possible to create virtual interfaces on the network card in such a way that each one belongs to a different VLAN. For this reason, the static configuration of the MAC tables can lead us to doubt this, because when we define an entry in the table it is necessary to indicate: VLAN, MAC address and physical port through which that VLAN and that MAC is accessed. How can we then define that the same MAC is in 2 different VLANs? The problem seems apparently significant. MAC tables allow us to repeat the entry as long as the VLAN changes. Therefore, if we had an ES in several VLANs this is what we would see:

TABLE I: MAC static configuration example

VLAN	MAC Address	Type	Ports
VLAN 1	MAC_ES	Static	Fa 0/X
VLAN 2	MAC_ES	Static	Fa 0/X

If we want to do a multicast to the network what we are really doing is a broadcast within the same VLAN. This means that by going to the destination MAC address FF: FF: FF: FF (broadcast) we will be multicasting the network. To demonstrate this theory, we proceeded to use the software "Packet Tracer" from Cisco Systems as we can see in Fig.2.

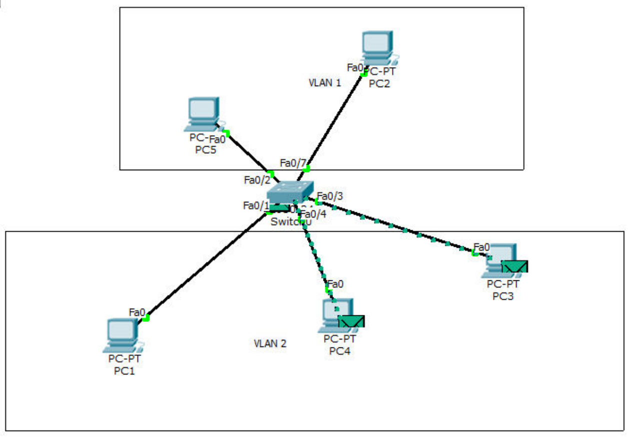


Fig. 2. Broadcast in VLAN2.

III. Emulation Of AFDX Networks

Although we have already checked for proper operation with 'Packet Tracer' software, sending 'multicast' brings a more realistic example of AFDX networks where there was at least one link that was shared by more than one VLAN and definitely check this section.

For this we first use the GNS3 software as we can see in Fig. 3 because 'Packet Tracer' is a valid software when simulating the configuration of the switches, routers, ... however it does not allow us to configure the ES so we could not introduce the same equipment in different VLANs. In principle it was believed that instead it could be with GNS3 but as in Packet Tracer no configuration is possible. However, two fundamental advantages presented GNS3 with respect to Packet Tracer. The first, GNS3 allows us to introduce the images of the Cisco routers in the software (routers 3600, 3700, ...) while in Packet Tracer the models and features are fixed, thus limiting the configuration and preventing the implementation of certain features that we needed. The second, GNS3 gives us the option of capturing the traffic in the different interfaces of the computers through Wireshark, which makes it perfect to know what kind of traffic is going through a certain point and what origin and destination it has, as well as the VLAN to the one that belongs, encapsulation, ...

As an additional point, it should be noted that the use of OMNET ++ was also raised. But, despite the power of the software that allows us to configure a switch, a link (ethernet, fast ethernet, fiber optic, ...), a router, ... with the characteristics that we want (type of service policy, queues, speed of switching, delays, ...) does not meet the requirements of what we are looking for. The reason is that the realization of an AFDX network was extremely complex since it is necessary to implement the entire implementation of switches, ES, physical medium, which could perfectly result in an entire project just for that. It is for this reason that basically the idea of OMNET ++ was discarded.

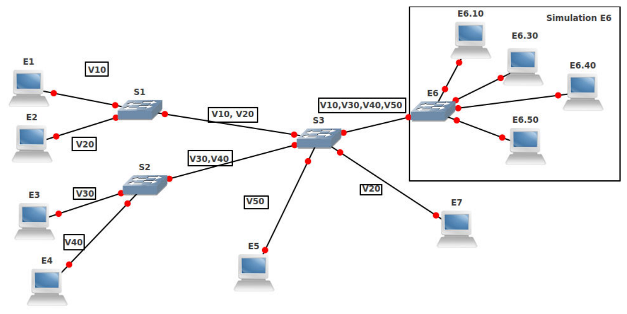


Fig. 3. AFDX network simulation on GNS3.

Once tested with GNS3 we used real Cisco Catalyst 3750 switches to see if everything happened as it was in the software. The only change that occurred was to simplify the design of the network a bit so that assembly and configuration would not be that long.

As expected in Fig. 4, the implementation with real switches and the one simulated in GNS3 finished corroborating that the configuration of the network was possible and that it worked correctly.

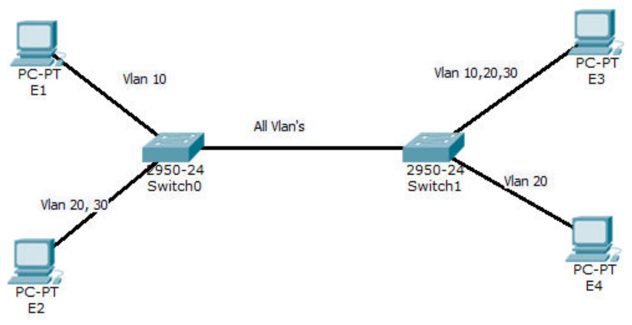


Fig. 4. Simulation on real equipment.

IV. Traffic Generation

Once we have seen that the configurations and the general idea of the functioning of the AFDX networks is feasible from commercial switches, we are going to emulate an ES model in our Linux computers. For this, as we have explained, we have the parameters BAG and Smax that are initially generated and controlled by the ES. Therefore, first of all we need to know how to create the Ethernet frames and then the traffic policy that we are going to install. That is, the configuration of the BAG and the Smax.

This second option allows, after choosing the source and the time range in which the observations were done, gets an average of transits that meet these requirements, allowing obtains a measure of source's data more reliable.

In this work we are not working above the link layer, that is, we never get to work in the network layer, or what is the same at the IP level. This means that we do not send packets to a destination IP but to a MAC. A priori, this is not possible in a network because the packets require carrying all the encapsulated layers and we cannot send information without going through the network layer first. However, what we are going to do is open a RAW socket that, in summary, allows us to send ethernet frames without having previously defined data in the application, presentation, session, network layer ... for this we will make a C program to use this tool.

Our C program is only developed for Linux machines and asks for five parameters for the input: interface that we will use to send traffic, BAG, Smax, VLAN number and VLAN priority.

Once the data has been entered, we will create the Ethernet 802.1q header from them and open a socket. We will introduce as data of the ethernet frame (between 46 and 1500 bytes, it will depend as soon as we have defined the Smax) all the bytes with the hexadecimal value 0x01, this value is simply used as a fill of the plot since what interests us is to send the frames , what goes inside is not indifferent. However, for reasons that we will see later we will include in the final 15 bytes the time with precision up to microseconds. That is, we will have the data field all to 0x01 except the last 15 bytes. Once our frame has been created it will be sent by the socket with destination MAC address FF: FF: FF: FF: FF (broadcast) as already explained and the BAG time will be expected until sending another frame that the only thing that will vary will be sent in its last 15 bytes.

We will be sending Smax-sized ethernet frames and each BAG. Theoretically this is expected, however the reality is a little different, because the program needs a process time and therefore in time between frames can never be the value of the BAG but a little higher. This does not present any problem because according to the standard the minimum time has to be that of BAG but it does not specify maximum times so our program is perfectly valid.

On the other hand, this is only executable in an interface, that is to simulate the operation of a computer with several VLANs we will need to do the same as we did with GNS3. In the same way you could think of running the same program at the same time in different Linux command prompts, however, the ES follow a concrete traffic policy storing the information at the exit and we do not have the specification of what is The policy of sending our network card and therefore we cannot ensure that the operation is the same as in an ES AFDX.

V. Results

So far we have managed to send the Ethernet frames through the RAW Socket but we have certain problems when it comes to measuring the transmission times. The objective of creating the ethernet frames was to be able to measure in some way the transmission time in our emulation in the networks, remember that we are only working up to link level and that our traffic is also unidirectional so the package is sent but not no type of ACK or response pattern is received.

Therefore, to measure the transmission time in our network we need some kind of software that allows us to measure the time of transmission of traffic. This is not trivial in any case because, for example, when we send a ping, we know the time it takes to get to the destination and return but not the time only one way or return. The most logical and quick to think about to solve this problem is to know the time at which it arrives at the destination and subtract the departure time from the origin. It is an option, however, in no case feasible because the clocks are never synchronized at the level of seconds much less in microseconds that is in the order that we want to measure. We find here the first obstacle to obtain measurements.

As we can remember from the explanation about the program in C we had introduced the time in the last 15 bytes of the plot precisely to try to solve this problem.

What we will do is send a ping directly between the two computers we want to measure and know what is the transmission time of that ping (round trip). With this we will know the time difference between the clocks that must be constant in theory will be (assuming that the ping takes the same time to go and return) [7]:

$$Cons = Hdest - \frac{Ping - Tprocess}{2} - \text{Hsour} \quad (1)$$

Where,

Cons: difference between watches,

Hdest: departure time of the destination,

Ping: ping transmission time,

Tprocess: time it takes the destination to receive the ping request and respond with the ping reply,

Hsour: departure time of the origin.

And therefore our outbound transmission time will finally be:

$$Ttx = Hdest - Cons - Hsour \quad (2)$$

Where Ttx, is the transmission time.

Once we know the constant between the clocks we carry out the different transmissions with our program in C and capture the packages in reception with Wireshark exporting them as .xml files [8].

On the other hand, we have generated a program in Python that reads each packet saved in the .xml file, taking the time sent from each packet (departure time) and the arrival time to destination and then applying the formula described in eq. 2. In this way we already have the transmission time of all the Ethernet packets sent.

So, we have divided the results in different topics:

A. *Real example of AFDX network*

To carry out the measurements and results we have created our own example shown in Figure 5 that represents a real example of the AFDX network (very similar to the one we have shown in the GNS3 simulation). In this example all the data is created by E1, E2, E3, E4 or E5 towards the corresponding ESs within its same VLAN. So E1 will transmit to E6.10 and E8, E2 to E7, E3 to E6.30, ...

The objective of constructing 4 ES's called E6.10, E6.30, E6.40 and E6.50 was due to the impossibility, as we have done in the previous GNS3 example, to build a single E6 and that it fulfilled the requirements of the standard of the ES in the AFDX networks when multiplexing the data at the output. In this way we ensure that they are met although in a real example it is impractical.

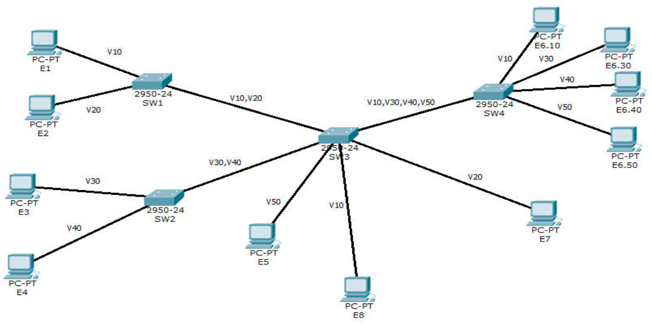


Fig. 5. AFDX example.

This simulation was performed to verify that the software developed and the network configurations on the switches worked correctly. On the other hand, an attempt was also made to measure the point-to-point transmission time and draw some conclusions regarding the transmission times. However, it did not make much sense to do it, since we did not have any real AFDX model that implemented our same network, nor any other, and therefore no reference to whether the times obtained were good or not.

Tested the proper functioning of the configuration and software we decided to focus on two more points only. The first point was to measure the latency of a single VLAN in a switch trying to visualize the delay introduced by a switch on a VLAN depending on the priority and step to see the operation of the SPQ in the queue. The second point was to develop a more complex software that would allow us to really emulate an ES allowing us to configure all the necessary VLANs in a single Linux computer and not just one per computer.

B. *Networks results*

At this point we will work with a very simple network example shown in Fig. 6. For this we measure first the technological latency of the switch (delay that introduces the technology in the network) and second the delay in VLAN 10 and 20 [9].

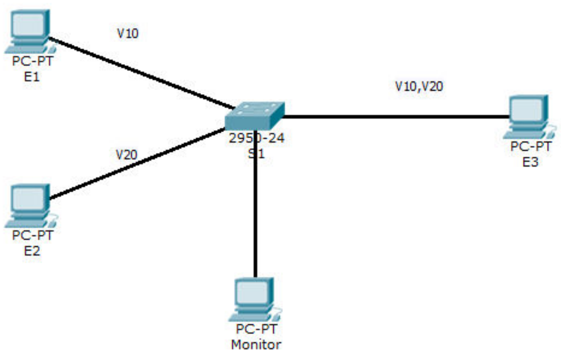


Fig.6. Monitoring network.

To measure the technological latency, we sent from E1 to E3 a frame of 1200 bytes every 128 ms and then every 1 ms while the network was empty. Once this is done, we have monitored the traffic at the input and output ports of the switch so that we can see the time difference, shown in Tables II and III.

TABLE II: Technology latency every 128 ms

Input time (ms)	Output time (ms)	Technology Latency (µs)
0	0.006	6
128.145	128.15	5
256.057	256.063	6
640.067	640.68	1
896.157	896.159	2
1408.339	1408.441	2

TABLE III: Technology latency every 1 ms

Input time (ms)	Output time (ms)	Technology Latency (µs)
0	0.008	8
0.958	0.96	2
1.958	1.96	2
8.953	8.955	2
9.953	9.954	1
13.2	13.202	2

These values were extracted randomly after having captured between 312 and 25000 frames respectively in each capture. However, more data were shown to attest that the data shown in the tables were representative.

As you can see, the technological latency is over 2 µs, although sometimes we find values that are a little larger or smaller. According to the ARINC 664 standard this value should not exceed 100 µs and we can conclude that in this case the requirement is accomplished.

C. *Emulating the end system*

We have already explained that the ES is responsible and that it needs part of the first program that allowed us a first approximation to an emulation. However, as we have said, it did not allow us to introduce more than one interface and therefore it did not end up being an ES. It also introduced a much more expensive task of configuring the network.

It is for this reason that an improved software was decided to implement following the indications of the ARINC 664.

First, an ES must have a FIFO stack of 8 Kbytes, according to the specification in the standard. This means that before sending the frames they have to go through the FIFO type queue. Although the concept is easy to understand, the program model is a little more complex and which we will explain briefly below [10].

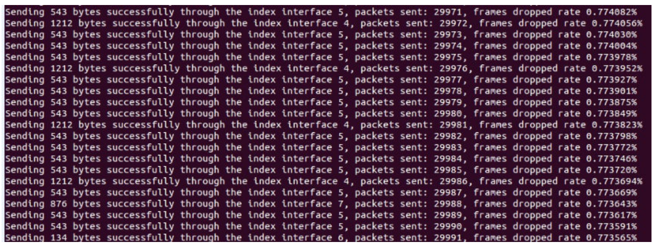
When we start the program, it will ask us how many VLANs we want to introduce. From there, it will create as many parallel processes as the number that we have entered. Then we will enter the values of each VLAN manually (interface, BAG, MTU, VLAN number and priority). Once this is done, each process will create its frames independently. These will be read by the parent process that will put them in a reception buffer and from there to the FIFO if there is enough space available, if not, a message announcing "Frame dropped" will be displayed and the program will continue running normally. Finally, the first value stored in the FIFO will be stored in an output buffer that will send the data to the output of the interface and it will be announced that those bytes have been left free in the stack.

The child processes created when we specify how many VLANs we want in our ES are executed in an infinite loop with a waiting time between creation and sending of frames to the reception buffer of the parent process equal to the value entered as BAG. In the same way, the parent process executes an infinite loop without any kind of wait if there is data in the FIFO or in the reception buffer, but it will wait until there is data available to read.

To verify the correct operation of the program we have used the example shown in Table IV, obtaining the next output of the program in Figure 7.

TABLE IV: Configuration example

Interface	BAG	MTU	VLAN	Priority
eth0.10	4	1212	100	1
eth0.20	1	543	1200	1
eth0.30	16	134	500	1
eth0.40	64	876	304	1



```
Sending 543 bytes successfully through the index interface 5, packets sent: 29971, frames dropped rate 0.774082%
Sending 1212 bytes successfully through the index interface 4, packets sent: 29972, frames dropped rate 0.774056%
Sending 543 bytes successfully through the index interface 5, packets sent: 29973, frames dropped rate 0.774030%
Sending 543 bytes successfully through the index interface 5, packets sent: 29974, frames dropped rate 0.774004%
Sending 543 bytes successfully through the index interface 5, packets sent: 29975, frames dropped rate 0.773978%
Sending 1212 bytes successfully through the index interface 4, packets sent: 29976, frames dropped rate 0.773952%
Sending 543 bytes successfully through the index interface 5, packets sent: 29977, frames dropped rate 0.773927%
Sending 543 bytes successfully through the index interface 5, packets sent: 29978, frames dropped rate 0.773901%
Sending 543 bytes successfully through the index interface 5, packets sent: 29979, frames dropped rate 0.773875%
Sending 543 bytes successfully through the index interface 5, packets sent: 29980, frames dropped rate 0.773849%
Sending 1212 bytes successfully through the index interface 4, packets sent: 29981, frames dropped rate 0.773823%
Sending 543 bytes successfully through the index interface 5, packets sent: 29982, frames dropped rate 0.773798%
Sending 543 bytes successfully through the index interface 5, packets sent: 29983, frames dropped rate 0.773772%
Sending 543 bytes successfully through the index interface 5, packets sent: 29984, frames dropped rate 0.773746%
Sending 543 bytes successfully through the index interface 5, packets sent: 29985, frames dropped rate 0.773720%
Sending 1212 bytes successfully through the index interface 4, packets sent: 29986, frames dropped rate 0.773694%
Sending 543 bytes successfully through the index interface 5, packets sent: 29987, frames dropped rate 0.773669%
Sending 876 bytes successfully through the index interface 7, packets sent: 29988, frames dropped rate 0.773643%
Sending 543 bytes successfully through the index interface 5, packets sent: 29989, frames dropped rate 0.773617%
Sending 543 bytes successfully through the index interface 5, packets sent: 29990, frames dropped rate 0.773591%
Sending 134 bytes successfully through the index interface 6, packets sent: 29991, frames dropped rate 0.773565%
```

Fig. 7. Output of the program in Linux Command Prompt.

As you can see in Figure 7, the 4 interfaces are sending the expected. We can also observe an average of rejected frames due to the fact that the FIFO was filled with approximately 0.77%.

VI. Conclusions

In this paper we have demonstrated that an AFDX network can be configured based on the typical characteristics of VLANs in Cisco commercial switches. However, it will be a future objective to conclude whether really after making the relevant network configurations, the delays, queues, ... are fast enough to obtain results similar to those of an actual AFDX network.

An interesting point could be to study in depth OMNET ++ to build simulations of AFDX switches from the existing core4inet framework developed for real-time networks and compare this simulation with the times obtained in a network emulated with commercial switches.

On the other hand, it is also a conclusion that it is possible to emulate an ES with Linux computers respecting traffic policies in accordance with the ARINC 664 standard. In this case it could also be interesting for a future to add to the program a specific type of distribution for the injection of traffic such as: Poisson, normal, ... so that this is not constant and there are variations in it as well as in the size of the frame (always without exceeding the Smax) to make it even more real

REFERENCES

[1] "*Draft 3 of project paper 664 aircraft data network part 7 Avionics Full Duplex switched ethernet (AFDX) network",* AERONAUTICAL RADIO, INC. July 1, 2004.

[2] "*AFDX® / ARINC 664 Tutorial",* TechSAT GmbH, Poing (Germany), 29th August 2008.

[3] Christian M. *"The Evolution of Avionics Networks From ARINC 429 to AFDX"*, Faculty of Informatics, Technical University of Munich, Seminar Aerospace Networks (SS2012), Network Architectures and Services, August 2012.

[4] Henri Bauer, Jean-Luc Scharbarg, Christian Fraboul, "*Worst-case end-to-end delay analysis of an avionics AFDX network",* Airbus France Université de Toulouse - IRIT/ENSEEIHT/INPT. March 2010, Design, Automation & Test in Europe Conference & Exhibition (DATE 2010).

[5] J. Javier Gutiérrez, J. Carlos Palencia and Michael González Harbour *"Response time analysis in AFDX networks with sub-virtual links and prioritized switches", Santander (Spain) February 2012* , XV Jornadas de Tiempo Real, JTR 2012.

[6] William D. Wargo and Joachim Schuler, "*Testing aircraft AFDX systems" Test & Measurement World, special issue, Cambridge (USA), May 2012.*

[7] *"Protocols for Aerospace Control Systems A Comparison of AFDX, ARINC 429, CAN, and TTP",* TTTech Computertechnik AG, 2005.

[8] Linux socket part 17 Advanced tcp/ip - the raw socket program examples: http://www.tenouk.com/Module43a.html *(available in october 2019).*

[9] Sending raw Ethernet packets from a specific interface in C on Linux: https://gist.github.com/austinmarton/1922600 *(available in october 2019).*

[10] Dongning Qu ; Bo Yang ; Tao Gao ; LuLu Yuan ; Xi Chen "*ARINC664 bus function test and its fault injection based on Ethernet card*" 2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA).