KINETIK, Vol. 2, No. 4, November 2017, Pp. 251-262
ISSN : 2503-2259
E-ISSN : 2503-2267

Design and Implementation of COTS-based Aircraft Data Network Using Embedded Linux

Catur Wirawan Wijiutomo*[1], Endro Ariyanto[2]
[1,2]Universitas Telkom
caturwijiutomo@telkomuniversity.ac.id[*1], endroa@telkomuniversity.ac.id[2]

Abstract

Aircraft Data Network (ADN) is a data communication developed specifically for the aircraft environment. Such environment requires data communication system that can work in real-time and has a high level of reliability. One of the standards in ADN is Avionics Full-Duplex Switched Ethernet (AFDX) based on the ARINC 664 specification being data communication standard for ADN utilizing IEEE 802.3 standard on physical layer 1 and 2. There are some challenges to implement the standard on the component of COTS By utilizing Linux-based embedded systems. The features designed and implemented in this paper are the switching and fault tolerant data components of ARINC 664. From the tests obtained, there are several results showing that ADN functional features can be implemented and can simulate ADN mechanisms including fault tolerant capabilities. However, there are performance limitations exemplified by the obtained average jitter value of 3380 microseconds not meeting the requirements for aircraft use required to have the range of 500 microseconds.

Keywords: *Aircraft Data Network, AFDX, COTS*

1. Introduction

Started from 1988 to the most recent implementation on Airbus A320 aircraft, the aircraft industry has switched to an all-electronic fly-by-wire system. Aircraft is fully controlled using electrical signals and no longer utilizing mechanical signals. Moreover, aircraft employs a real-time system requiring reliable data communication between the avionics subsystems in the aircraft. Boeing began developing and introducing a full Aircraft Full Duplex Switched Ethernet (AFDX) platform that is implemented based on the ARINC 664 standard specification using IEEE 802.3 (Ethernet Commercial Off The Shelf (COTS)) components on the physical layer and data link layer [1].

In the case of special engineering such as aircraft, the problems encountered in the development of aircraft technology are the duration of development time and the amount of fund required for carrying out related research. The use of Ethernet as a component of COTS will save time and development costs because Ethernet compliant with the IEEE 802.3 specification standard itself is already a mature technology and continues to grow since 1970 and continues to be developed to present time [2]. However, designing a system that only capitalizes the COTS component to achieve a deterministic performance according to real-time system requirements can present a challenge. A situation needed to be proven by this study.

The functional feature that will be selected to be implemented in COTS for aircraft data network is data switching serving to forward data from the sender to the receiver at a particular address deterministically [3][4]. The deterministic nature of the functional data switching is defined as:

1. A network having a jitter value of <500 microseconds to ensure almost zero delay variance.
2. Having latency value of <150 Ms.
3. Limiting package receive meeting frame filtering rules and ensuring all data transmissions completed on paths and destinations defined by the traffic policing rules.

In addition to the deterministic switching data feature, there is also an analysis of fault tolerant feature implementation which is one of the requirements on real-time system in an environment required a high level of safety.

This paper describes the study and implementation to design Aircraft Data Network platform based on real-time system utilizing Embedded PC based on Linux as COTS component. This platform should meet ADN standard employing ARINC 664 specification. The stages include

Wijiutomo, C., & Ariyanto, E. (2017). Design and Implementation of COTS-based Aircraft Data Network Using Embedded Linux. KINETIK, 2(4). doi:http://dx.doi.org/10.22219/kinetik.v2i4.229
Paper submitted on June 14, 2017; Revision on July 19, 2017; Received August 11, 2017

software and hardware designing and developing in accordance with the needs of the system before finally tested to conclude the system made entirely with the COTS component can meet the deterministic network properties expected by Aircraft Data Network (ADN). In systems implemented in the ADN scope, a deterministic network is required, so the designed switches must be able to handle frame filtering, traffic policing and of course fault tolerant functions to ensure that all data frame transmissions on the system are running deterministically.

2. Research Method

The research was started by carefully studying ARINC 664 standard features to be utilized to establish aircraft data network system. There were some relevant standards employed in this study as a reference of aircraft data network [1]:

1. The application of ARINC 664. ARINC 664 is a communication standard created to meet the needs of the modern aircraft industry requiring high bandwidth for data connections between subsystems. Moreover, it uses minimal cable connections to reduce aircraft load. This has not yet been achieved by its predecessor, ARINC 429. In a system having many end points like an Airbus, the unidirectional use of point-to-point data buses becomes extremely ineffective. This is a weakness of ARINC 429, having a standard requiring a lot of cabling that will increase the load of aircraft. On the other hand, ARINC 664, implementing Ethernet usage, can use switching techniques to meet the same infrastructure needs, with much less cabling.
2. The application of virtual link. AFDX uses the concept of a virtual link to replace the unidirectional bus connection concept used by the previous standard, ARINC 429. Using virtual links, multiple point-to-point connections were made in the network without the need for physical cables for each link. Thus, minimum cabling on an aircraft can be managed. Since networks on AFDX are profiled, addressing and bandwidth requirements for each VL must also be initially determined. In its physical topology, AFDX consists of 3 components, namely Avionics Subsystem, End Systems and AFDX Interconnect (Switch).
3. The application of Full Duplex Switch. Full duplex communication can transmit and receive packages simultaneously. This will overcome the possibility of collision occurring in case of existing transmission. If multiple collisions occur, large delay transfers are also possible. It is certainly not acceptable in aircraft data networks requiring reliable and real-time connections. It underlies why Full Duplex Switched Ethernet can eliminate the possibility of collision, becoming absolute standard on ADN. Figure 2 illustrates each End-System embedded in each avionics subsystem connected to the switch through Full-duplex link consisting of twisted pair to transmit (Tx) and twisted pair to receive (Rx).
4. The application of Virtual Links. The main idea of Virtual Links was based on ARINC 664. Part 7 maintained the concept of point-to-point links. However, in reducing the number of wiring by changing the use of physical paths on point-to-point connections such as ARINC 429, specified virtual path was chosen. The following are some specifications of Virtual Links based on ARINC 664 standard:
   - In theory, a network can define up to 64k (216) Virtual Links based on a 16-Bit Identifier on a MAC Destination Field in an Ethernet Frame.
   - An End-System can have more than one Virtual Link.
   - End-System performs Traffic shaping and Integrity Checking on every Virtual Link.
   - Switches Traffic manages policing on each Virtual Link.
   - By combining Traffic shaping and policing, the formation of an outline of deterministic network behaviour can be established.

In addition to functional communication standards, Standard fault tolerant was applied. Fault tolerance is the ability of the subsystem to recover from component failures without interrupting the service. Devices with fault tolerant capabilities usually have backup strategies with redundant components and control each other. One component is passive or in a stand-by state and other components are active. The passive components will be activated when other components fail and will provide services that were previously provided by failed components.

For the method of obtaining fault tolerance, the basic principle of fault tolerant design was redundancy. Here are three basic techniques used to meet the fault tolerant system [5]:

1. Spatial (Hardware Redundancy).
   This technique was established by creating hardware being identical to be a backup in the time of main hardware failure.

2. Informational (Redundant Data Structure)
   This technique was established on more software oriented. Therefore, this technique was more flexible because the software can be changed or adjusted to the needs of the existing system.
3. Temporal (Redundant Computation)
   This technique usually required computation and was considered having slow speed in the system recovery process after experiencing failure compared to spatial techniques.

3. Design and Implementation

3.1 System Design

The built system consists of two parts. The End System included AFDX Package Transmitter being responsible for sending AFDX packages and AFDX Switch being responsible for receiving the packages as well as forwarding them to another port based on the specified rule. This rule was implemented using Virtual Link illustrated by Figure 1. When handling Hardware approach, both software will be implemented on the Embedded PC with the design of the previously mentioned specifications. AFDX Package Transmitter will be installed on all PCs acting as End System while AFDX Switch was installed on all PCs acting as Switch. Because the AFDX package structure is different from ordinary TCP/IP packages, it took programming at the data link layer level to be able to manipulate raw packages into AFDX packages that met the ARINC 664 standard specification. The solution of the problem employed Libpcap as a library manipulating package frames at the kernel level, capturing and sending raw package regardless of protocol stack used [6][7].

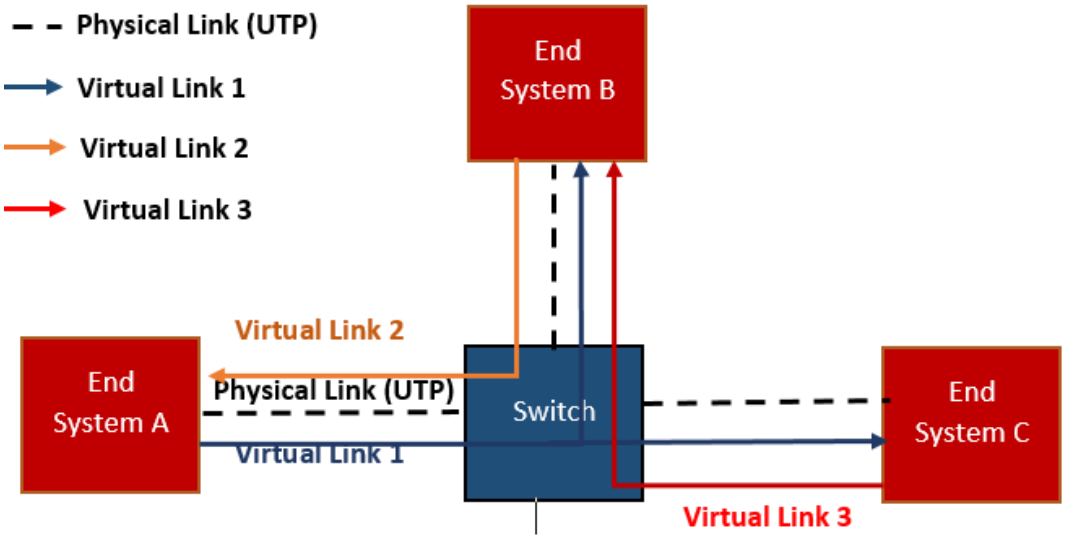

Figure 1. ARINC 664 System

In this paper, AFDX packages were built by relying on the data link layer manipulation capabilities of Libpcap as well as native Linux libraries having the ability to manipulate and define their own headers and content from a raw Ethernet frame. In the established system, the AFDX package structure was defined to have 14 Bytes Ethernet Headers (6 Bytes Destination Address, 6 Bytes Source Address, 2 Bytes Type), 20 Bytes IP Headers, 8 Bytes UDP Header and 58 Bytes AFDX Payload wrapped and defined to a data structure that will be injected to the package to send AFDX Package Transmitter. This procedure is depicted by Figure 2.

The sender and receiver system were divided into two End Systems. For example, the sender End System and the receiving End System (sniffer). Sender was responsible for sending packages redundantly and sniffer was responsible for receiving the package. Both End Systems will be built on two PCs according to the design specification. One PC will act as a sender, and the other PC will act as a sniffer.

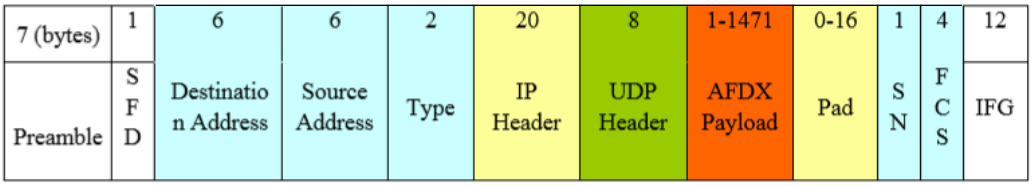

Figure 2. ARINC 664 Frames

AFDX Package Transmitter was part of a system implementation having task of delivering pre-defined and pre-built AFDX packages from PC End System to another End System via the embedded PC Switch to AFDX Switch. AFDX Package Transmitter was written in C language, compiled using GCC and supported by Libpcap libraries and native libraries from Linux to enable applications sending AFDX packages to the network [8].

Some alternatives that have been addressed to implement fault tolerant features on two designed interfaces are:

1. Fork & Thread Method
   Based on the results of the analysis when the system was built with the fork and thread method, it is concluded that this method becomes less appropriate to use because of the delay when eth1 took over the transmission of big data causing a number of missing packages.
2. Bonding Method: Broadcast
   From a technical point of view, this method is reliable and the most similar method to AFDX. On the other hand, looking at the device specifications used in the system manufacturing process, this method is considered too heavy. Hence, the device cannot handle the package receiving process, especially in large quantities because the End System required to dismantle, read and eliminate the same package. Meanwhile, the system does not apply the principle of scheduling, causing the number of missing packages.
3. Bonding Method: Active – Backup
   The difference of this method with broadcast method is in the delivery process involving 2 pieces of path where a path is passive and only active when the main line fails. Thus, there is fewer number of packages received in the End System.

Based on the methods mentioned above, it was decided to make both End Systems work redundantly and became fault tolerant. The method used was bonding: active-backup. By utilizing bonding, the two interfaces on each end-system will be regarded as the same path. When sending the file only one slave (interface) was active, another slave will be active if the first slave was interrupted. When bonding was active, MAC address read recipient only when MAC address slave was active.

AFDX Switch was a part of system implementation which was responsible for frame filtering and traffic policing of packages sent by AFDX Package Transmitter. After receiving the AFDX package sent by the transmitter, AFDX Switch checked the validity of the received package. If the packages met the established rule, then the package will be forwarded to the destination End System based on the defined Virtual Link ID. Similar to AFDX Package Transmitter, AFDX Switch was also written in C language, compiled using GCC and supported by Libpcap libraries and native libraries from Linux enabling applications built to receive and forward AFDX packages to the network. The use of System Call Fork from Linux also allowed the application to handle many network interface adapters at once. In addition, because the system was implemented with the Embedded Device approach, the program was also embedded as a Linux service. Therefore, the program always run automatically whenever any system was booted.

Embedded system used for switches as shown on Figure 3 was designed using an Intel i5 processor having an x86 processor architecture. The x86 processor architecture is now widely used in many embedded systems. This is because as a component of COTS. This architecture continues to evolve with new innovations, but with backward compatibility, efficient energy consumption, lots of support tools available for development as well as more varied costing schemes depending on needs. In the field of embedded networking, the x86 architecture is ideally used on systems with bandwidth of 10/100 Mbps [2]. Because the built system was designed with

ethernet and UTP cables that had bandwidth of 100 Mbps, this architecture was certainly sufficient for use in the system. The PC used as a switch required 3 ports for testing because the motherboard already had 1 default ethernet port. Hence it needed other 2 ethernet ports. It was achieved by adding an ethernet adapter card through the PCI slot and a USB ethernet adapter due to the limitations of PCI slots provided by the motherboard. PCI Bus had a bandwidth of 133 Mbps [9] and USB had a bandwidth of 480 Mbps [10], so it was adequate to use an ethernet adapter possessing a bandwidth of 100 Mbps.

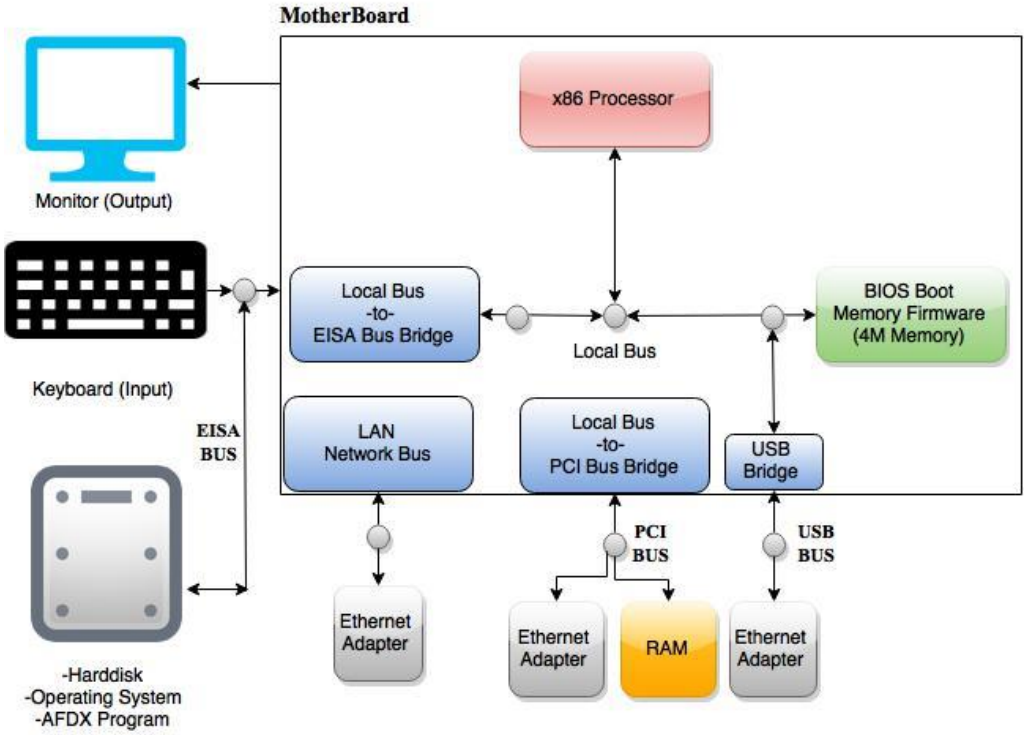

Figure 3. Embedded System Design

The explanation of the previous Figure 4 is as follows [11]:

1. At the initiation stage, Package Transmitter will build the AFDX package from the datalink layer then defined the value of the Virtual Link ID of the package
2. The transmitter will send the AFDX package to the destination End System according to the defined Virtual Link ID
3. When the Switch received the package, it will initially check whether the package was coming from the corresponding Virtual Link ID. Otherwise, the package will be dropped. In the case of appropriacy, it will perform frame filtering according to the defined rule. If the package did not match the rule, then the package will be dropped. In the case of appropriacy, the package will be forwarded throughout-interface according to the definition of Virtual Link ID that had been previously given. If the out-of-interface port packages still passed through the Switch again, it will be checked and forwarded again, but if not, the package can be directly received by End System.

The system was implemented by running both pre-made programs, AFDX Package Transmitter and AFDX Switch on 4 pre-designed embedded PCs. The AFDX Switch program ran as a Linux service on a PC that will act as a switch, so the program will always run automatically every time the system was booted to meet the embedded system implementation approach. The AFDX package transmitter program was executed through 3 PC units acting as End System and was assigned sending AFDX packages to other End Systems via a PC acting as an AFDX Switch. In addition, the service abortion and operating system modules were not required due to system load reduction. To run the test in accordance with the topology established in Figure 1, three PC units acting as End System will be physically connected to the PC acting as a switch through the Ethernet port using UTP cable. The process of system testing implementation was completed in

2 sessions. In the first session, all PCs used for implementation use Ubuntu standard as the operating system. After the first test session was performed and the results were recorded, the Ubuntu operating system used in the previous test was customized using ChronOS, and a second testing session was conducted.

In order for the established system to send files redundantly and became fault tolerance, bonding on Linux system interfaces was conducted. It was done with the/etc/network/interfaces file on Linux. By utilizing bonding on two interfaces that were configured to be active-backups, the two interfaces will be regarded as the same path. Therefore, if the main line experienced problems then the second line will take over the main line task.

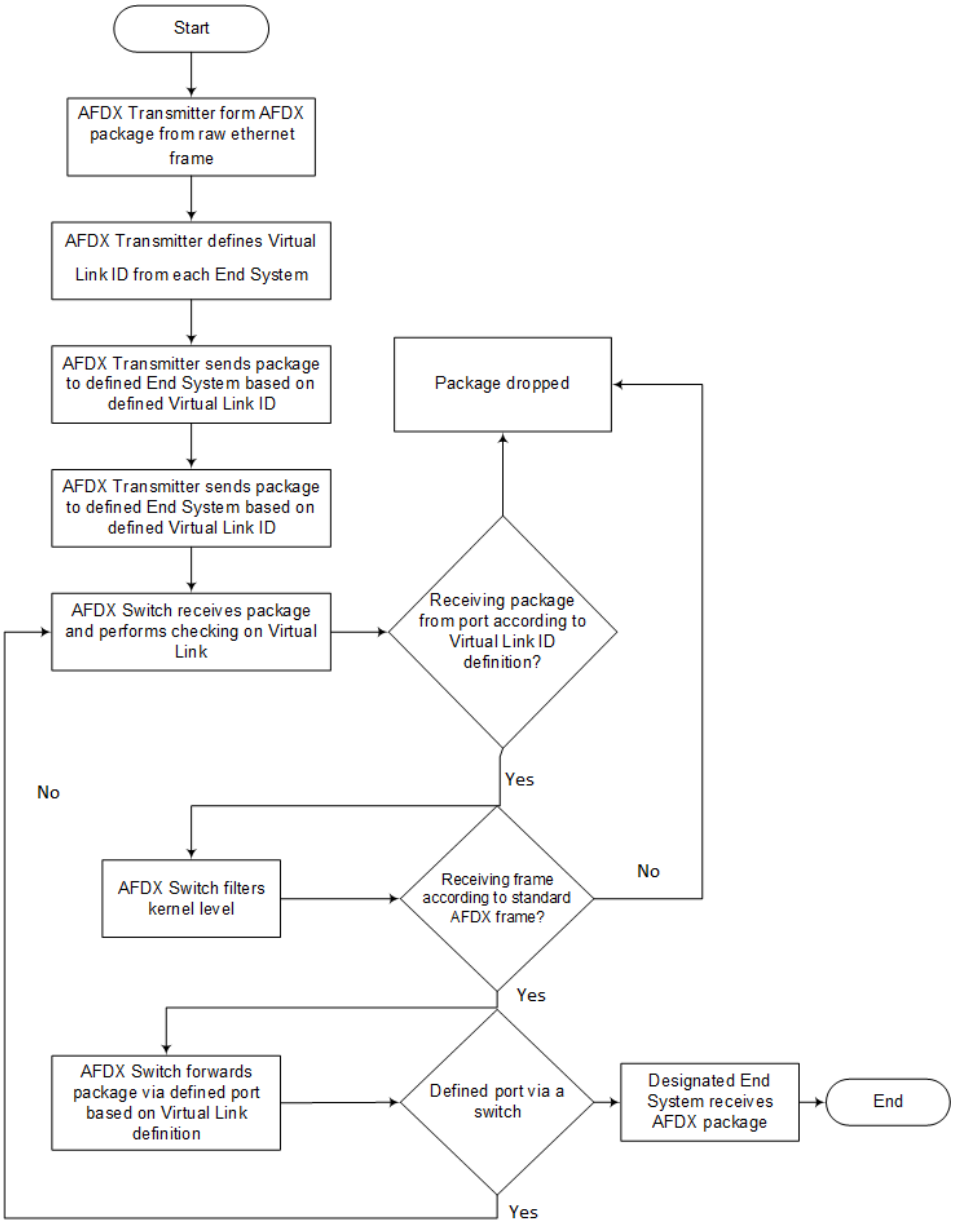

Figure 4. Traffic Policy [11]

3.2 System Testing

This chapter will discuss the testing and analysis from the designed system. The testing and analysis were completed with the help of Wireshark tools utilized to measure the system performance. The system was tested on two platforms namely Ubuntu 14.04 and ChronOS. The test results of both platforms were then compared, so that it can be concluded which platform was more reliable for system implementation.

3.2.1 Data Switching Features

The testing will be accomplished using the help of Wireshark tools, used to analyze the existing packages on the network. The testing used the following specifications in Table 1 and Table 2.

Table 1. Test Parameter

Parameters	Defined values
Lmax	1518 byte
Max Jitter	500 Microseconds
Max Latency	150 Milliseconds
Lmin	64 bytes
Bandwidth	100000 bit/s
Frame sent	10 frames
Frame validity	Valid/not valid

Table 2. Traffic Parameter

No	Multicast/Unicast	Sender	Receiver	Virtual Link ID
1	Multicast	ES A	ES B & ES C	01
2	Multicast	ES B	ES A & ES C	02
3	Multicast	ES C	ES B & ES A	03
4	Unicast	ES A	ES B	04
5	Unicast	ES B	ES C	05
6	Unicast	ES C	ES A	06

The testing for functional routing is completed based on the rules in Table 2. The package delivery was performed 5 times on each platform. Moreover, the availability of the switch to recognize which packages should be dropped based on the pre-defined package size limit specification was tested.

The test obtained results informing that the system was able to filter the package based on the size of its frame as shown in Table 3, so the delivery of packages with the size larger than Lmax of 1518 bytes or smaller than Lmin of 64 bytes will cause the package not getting to the destination, showing 0%. Nevertheless, the package sent to the specification specifications succeeded 100% to the destination. From both tests, it can be seen that there was no difference between two platforms. Conclusively, the real-time Linux kernel does not add too much points in terms of ensuring the validity of the passing frame through the filtering functionality of the system. In addition, COTS component is sufficiently reliable to ensure the validity of the AFDX package based on its frame size.

Table 3. Traffic Parameter

No	Source End System	Virtual Link ID	Destination End System	Package Size (bytes)	Package Received (%)	Validity
1	ES A	01	ES B & ES C	100	100 %	Valid
2	ES A	04	ES B	100	100 %	Valid
3	ES B	02	ES C	100	100 %	Valid
4	ES C	03	ES A	100	100 %	Valid
5	ES A	01	ES B & ES C	1600	0 %	Valid
6	ES A	04	ES B	1600	0 %	Valid
7	ES B	02	ES C	1600	0 %	Valid
8	ES C	03	ES A	1600	0 %	Valid
9	ES A	01	ES B & ES C	50	0 %	Valid
10	ES A	04	ES B	50	0 %	Valid
11	ES B	02	ES C	50	0 %	Valid
12	ES C	03	ES A	50	0 %	Valid
% Validity of Frame Filtering Rules						100%

Based on the AFDX specification in ARINC 664, latency is defined as the time between when the package is ready to transmit with the duration of completed transmission. Therefore, in this paper, the latency was calculated as the average of the time difference between package transmissions. The package delivery time will be calculated with the help of Wireshark tools, by comparing Linux operating system with Ubuntu and ChronOS. The time measurement format was Epoch Time (Unix Time). The difference between arrival time between packages was compared, and the results are illustrated in Figure 5.

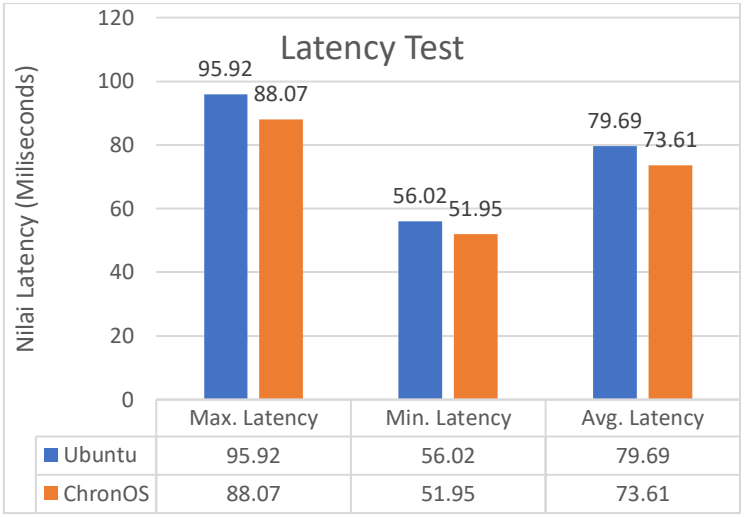

Figure 5. Latency Test on ChronoOS and Ubuntu

The system implemented in ChronOS had a lower latency value than the system implemented in Ubuntu, so it can be concluded ChronOS had a higher performance compared to Ubuntu with a standard Linux kernel. In addition, COTS components were reliable enough to meet the requirement of max latency <150 ms based on ARINC 664 specification [1].

Large Jitter measurements on the system were calculated using Wireshark tools. AFDX transmitter sent packages from End System A to End System B. Wireshark was installed on both End Systems. Afterwards, the delivery and arrival time of the package were recorded and calculated based on the difference of the obtained delay value.

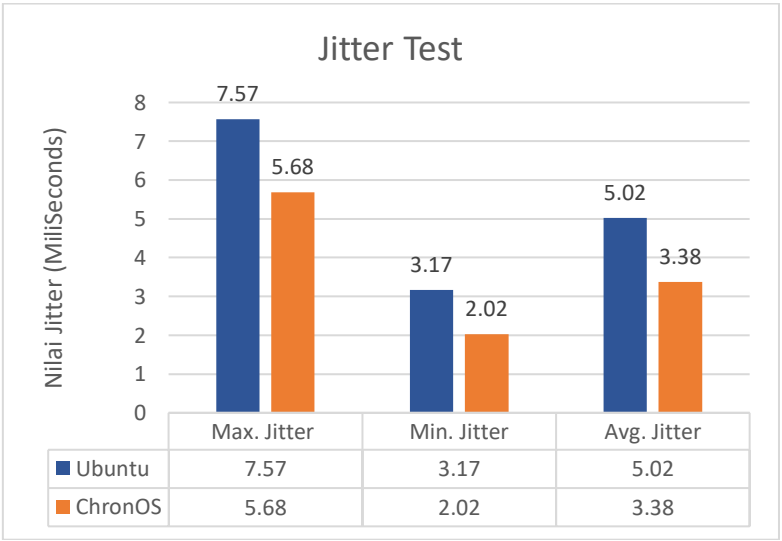

Figure 6. Jitter Test on ChronoOS and Ubuntu

In measuring the jitter value, the test results show that the system implemented on ChronOS had a lower jitter value than the system implemented in Ubuntu as shown in Figure 6.

Hence, it can be concluded that ChronOS had a higher performance compared with Ubuntu using a standard Linux kernel. However, the obtained average jitter value was 3380 microseconds failed to the ARINC 664 specification having max jitter of 500 microseconds (Committee, 2002). It was because the hardware used was not adequate. Additionally, there are some weaknesses in the Linux 2.4 scheduler, experiencing the average of high time-slices, poor I / O-bound task priority, and weak support for real-time application. To be able to meet this specification, in the next study, it is recommended to use Real Time Operation System platform designed to meet Flight-Critical System specifications and uses a more reliable Ethernet Adapter.

3.2.2 Fault Tolerant Features

The fault tolerant feature used redundancy on two interfaces using bonding feature on Linux [12]. The test results are presented in Table 4 and Table 5.

Table 4. Fault Tolerant Scenario Test

No	Conditions	Description	Results		Status
			Planning	Actual	
1	Package delivery via interface eth0 and eth 1 under normal circumstances	eth0	450 packages are sent through eth0	450 packages are sent through eth0	success
		eth1	450 packages are sent through eth1	450 packages are sent through eth1	success
2	Package delivery through interface bond0 under normal circumstances.	-	450 packages are sent through eth0	450 packages are sent through eth0	success
3	Package delivery via bond0 when eth0 and eth1 have hardware failure (error)	eth0 *error*	450 packages are sent through eth1	450 packages are sent through eth1	success
		eth1 *error*	450 packages are sent through eth0	450 packages are sent through eth0	success
		eth0 *error* when sending package	450 packages are sent	450 packages are sent	success
		eth0 *error* and back to normal state	450 packages sent through eth0, eth1 and back to eth0	450 packages sent through eth0, eth1 and back to eth0	success
4	Testing timing on fault tolerance	-	-	2,652505 Ms	unsuccessful

Table 5. Fault Tolerant Test

Sending	Output Time	
	Average Normal Package Increment (Microseconds)	Difference of Error (Microseconds)
1	8845,324	18440
2	10972,28	9520
3	9808,054	20150
4	9822,26	8020
5	8421,655	9590
6	10744,79	14220
7	9945,548	8960
8	9862,17	15340
9	9383,097	9170
10	9249,799	10170
Average Delivery (milliseconds)	9,705498	12,358
Average Difference Delivery (milliseconds)	2,652505	

4. Conclusion & Future Works

The previous chapters have shown and explained about the results of simulations conducted by the authors, so that the results of the analysis obtain some conclusions and provide suggestions for further research as follows:

Based on AFDX functional related tests that have been conducted in this paper, the following conclusions can be drawn:

1. The COTS component can perform the AFDX package formation function using raw Ethernet package, check the validity of AFDX packages using frame filtering function, and well define the virtual link path for those packages using traffic policing function. Therefore, it can meet the AFDX network characteristics where all data transmissions must be defined statically.
2. Latency with an average value of 73.61 milliseconds for package delivery made by a system using ChronOS is able to meet the ARINC 664 standard specification of <150 Ms, so the COTS component is considered reliable enough to implement a system with a latency value meeting the specification.
3. Jitter experienced by the system with an average value of 3380 microseconds still cannot meet the standard specifications. According to ARINC 664, the minimum jitter of the system must be <500 microseconds, so the system has not been able to meet the deterministic network properties expected by the Aircraft Data Network.
4. The operating system kernel greatly affects the performance of package processing. Systems that have kernels designed to achieve real-time properties such as ChronOS have better performance than the standard Linux kernel compared to that of Ubuntu.
5. Although not yet being able to apply the same redundancy principle as AFDX, the established system using COTS ethernet component can transmit redundant data and be fault tolerant.

For fault tolerant feature, the system is able to guarantee data availability when hardware error occurs although the resulting delay (2.65 Ms) has not met the maximum delay limit set by AFDX (0.5 Ms). The suggestions for the next research include:

1. To add Frame Check Sequence (FCS). Check the frame can be performed more accurately. Use of FCS will cause errors or broken frames to be dropped so as not to overload the network.
2. To use scheduling function specially designed for real-time system. It is intended that the BAG rate of the system can be set and measured, and therefore, obtained a more reliable performance because the Linux scheduler as a default scheduling function of the Linux kernel used on the current system is still less reliable to handle the needs of real-time systems
3. To use Realtime Operating System (RTOS) designed for Flight-Critical System such as RTLinux and VMWorks. Hence, the performance can reach the size meeting the standardized requirement.

References

[1] A. Committee, "*Aircraft Data Network Part 7, Avionics Full Duplex Switched Ethernet (AFDX) Network, ARINC Specification 664*," Annapolis, Maryl. Aeronaut. Radio, 2002.
[2] K. Christensen, P. Reviriego, and B. Nordman, "*IEEE 802.3 az: The Road to Energy Efficient Ethernet*," IEEE, 2010.
[3] H. Jung, "*Fast Transmission Mechanism of Emergency Data in AFDX Network Systems*."
[4] T. DeChiara, "*Flight Control Computers with Ethernet Based Cross Channel Data Links*," US Pat. App. 11/710,207, 2007.
[5] A. Sreekumar, K. Swetha, A. Swetha, and V. R. Pillay, "*Enhanced Performance Capability in a Dual Redundant Avionics Platform – Fault Tolerant Scheduling with Comparative Evaluation*," Procedia Computer Science, Vol. 46, Pp. 921–932, 2015.
[6] F. Brajou and P. Ricco, "*The Airbus A380-an AFDX-Based Flight Test Computer Concept*," AUTOTESTCON 2004. Proceedings, 2004.
[7] P. B. Champeaux, D. Faura, M. Gatti, and W. Terroy, "*A Distributed Avionics Communication Network*," in 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), Pp. 206–209, 2016.
[8] Y. Xiao, L. Li, and J. Wen, "*Network Program Architecture Based Winpcap and Sock*," Ordnance Industry Automation, 2005.
[9] P. Sig, "*PCI Local Bus Specification Revision 2.2*," PCI SIG, 1998.
[10] U. Specification, "*Revision 2.0, 2000*," Hewlett-Packard Company, Intel Corporation, Lucent.
[11] Faisal Defry Hussainy, "*Implementation Full Duplex Switced Ethernet to Aircraft Data Based on Cots Embedded Device - Documents*," Universitas Telkom, 2015.
[12] A. Rahadian, "*Implementation of Data Redundancy to End–System Aircraft Data Network Based on Cots Embedded Device*," Universitas Telkom, 2016.