BAŞKENT UNIVERSITY
INSTITUTE OF SCIENCE AND ENGINEERING
DEPARTMENT OF DEFENSE TECHNOLOGIES AND SYSTEMS
MASTER OF SCIENCE IN DEFENSE ELECTRONICS AND SOFTWARE

AFDX (AVIONICS FULL-DUPLEX SWITCHED ETHERNET) NETWORK SIMULATION AND PERFORMANCE ANALYSIS

BY

İPEK PEŞKİRCİOĞLU GÖKÇE

MASTER OF SCIENCE THESIS

ANKARA – 2022

BAŞKENT UNIVERSITY
INSTITUTE OF SCIENCE AND ENGINEERING
DEPARTMENT OF DEFENSE TECHNOLOGIES AND SYSTEMS
MASTER OF SCIENCE IN DEFENSE ELECTRONICS AND SOFTWARE

AFDX (AVIONICS FULL-DUPLEX SWITCHED ETHERNET) NETWORK SIMULATION AND PERFORMANCE ANALYSIS

BY

İPEK PEŞKİRCİOĞLU GÖKÇE

MASTER OF SCIENCE THESIS

ADVISOR

ASSIST. PROF. DR. MURAT ÜÇÜNCÜ

CO-ADVISOR

PROF. DR. Ece GÜRAN SCHMIDT

ANKARA - 2022

BAŞKENT UNIVERSITY

INSTITUTE OF SCIENCE AND ENGINEERING

This study, which is prepared by İpek Peşkircioğlu Gökçe for the Defense Electronics and Software program, has been approved in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in the Defense Technologies and Systems Department by the following committee.

Date of Thesis Defense: …/…/2022

Thesis Title: AFDX (Avionics Full Duplex Switched Ethernet) Network Simulation and Performance Analysis

Examining Committee Members	**Signature**
Prof. Dr. Klaus Werner Schmidt, Middle East Technical University	....……....
Assist. Prof. Dr. Tülin ERÇELEBİ AYYILDIZ, Baskent University	....……....
Assist. Prof. Dr. Murat ÜÇÜNCÜ, Baskent University	....……....

APPROVAL

Prof. Dr. Ömer Faruk ELALDI

Director, Institute of Science and Engineering

Date: …. /…. /2022

.....

BAŞKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
YÜKSEK LİSANS TEZ ÇALIŞMASI ORJİNALLİK RAPORU

Tarih: …/…./2022

Öğrencinin Adı, Soyadı	:	İpek Peşkircioğlu Gökçe
Öğrencinin Numaras	:	22110116
Anabilim Dalı	:	Savunma Teknolojileri ve Sistemleri Anabilim Dalı
Programı	:	Savunma Elektroniği ve Yazılımı Tezli Yüksek Lisans Programı
Danışmanın Adı, Soyadı	:	Dr. Öğr. Üyesi Murat ÜÇÜNCÜ
Tez Başlığı	:	AFDX Network Simulation and Performance Analysis

Yukarıda başlığı belirtilen Yüksek Lisans tez çalışmamın; Giriş, Ana Bölümler ve Sonuç Bölümünden oluşan toplam ... sayfalık kısmına ilişkin, .../.../2022 tarihinde tez danışmanım tarafından Turnitin adlı intihal tespit programından aşağıda belirtilen filtrelemeler uygulanarak alınmış olan orijinallik raporuna göre, tezimin benzerlik oranı %...dir.

Uygulanan filtrelemeler:
1.Kaynakça hariç
2.Alıntılar hariç
3.Beş (5) kelimeden daha az örtüşme içeren metin kısımları hariç

"Başkent Üniversitesi Enstitüleri Tez Çalışması Orijinallik Raporu Alınması ve Kullanılması Usul ve Esaslarını" inceledim ve bu uygulama esaslarında belirtilen azami benzerlik oranlarına tez çalışmamın herhangi bir intihal içermediğini; aksinin tespit edileceği muhtemel durumda doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi ve yukarıda vermiş olduğum bilgilerin doğru olduğunu beyan ederim.

Öğrenci İmzası:
İpek Peşkircioğlu Gökçe
…….

ONAY
Tarih: …/…/2022
Öğrenci Danışmanı
Dr. Öğr. Üyesi Murat ÜÇÜNCÜ
…….

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Assist. Prof. Dr. Murat ÜÇÜNCÜ who supports me, gives me feedback and guides me through the process.

I would like to thank my co-advisor Prof. Dr. Ece Schmidt for her assistance and dedicated involvement in every step throughout the process. This thesis would have never been accomplished.

I would also like to thank my beloved family for supporting me throughout this thesis and my whole education life.

Finally, I must express my gratitude to my husband for providing me with continuous encouragement and unfailing support both morally and technically throughout this process.

İpek PEŞKİRCİOĞLU GÖKÇE Ankara-2022

ABSTRACT

İpek Peşkircioğlu Gökçe

AFDX Network Simulation and Performance Analysis

Baskent University Institute of Science and Technology

The Department of Defense Technologies and Systems

2022

AFDX (Avionics Full Duplex Switched Ethernet) also known as ARINC 664 Part 7 is a leading ethernet-based avionics data network used for safety-critical applications having real-time requirements with dedicated bandwidth utilizations. In order to construct a proper AFDX architecture, network configuration aspects such as Bandwidth Allocation Gap, Virtual Link assignment and network topology should be defined by considering performance metrics including line utilization, average and worst-case timings, switch queueing latencies and buffer occupancies in order to satisfy real-time requirements. This thesis is intended to present an AFDX Simulation model that evaluates mentioned aspects of the network before setting-up the actual system. To this end, first the existing OMNeT++ AFDX Model is improved to make a more realistic and easily configurable simulation. Additionally, in order to make the simulation modifiable for those who are not familiar with the OMNeT++ environment and get readable results, a new network configuration and analysis tool, named as ANCAT is proposed. AFDX model and ANCAT are verified with multiple custom-designed experiments and comparison to analytical queueing models. Finally, some realistic network scenarios that both evaluate AFDX performance and demonstrate the capability of the developed OMNeT++ model is represented.

Keywords: AFDX, Avionics Network, Network Simulation, OMNeT++, Performance Analysis

Co-Advisor: Prof. Dr. Ece GÜRAN SCHMIDT (Coadvisor), Middle East Technical University

Advisor: Assist. Prof. Dr. Murat ÜÇÜNCÜ (Advisor), Baskent University

ÖZET

İpek Peşkircioğlu Gökçe

AFDX Ağ Simulasyonu ve Performans Analizi

Başkent Universitesi Fen Bilimleri Enstitüsü

Savunma Teknolojileri ve Sistemleri Anabilim Dalı

2022

ARINC 664 Part 7 olarak da bilinen AFDX, günümüzde emniyet açısından kritik, gerçek zamanlı gereksinimleri ve kendine ayrılmış bant genişliği ihtiyacı olan hava aracı sistemlerinde yaygın olarak kullanılan gerçek zamanlı bir ethernet protokolüdür. Gerçek zamanlı gereksinimleri karşılayabilecek ve en kötü durumlarda bile beklenildiği gibi çalışabilecek bir AFDX mimarisi kurabilmek için AFDX'e has bant genişliği yerleşim aralığı (BAG), sanal bağ atamaları ve ağ yağısı gibi ayarlar, uçtan uca gecikmeler, anahtar gecikmeleri ve doluluk oranları gibi performans kriterleri göz önünde bulunudurularak tasarım yapılmalıdır. Bu çalışmada, bir AFDX mimarisini fiziksel olarak kurup çalıştırmadan söz konusu kıstasları elde edebilmek ve inceleyebilmek için kullanılmak üzere bir AFDX simülasyonu hazırlanması amaçlanmıştır. Bu amaçla, daha önceden oluşturulmuş OMNeT++ AFDX modelindeki eksiklikler giderilmiş, model daha gerçekçi ve kolayca konfigüre edilebilir hale getirilmiştir. Ayrıca, daha önce OMNeT++ ile uğraşmamış kişiler için de simülasonu daha kolay konfigüre edilebilir hale getirebilmek ve okunabilir simülasyon çıktıları elde edebilmek için yeni bir ağ konfigürasyon ve analiz aracı (ANCAT) geliştirilmiştir. Son olarak AFDX modeli ve ANCAT aracını kullanarak teorik ve gerçekçi senaryolar içeren pek çok deney yapılmış, bu deney sonuçlarından yola çıkarak ürünler doğrulanmıştır.

Anahtar Sözcükler: AFDX, Aviyonik Ağları, Ağ Simülasyonu, OMNeT++, Performans Analizi

Eş Danışman: Prof. Dr. Ece GÜRAN SCHMIDT, ODTÜ

Danışman: Dr. Öğr. Üyesi Murat ÜÇÜNCÜ, Başkent Üniversitesi

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SYMBOLS AND ABBREVIATIONS

ADIRU	Air Data Inertial Reference Unit
AFDX	Avionics Full Duplex Switched Ethernet
ARINC	Aeronautical Radio, Incorporated
ANCAT	AFDX Network Configuration and Analysis Tool
B	Byte
BAG	Bandwidth Allocation GAP
CI	Confidence Interval
COTS	Commercial Off-The-Shelf
ES	End-system
FCS	Frame Check Sequence
FIFO	First in, First out
FM	Flight Manager
FMS	Flight Management System
ID	Identification
IFG	Inter-Frame Gap
IEEE	Institute of Electrical and Electronics Engineers
IMA	Integrated Modular Avionics
KU	Keyboard Unit
LRU	Line Replaceable Unit
MAC	Media Access Control
MFD	Multi-Function Display
msec	milliseconds
OMNET++	Objective Modular Network Testbed in C++
OMNEST	An extended version of OMNET++–
PHY	Physical Layer Device
PSN	Previous Sequence Number
QOS	Quality of Service
RDC	Remote Data Center
Rx	Receive
SFD	Start Frame Delimiter
SN	Sequence Number
SQDSR	Shared Queue based Dynamic Slot Reservation
SW	Switch
TTEthernet	Time-Triggered Ethernet
Tx	Transmit
UDP	User Datagram Protocol
VL	Virtual Link
VL-ID	Virtual Link Identification

1. INTRODUCTION

Avionic systems involve dozens of electronic devices fitted into satellites, aircrafts or spacecrafts. These devices can be display units, navigation systems, communications modules, flight or fire control computers among many other devices [1]. Diverse devices utilize a diverse type of data and interfaces with different priorities. When all these are considered, communication between sub-units may be quite a challenge. To simplify the development and integration of avionics software and hardware, avionics systems are migrating towards integrated modular avionics (IMA) [2]. In IMA systems, flexible and reprogrammable modules with higher speeds have started to replace traditional, application-specific and non-adaptive avionics standards with lower bandwidths like MIL-STD-1553 [3]. Within this context, Avionics Full-Duplex Switched Ethernet (AFDX) protocol is standardized as ARINC (Aeronautical Radio, Inc) Specification 664 Part 7 in association with avionics manufacturers like Airbus, Boeing, Rockwell Collins, Honeywell, etc. [4]. Airbus A380/A350/A400M, Boeing 787 Dreamliner, ARJ21 and Superjet 100 can be counted among airplanes using AFDX [5].

The physical and Media Access Control (MAC) layers of AFDX are based on IEEE 802.3 Ethernet standard and it speeds up to 100 Mbps rates. Network architecture is composed of interconnected switches (SW) and end-systems (ES) communicating through those switches. AFDX ensures the deterministic quality of services (QOS) with dedicated bandwidth by establishing a connection-oriented structure. At the ES level, QoS support is provided by output traffic regulation and priority-based switching. In addition, AFDX offers a strong fault-tolerant network capability by using redundant switches and network interfaces.

In modern avionic networks, line replaceable units (LRU) are gathering more diverse data with larger amounts than before. Thus, the amount of real-time data that is circulating through the network is increased. End-to-end delay limits shouldn’t be exceeded in the devices used in distributed architectures to ensure that control loops run properly, especially if they are safety critical. Hence making a performance analysis before realizing the actual system is essential and may save lives [6].

When designing a network, worst-case scenarios can be foreseen with mathematical modeling [7]. Nonetheless, this mathematical model may either remain incapable of giving

average results or may bring out exceedingly pessimistic outcomes. To avoid such cases, it is crucial to evaluate the design with simulation models as realistic as possible.

NS2 [8], [9] and OPNET [10] simulators were used to simulate AFDX. However important details like message set or model parameters were not included formerly. If different real-time open-source network simulators are compared [11]–[13], it can be seen that the OMNeT++ framework is more advantageous in terms of timings, memory needs and visualization abilities compared to other popular network simulators like NS2/NS3. OMNeT++ [14] is used to create network simulations particularly. In [15], an AFDX Model is derived over TTEthernet layer and in [16] the results gathered from an AFDX model built over the INET Ethernet model [17] are compared with those gathered from TTEthernet hardware.

Among other simulation modules, there exists an AFDX Model that is originated by OMNEST[18] and then opened to community by OMNeT++ [19]. This module includes redundancy management and queueing behavior for both switch and end-systems in addition to the basic AFDX MAC layer and AFDX switch implementation.

The contributions of the thesis are as follows:

1) Extension, verification and update of the AFDX model developed and published by OMNeT++ [19] which previously developed by OMNEST++ for commercial use [18].
    - A realistic AFDX simulation model that is closely following ARINC 664 p7 standard [4]. The model implements the link layer AFDX functionality together with redundancy features.
    - Automatized simulation configuration which takes simulation parameters from an input configuration file in a standard format such as Microsoft Excel. Automatized simulation output report generation which provides detailed measurements results and summaries containing average and maximum measurement values per VL and per Switch.
    - Custom tailored experiments with deterministic and computable results for the verification of the simulation correctness. Furthermore, Little's Law is checked for the queues in the switch. A detailed

breakdown of latency components of the ES and Switch model is presented and compared with the expected results.

- Publishing the extended verified model to the community [20]

2) Determining realistic AFDX message parameters based on avionics components specifications and the current application expectations. Construction of realistic message sets and network topologies.
3) Extensive performance evaluation of AFDX under these realistic messages and topologies. Evaluation under selected fault scenarios.

2. BACKGROUND

2.1. AFDX Overview

Since previous standards could no longer meet the requirements of modern-day state-of-the-art air vehicles, AFDX was proposed and trademarked by AIRBUS [5]. It is based on IEEE802.3 Ethernet by physical and MAC layers and complies with UDP/IP in the transport layer [13]. It uses Ethernet frame definition and IEEE802.1d switching protocol but it is essentially different from commercial Ethernet which provides guaranteed bandwidth and bounded end-to-end latency.

Key AFDX network components are end-systems, switches and virtual links (VL) as shown in Figure 2.1. Switches are connected to either an end-system or each other. End-systems are inputs and outputs of the architecture and each end-system is connected to a switch by a certain switch port. The physical links between switches and end-systems are full duplex 100Mbps Ethernet lines. Switches are dual-redundant (Switch-A and Switch-B) hence each connection is repeated for A and B switches [21].

The commercial Ethernet uses collision detection methods. Collided packets get dropped and retransmitted later which make the communication indeterministic and unreliable[1]. In AFDX, due to the Virtual Link (VL) structure and the fact that the link between elements is full duplex, there are no packet collisions [22].

The data packets are generated in each subsystem and they are forwarded into the network by source end-systems. The switch directs the packet to the intended destination end-system(s) when it received it and that completes an information interchange between multiple avionic subsystems [13].

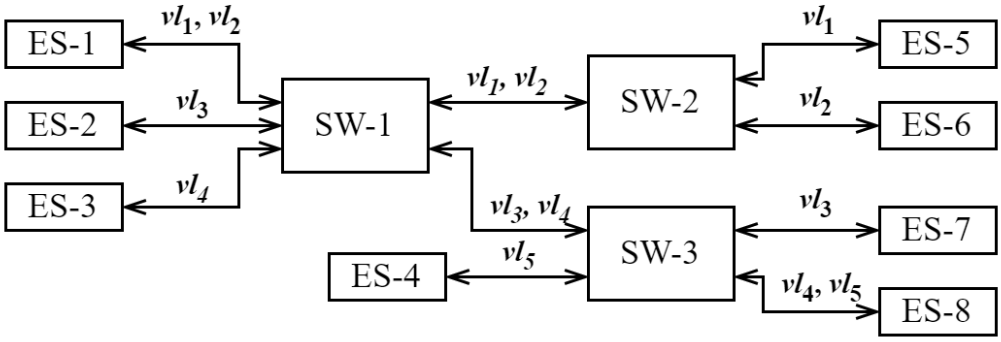

Figure 2.1 Example AFDX Network

2.1.1. AFDX Frame

AFDX frame format is based on standard Ethernet, IEEE 802.3 Standard. Frame fields and lengths of each are given in Table 2.1. Thus, an AFDX message length (L_i) and the total frame length (S_i) which is composed of L_i and 20B of PHY (Physical Layer Device) overheads for a VL_i can take values between the ones shown in (2.1) and (2.2).

$$L^{min}{}_i = 64B, \qquad L^{max}{}_i = 1518B \tag{2.1}$$

$$S^{min}{}_i = 84B, \qquad S^{max}{}_i = 1538B \tag{2.2}$$

The "Sequence Number" (SN) is incremented for each virtual link and used in redundancy management. In an AFDX network, one or many end-systems can be part of a host equipment [4]. This host equipment is identified with "Network ID" and "Equipment ID" fields defined in source MAC address which is shown in Table 2.2. Each end-system can contain one or many partitions which are identified by "Partition ID" (Table 2.4). Additionally, virtual links that are identified by "VL-ID" are stored in the destination MAC address which is shown in Table 2.3.

Table 2.1 AFDX Frame Format

PHY Overhead		**Ethernet Frame [64-1518]**								**PHY Overhead**
Preamble	SFD	MAC Address		Type	Ethernet Payload				FCS	IFG
		Destination	Source		IP Structure	UDP Structure	*AFDX Structure*			
							Payload	*SN*		
7B	1B	*6B*	*6B*	*2B*	*20B*	*8B*	*17B-1471B*	*1B*	*4B*	12B

Table 2.2 Source MAC Address Format

Constant field	**Network ID**		**Equipment ID**		**Interface ID**	**Constant**
0b0000 0010 0000 0000 0000 0000	Const.	Domain ID	Side ID	Location ID	0b001: Network A 0b010: Network B	0b00000
24-bits	4-bits	4-bits	3-bits	5-bits	3-bits	5-bits

Table 2.3 Destination MAC Address Format

Constant field (24 bits)	VL-ID (16 bits)
0bXXXX XX11 XXXX XXXX XXXX XXXX XXXX XXXX	user defined

Table 2.4 IP Addressing Format

Class A	Private IP address	User Defined ID	Partition ID	
1-bit	7-bits	16-bits	(Spare fields) 3-bits	5-bits

2.1.2. Virtual Link (VL) and BAG Concepts

A virtual link is a one-to-many static path between end-systems [23]. Therefore, only one partition can be the source of a VL. VLs are identified by VL-IDs. During the design phase of an AFDX network, VL-IDs and their dedicated bandwidths are allocated and cannot be changed in runtime [24]. Virtual link concept enables an end-system to have the ability of isolating different nodes logically from each other [25]. Thanks to this concept, bandwidth utilization of a VL by one partition won't be affecting other VLs. [4]. VL owes this ability to two parameters. Bandwidth Allocation Gap (BAG) and maximum allowed frame size [26].

VLs can be characterized by dedicated Bandwidth Allocation Gap (BAG) values. BAG is the minimum time slot (in milliseconds) between successive packets. It can be valued as defined in (2.3). It is not only binding for all partitions sharing the same VL-ID but also it establishes a period for a VL itself [21].

$$BAG = 2^k \ [in\ ms],\ k \in Z, 0 \leq k \leq 7\ [4] \tag{2.3}$$

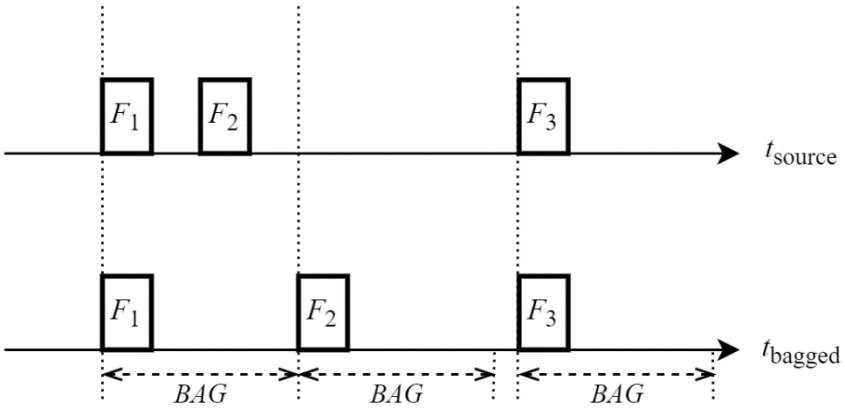

Figure 2.2 Regulated and Unregulated Flow (BAG)

2.1.3. End-System

An end-system is an interface between the avionics subsystem and AFDX the network. End-systems are inputs and outputs of the network and they act as receivers and transmitters for subsystems connected to them.

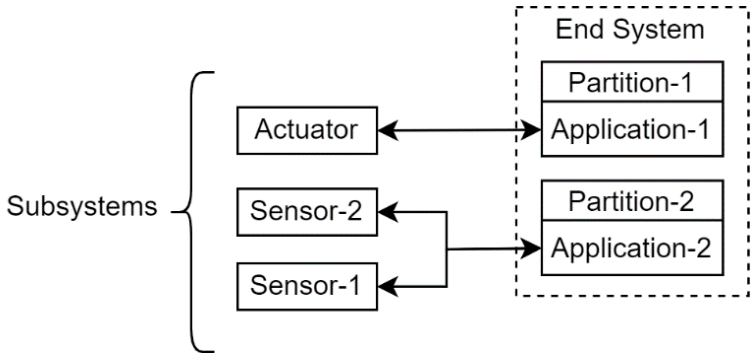

Figure 2.3 Partitions in End-System

Different applications that are running in a subsystem with certain time interval and dedicated memory, can be named as partitions [24]. One or more partitions within the same subsystem can be connected to one end-system (Figure 2.3). They might either be using the same or different VL-IDs. If there are multiple partitions connected to the same end-system, then multiplexing is needed to serialize frames coming from various sources i.e., partitions.

In addition to the multiplexing, frames of each VL are exposed to a regulation according to BAG values that are assigned to them (VLs). As a result of solely this regulation, the time slot between two successive frames of each VL will take at least a BAG amount of time. The main intention behind this concept is to restrict instantaneous frame rates per VL basis.

These two behaviors come together and create the scheduler. Due to the scheduling, frames of each VL will be showing up in a bounded time interval which is called maximum admissible jitter. Traffic flow itself does not cause this jitter but scheduling does [4].

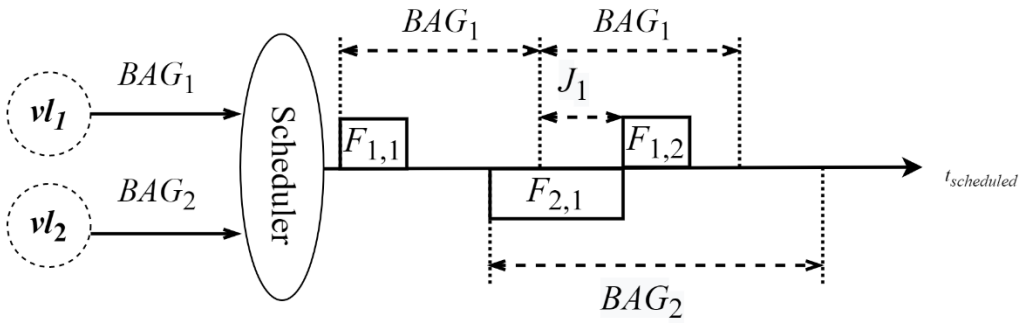

Figure 2.4 End-System Scheduling

Scheduled packets have one final stop before leaving the end-system: redundancy manager. Unless stated otherwise, each frame is sent across both A and B networks [4]. All packets are directed to the receiving end by passing through both networks. Once they are arrived, they are passed through an integrity checker and then finally redundant frames are eliminated within redundancy management [1].

In the transmitting end, the sequence number in the AFDX frame (Table 2.1) is incremented by one and it is wrapped-around to 1 when reached to 255 [27]. On the other hand, at the receiving end, the SN is checked as if it satisfies the equation in (2.4) where PSN indicates the Previous Sequence Number.

$$PSN + 1 \leq SN \leq PSN + 2 \text{ [4]} \qquad (2.4)$$

To identify the redundant frames, skew between switch-A and switch-B is checked as it should not exceed a certain “skewMax” value which is defined in design phase. If the time difference between two frames having the same sequence number (which means one of them is redundant) is less than “skewMax” value, then the later one will be discarded. Otherwise, later frame will be considered as a new one and accepted. [28]

2.1.3.1 Performance Metrics at the End-System

2.1.3.1.1 Jitter

In a transmitting end-system, frames appear at the output of the scheduler in a bounded time window. This window is called “Maximum Admissible Jitter” (J_n^{max}) and it is introduced by the traffic shaper i.e., scheduler. Jitter measurement starts at the beginning of the BAG interval and ends at the very first bit of the frame getting transmitted in that BAG slot. Maximum admissible jitter is actually the total jitter that can happen to the most unfortunate frame. Hence it is calculated from the perspective of the frame at the end of the

line. It is composed of fixed technological latency which can be maximum $40\ \mu s$, plus the amount of time that spent until all previous frames left the physical line i.e., sum of contention delays (d_i) for each frame i. It is limited to $500\mu s$ by the standard in [4]. In the light of these information, it can be calculated by equation (2.6) where V_n denotes all the VL-IDs scheduled by ES_n and C is the data rate of the physical line in bit per seconds (bps) [29].

$$d_i = \frac{(20 + L_i) \times 8}{C} = \frac{S_i\ in\ bits}{C} \tag{2.5}$$

$$J_n^{max} \leq \min\left(500\mu s, 40\mu s + \frac{\sum_{i \in V_n}(20 + L_i^{max}) \times 8}{C}\right) \tag{2.6}$$

2.1.3.1.2 Latency in Transmission

Transmission latency is the overall time spent by a frame until it leaves the end-system. Let t_0 be the time when the last bit of a frame leaves its host partition and $t_1$be the time when that last bit of the frame is transmitted on the physical line. In that case, transmission latency would be the time difference between t_0 and t_1.

Transmission latency (L_n^{Tx}) at the ES_n is given in the equation (2.7) and it can be expressed as the sum of technological latency(TL_n^{Tx}) and configuration latency (TC_n^{Tx}) at the ES_n.

$$L_n^{Tx} = TL_n^{Tx} + TC_n^{Tx} \tag{2.7}$$

Technological latency is the time required to accept, process and begin to transmit the frame at the host partition. It is measured when end-system is not performing any other task hence it is independent of the traffic load. It is represented as a fixed hardware specific delay plus the time taken to transmit a frame to the physical layer i.e., contention delay defined in (2.5) and it is bounded by the standard [4] as in the equation (2.8).

$$TL_n^{Tx} \leq 150\mu s + d_n^{Tx}\ (2.5) \tag{2.8}$$

Configuration latency depends on the traffic and system configuration and it is basically arising due to the traffic shaping i.e., BAG. It depends on the maximum admissible jitter (2.6), the number of frames already in the queue and waiting to be sent and the BAG value of each. It can be expressed by the equation (2.9) for a VL_i at the ES_n and assuming there are p frames to be processed.

$$TC_{n,i,p}^{T_x} = p \times BAG_i + J_n^{max} \tag{2.9}$$

2.1.3.1.3 Latency in Reception

Reception latency is the overall time spent by a frame until it arrives at the target partition. Let t_0 be the time when the last bit of the frame leaves the physical media to enter receiving end-system and $t_1$be the time when the last bit of a frame enters at the target partition. In that case, reception latency would bet the time difference between t_0 and t_1.

Reception latency (L_n^{Rx}) at the ES_n is equal to technological latency (TL_n^{Rx}). It is denoted as (2.10) and bounded in the standard [4] as given in equation (2.11).

$$L_n^{Rx} = TL_n^{Rx} \quad (2.10)$$

$$TL_n^{Rx} \leq 150\mu s + d_n^{Rx} \text{ (2.5)} \quad (2.11)$$

2.1.4. Switch

AFDX nodes are usually combined in a star topology. AFDX switches can be in contact with up to twenty-four nodes [30]. These contacts might be either with an end-system or another switch, but it is a one-to-one connection. Switches use configuration/routing tables to relate port IDs with VL-IDs and route relevant frames through interested destination ports [24]. An AFDX switch is responsible for traffic policing, frame filtering and switching.

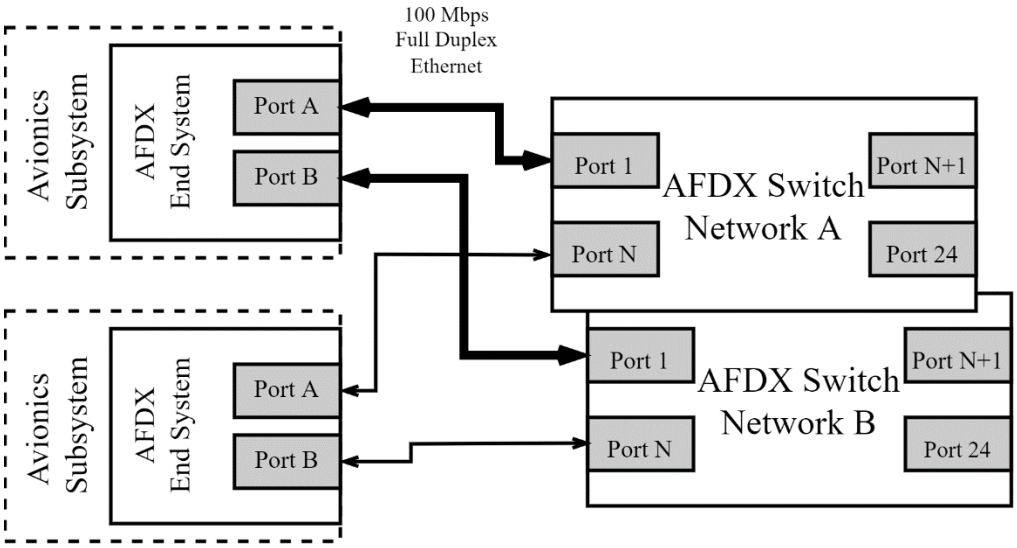

Figure 2.5 Switch – End-System Connections

Frame filtering is used to eliminate invalid frames. It checks frames integrity which means the validity of FCS field in Table 2.1, frame size which should not exceed certain limits ($[S_{min}, S_{max}]\ or\ [L_{min}, L_{max}]$) and frame path which is required to be a valid,

permitted VL-ID. Traffic Policing is used to keep bandwidth bounded and the token-bucket algorithm ensures that. The details about the token-bucket algorithm is discussed in chapter 2.1.4.1. After filtering and policing, frames are classified according to their priorities that can either be HIGH or LOW. Then they are forwarded to the related output ports according to the VL-port mappings stated in the configuration table. Finally, frames leave the switch if the line is not busy or stored in queues until line becomes available. Here, a queueing latency may step-in. This and other latencies are discussed in the next chapter.

2.1.4.1 Token Bucket Algorithm and Switch Jitter

AFDX Standard [4] proposes a token-bucket algorithm for traffic policing. In this algorithm, AFDX switch keeps an account $AC_i(t)$ in bytes for each VL_i (2.12). The credit AC_i is bounded by sigma (σ_i). In time t some credit that is proportional to rho (ρ_i is earned (2.14) where ρ_i is the allowed average data stream rate on VL_i for a BAG_i window (2.13). With every message passing, some credit is consumed with a certain amount which depends on the traffic policing type; byte-based or frame-based. The Equation (2.15) shows the frame-based policing, Equation (2.16) shows the byte-based policing. In case there aren't enough credits, the frame in subject will be dropped [31].

$$\sigma_i = S_i^{max} \times (1 + \frac{J_i^{switch}}{BAG_i}) \tag{2.12}$$

$$\rho_i = \frac{S_i^{max}}{BAG_i} \tag{2.13}$$

$$AC_i = AC_i + \rho_i \times t \tag{2.14}$$

$$AC_i = AC_i - S_i^{max} \ \ if \ \ AC_i > S_i^{max} \tag{2.15}$$

$$AC_i = AC_i - S_i \ \ if \ \ AC_i > S_i \tag{2.16}$$

The tricky part in token-bucket algorithm is the switching jitter ($J_i^{switch} \ \forall \ VL_i$). It is described as the time window that a frame is guaranteed to be placed in and it can be related to the previous frames. When the time difference between successive frames is constant i.e., jitter is zero, the maximum credit value (AC_i) can raise up to S_i^{max} (Figure 2.6). However, in a non-zero jitter case, the maximum credit will always be greater than S_i^{max}. This will enable the token-bucket algorithm to handle a moment when a frame comes early due to this jitter (Figure 2.7).

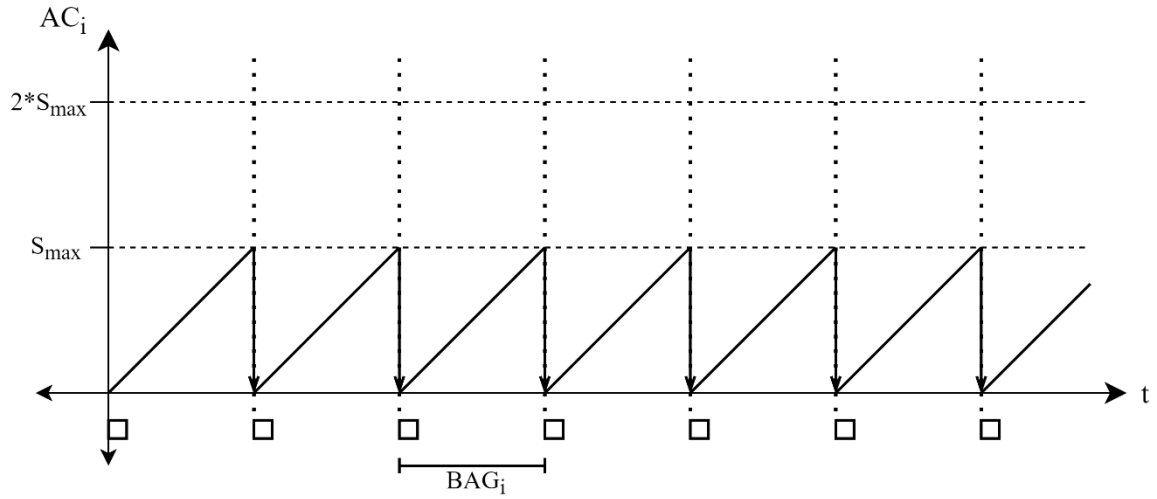

Figure 2.6 Traffic at Traffic Policing when Jitter = 0

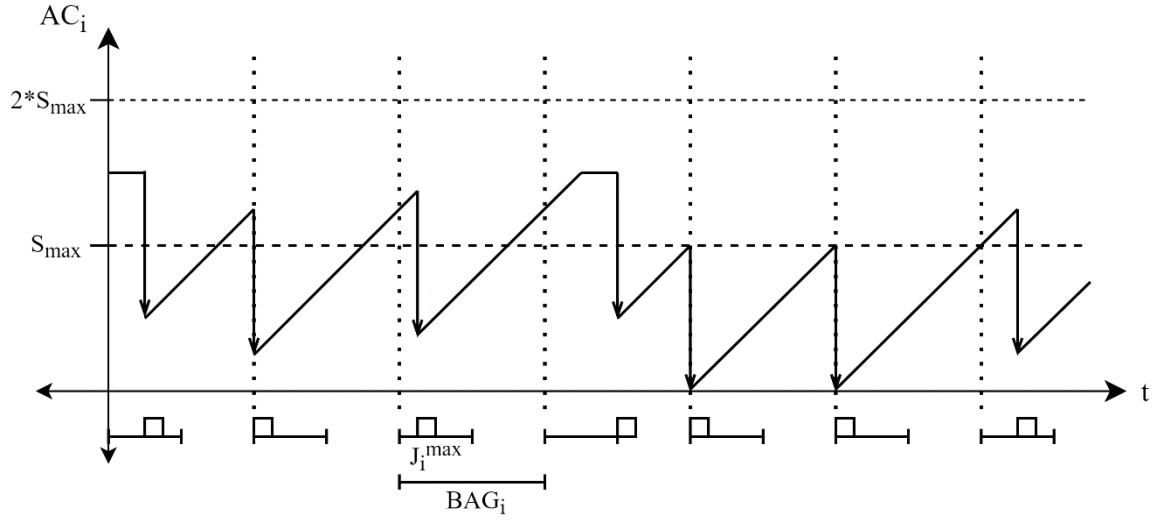

Figure 2.7 Traffic at Traffic Policing when Jitter ≠ 0

2.1.4.2 Latencies in an AFDX Switch

During the journey of a frame throughout the switch, it faces certain latencies such as queueing latency, technological latency, frame transmission time and inter-frame gap. Technological latency (T_{sw}) is due to transmission times in switch fabric, it is related with the hardware and bounded with $100\ \mu sec$. AFDX switches have output buffers for each output port and zero input buffers [32]. When there are multiple frames directed to the same port, queueing latency (T_Q) emerges due to this loading. The time required to transmit the frame on the medium is the frame transmission time finally, the inter-frame gap ($T_{min_{gap}}$) is the minimum slot that must remain between two successive frames. It is in seconds and evaluated as 12B at C bps.

2.1.5. End-To-End Delay

End-to-end delay is a quite important metric when working with AFDX or any other avionics protocol. It consists of the sum of latencies in end-systems and switches and transmission times in between [33]. On the other hand, this delay defines the total amount of delay that a frame will face and it is used when setting up an avionics architecture. Hence it is crucial to know end-to-end delays before realizing an avionics network.

2.1.6. Little's Law

Little's law is a quite simple mathematical formula and yet its paper [34] is one of the most cited papers ever [35]. In this paper it is proven that, for a queueing system in the equilibrium, the average number of items (L) must be equal to the average wait time (W) in the queue times the average arrival rate into the queue (λ). The Little's Law is given in equation (2.17). It can be applied to any queueing systems of all kinds such as people waiting for a coffee in a queue, products in a manufacturing line or messages queued in a network. The only assumption for this law is what goes in must come out.

$$L = W * \lambda \tag{2.17}$$

This law is considered to be important for this thesis because it will be used to evaluate some experimental results when verifying the model.

2.1.7. Confidence Interval

Confidence interval (CI) can be described as a range for estimates of a certain parameter that is unknown. It is computed to estimate the population mean based on the sample mean and a designated confidence level is specified when computing it such as 95% or 99% [36].

When modeling a system in a simulation environment to have an opinion about the behavior in advance, it is crucial to collect enough data and obtain results that reflect the real application. It is not likely to get true mean (μ) since simulation cannot be run forever. However, it is possible to say that the sample mean y_n is within %Δ band of the true mean μ with a confidence level of g% where n goes to infinity. Confidence g is calculated by equation (2.18 where S_n is the standard deviation for n samples, t_g is a constant that can be selected as 1.96 or 2.58 for designated confidence levels 95% or 99% respectively. The network configuration and analysis tool (ANCAT) proposed in this thesis in further chapters is able to calculate CI with both levels and uses it to justify its computations.

$$\%\Delta = \frac{S_n \times t_g}{y_n \times \sqrt{n}} \tag{2.18}$$

3. PREVIOUS WORK

3.1. OMNeT++ and Other Network Simulation Tools

OMNeT++ is a powerful discrete event simulation environment for modeling communications networks of numerous different domains. It is intended to be used for research purposes mostly hence under APL license it is free to use for non-profit users. Instead of providing major simulation components OMNeT++, provides basic tools to write a functioning simulation [37]. For many specific areas, OMNeT++ community developed frameworks/packages containing certain models of popular protocols. For example, the INET framework contains comprehensive models of the internet that are handling the protocol from physical to application layers and for both wired and wireless networks [17]. Whereas Mobility Framework includes implementation of some ad-hoc network models [38].

Providing a whole simulation ecosystem is not only a key feature of OMNeT++ but this is what distinguishes OMNeT++ from other network simulators such as NS [37]. In addition to that, OMNeT++ presents a hierarchical structure with a modular architecture and different user interface options. Most importantly it offers all these features with a lower complexity when compared to its peers [12].

To create simulations in OMNeT++, C++ and NED (NEtwork Description) languages are used. With `*.ned` files, the structure of the model is established. Each block in the network, connections between them and even the network itself are defined by so called simple and compound modules in a hierarchical order. In addition to that, configuration parameters can be defined and evaluated with default values in `*.ned` files. While `*.ned` files are providing a huge flexibility to OMNeT++ when defining the topology, in another network simulation tool OPNET, the models always use a fixed topology [39]. OMNeT++ provides class libraries in C++ and it is also used to define the functional behavior of the blocks when simulation is running. When OMNeT++ is compared to NS-2 and OPNET is terms of simulation libraries, NS-2 has less built-in functions and OPNET simulation library is in C instead of C++ which is more modular an modern which makes OMNeT++ more powerful than both in terms of libraries [39].

Furthermore, it is possible to interfere in the simulation behavior by using `*.ini` files. The `*.ini` files are not needed to be compiled and they can be used to re-evaluate the parameters that are defined in the `*.ned` file. Thus, it is possible to set up a simulation and compile it once, then change the configuration by just modifying the `*.ini` file. NS-2 and NS-3 also rely on C++ for simulation behavior [11]. But in NS-2 `oTcl` scripts are used to control the simulation and specify other aspects such as topology. `oTcl` scripting was preferred to reduce recompilations and save time in the past but that design choice influences simulation performance negatively [40]. That is why the `oTcl` scripts left their places to python scripts in NS-3 [12].

OMNeT++ provides a much more integrable, reusable and flexible architecture with less complexity when compared to the other open-source network simulation tools like NS-2 and NS-3 [12], [13]. It is open-source and free for non-profit users instead of OPNET [37]. It is constantly getting updates, properly documented and has a big community that is growing and contributing. It has a very good and easier to follow graphical interface when compared to NS-2, NS-3 and OPNET [16]. Due to all these reasons, OMNeT++ is preferred over other network simulation tools.

3.2. Other AFDX Simulation Models

Over the years, AFDX has been simulated with different tools such as NS-2, OPNET, Net2Plan, MATLAB/Simulink, QNAP2 (Queueing Network Analysis Package) and OMNeT++ in a multitude of works. In these previous works, AFDX is modeled with different tools by mimicking essential behaviors with the blocks at hand. Some essential metrics are recorded and results are compared with either realistic or theoretical results in order to authenticate the simulations.

The simulation in [41] is one of the oldest among all examples. In this work, QNAP2 which is a modelling environment that facilitates building, solving and handling queueing problems [42], is used to model an AFDX network. To simulate the behavior of different AFDX blocks, specialized queues are used and simulation results are validated by comparing them with the results calculated with Network Calculus [43]. Although it is theoretically verified, the simulation is run with only one topology which has only one switch.

There are several models that are established with NS-2 such as [9] and [8]. In these works, atomic AFDX behaviors are modeled with built-in ethernet elements. In [9], a network with two switches and several end-systems is investigated in terms of end-to-end delay and jitter where similar experiments are diversified with data flows of various priorities under different scheduling strategies in [8].

One of the most popular network simulators, OPNET is used to simulate AFDX as well. AFDX is modeled by using built-in OPNET blocks in [10], [27] and [25]. A topology including eight switches and nine end-systems which has been examined before in another paper [10]. In a similar fashion, the network in the [27] is also selected from a previous work that was examining a real-life scenario. In both works, the simulation is verified by comparing results with mentioned previous works. On the other hand, in [25], the effect of redundancy is examined by comparing a non-redundant network with a dual redundant one.

The work in [31] is unusual than the other works because real-time data is used to feed the simulation in that network. The main purpose of the proposed model is to verify some aircraft system functions without constructing an expensive AFDX network in hardware. For that purpose, an AFDX model is established in MATLAB/Simulink with modified built-in blocks and with the help of some additional physical tools, an AFDX network that is composed of real and simulated elements is constructed. Simulation is verified and benchmarked with three different scenarios.

In another paper [44], AFDX performance is evaluated over a model created in an open-source tool called Net2Plan. Net2Plan is used to plan and optimize networks and besides obtaining actual values with simulation, it is able to calculate worst case end-to-end latencies via Network calculus and trajectory approach [45]. In this paper, experiment is conducted for a complex network with eight switches and more than 70 LRUs.

Finally, OMNeT++ is used several times to simulate AFDX before OMNeT++ model [19] is published. Both [16] and [13] have benefited from the ease of using OMNeT++ INET model [17]. Due to the fact that AFDX and TTEthernet have a lot in common, the TTEthernet library of INET is used to create an AFDX simulation in [16] and this model is verified by comparing the results with previous works in [46] and [47]. On

the other hand, in [13] the AFDX model that is already verified and presented in [16] is used and simulation results are investigated.

3.3. OMNeT++ AFDX Simulations

AFDX is modeled with an OMNeT++-like program before [19] and improved in another thesis for the sake of another work [48]. Before discussing the contributions and development carried out over this model in the scope of this thesis, the original model and previous works will be explained in this chapter.

3.3.1. OMNEST Model

The first AFDX model was built by OMNEST team for one of their clients [18]. OMNEST is the commercial version of OMNET++ and it is open source as well. Apart from licensing, support and some other small features, they are nearly identical and can be used as substitutes for each other. When building this model, the OMNEST team considered using the blocks/components that are already available such as INET [17] and `queueinglib` [49]. Since Ethernet needs of AFDX are pretty simple, using INET would bring out a lot of questions that they are not interested in answering. To avoid additional complexity and possible performance issues, Ethernet and other higher protocol layers such as IP, TCP and UDP are not implemented. On the other hand, `queueinglib` was highly beneficial and it is used to model different functionalities in AFDX model. These uses are investigated in the following paragraphs.

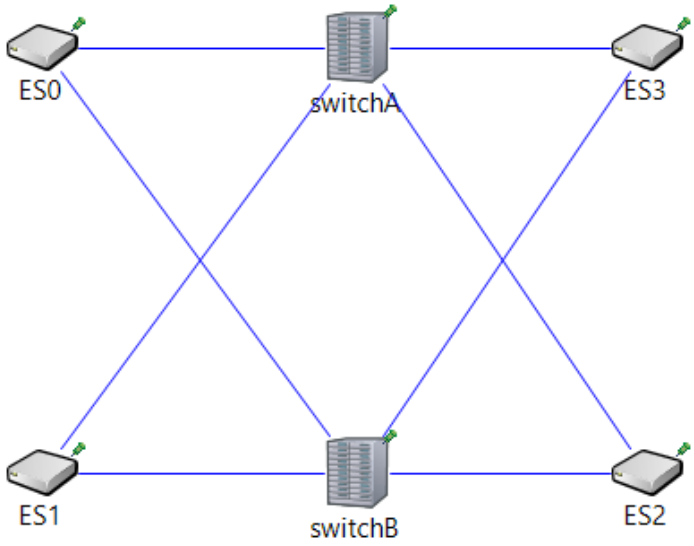

Figure 3.1 Example AFDX Network

An example AFDX network setup is given in Figure 3.1. Two main components of an AFDX network i.e., end-system and switch, are modeled with compound modules. The end-system module is responsible for message creation, BAG regulation, queueing and multiplexing, redundancy management and integrity checking. Whereas the switch module is responsible for frame filtering, traffic policing, queueing (at the transmitting end only), priority classifying, scheduling and routing. The OMNEST team mentioned some future work in the project summary [18]. In short, even though there are some unimplemented features in this model, it is highly comprehensive and useful.

In an OMNeT++ network, messages floating between blocks can either be in raw types such as `cMessage` or `cPacket` or complex types that are derived from them, such as `Job`. Both `cMessage` and `cPacket` contains variables that a network message expected to have such as, length, ID, creation time, priority, classification, type etc. where additionally `cPacket` messages consume time when transmitting through Ethernet lines.

In this OMNEST AFDX Model, `queueinglib` is used as an auxiliary library. Today, `queueinglib` is under version control but the version that is used in OMNEST AFDX model was an intermediate release and thus, not published. In the `queueinglib` version used in this model, all blocks send and receive messages in type `Job` that is derived from the raw type `cPacket`.

AFDXMessage.msg

```
27 message AFDXMessage extends queueing::Job
28 {
29     int networkId;
30     int equipmentId;
31     int interfaceId;
32     int virtualLinkId;
33     int seqNum;
34     int partitionId;
35     int udpSrcPort;
36     int udpDestPort;
37 }
```

Figure 3.2 AFDX Message Format

In OMNEST AFDX model, a message type called `AFDXMessage` (Figure 3.2) that is derived from `Job` of `queueinglib` is used. This message type contains AFDX specific fields such as source MAC address fields (network ID, Equipment ID, interface ID), destination MAC address fields (virtual link ID), UDP Structure fields (source and

destination port numbers), IP Structure fields (partition ID) and sequence number that are described in Table 2.1

3.3.1.1 End-System

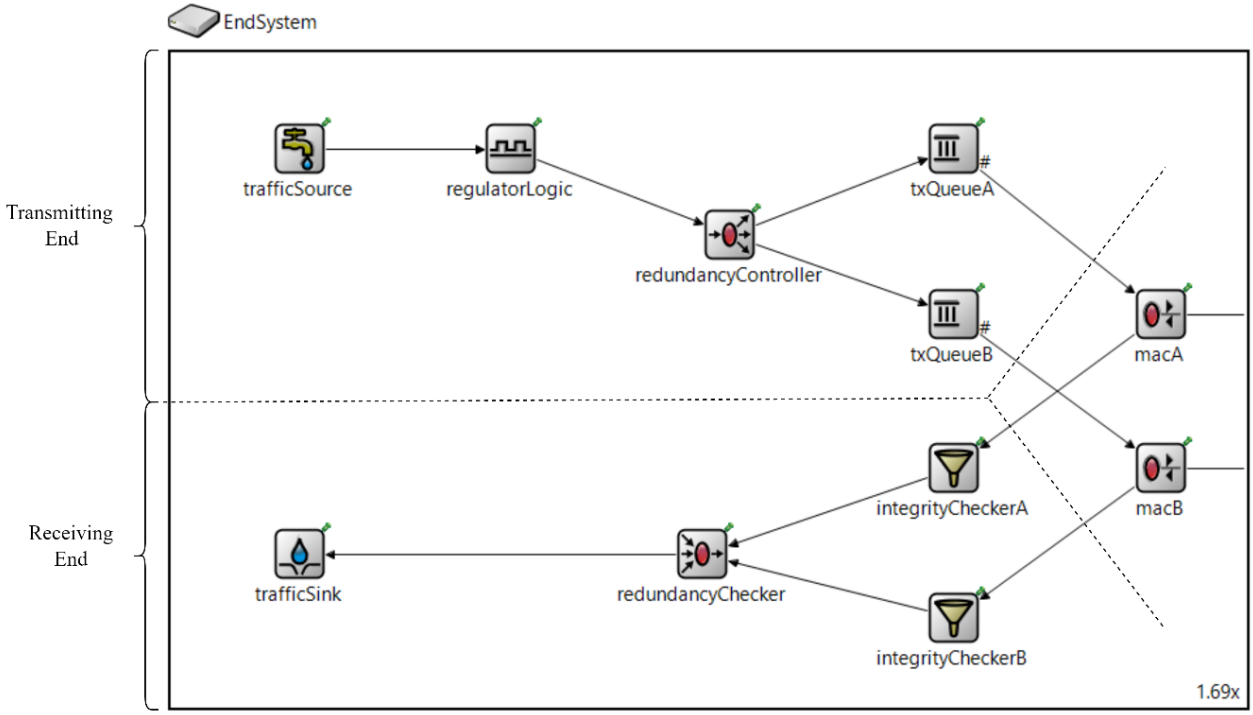

Figure 3.3 End-System Compound Module

End-system compound module is connected to the Ethernet line by two bidirectional ports, one is for network A and the other is for network B. It can be logically divided into two: receiving end and transmitting end (Figure 3.3). Transmitting end involves message creation, scheduling, redundancy management and transmitting MAC operations, where receiving end handles integrity and redundancy checking and receiving MAC operations.

The performance metrics that are explained in 2.1.3.1 or in other words, delaying elements in an end-system, are modeled in this model as well, but not entirely. Configuration latency (2.9) and technological latency (2.8) that are forming the transmission latency (2.7) are only partially handled. In terms of configuration latency, the jitter that emerges due to the scheduling (2.6) is introduced by `txQueue-MAC` blocks (3.3.1.1.1.4). But since BAG regulation (3.3.1.1.1.2) is missing in this model, configuration latency is inadequate. In terms of technological latency, the constant part of the equation that denotes the load-free hardware-dependent technological delay is not handled. Finally, the contention/transmission delay that is caused by previously sent

frames is simulated by the transmission line itself (that mimics ethernet cable) with combination of `txQueue-MAC` blocks (3.3.1.1.1.4).

Generally, the blocks in this end-system model are not VL-aware because this model doesn't support multiple traffic sources. Hence, only one VL type is transmitted through an end-system in each run.

3.3.1.1.1 Transmitting End

3.3.1.1.1.1 Traffic Source

`trafficSource` block is inherited from the `Source` block of the `queueinglib` library. A `Source` block is responsible for creating `Job`s by the specified rate, with pre-defined inter-arrival times and until a certain time or number of events is reached. These specified values are configuration parameters of this block (Table 3.1) and they must be specified either in `*.ned` or `*.ini` files. Note that these parameters are generic parameters for traffic specification. A `trafficSource` creates messages just like `Source` does but its output is `AFDXMessage` instead of `Job`. Thus in addition to the values listed in Table 3.1, it needs AFDX frame fields (Table 2.1) to be defined in `*.ini` or `*.ned` file in order to create the message. `trafficSource` doesn't discriminate against VL-IDs. It creates and schedules new `AFDXMessage`s with the values specified in configuration in a superficial order.

Table 3.1 Traffic Source Configuration Parameters

Parameters	Definition
interArrivalTime(s)	Time difference between successive messages
startTime(s)	Time that indicates the creation of the first message
stopTime(s)	Time that indicates the creation of the last message
jobCounter	Maximum number of messages to be created

Figure 3.4 shows the `Source.ned` file `parameters` section that contains parameter definitions with default values. Figure 3.5 shows the actual value assignments of those configuration parameters consisting of `AFDXMessage` fields (Figure 3.2) and `Job` fields i.e., `jobPriority` and `jobKind`.

```
simple Source
{
    parameters:
        @group(Queueing);
        @display("i=block/source");
        int numJobs = default(-1);
        volatile double interArrivalTime @unit(s);
        string jobName = default("job");
        volatile int jobKind = default(0);
        volatile int jobPriority = default(0);
        double startTime @unit(s) = default(interArrivalTime);
        double stopTime @unit(s) = default(-1s);
    gates:
        output out;
}
```

Figure 3.4 Configuration Parameters in Source ned File

omnetpp.ini

```
1  [General]
2  network = AFDXExampleNetwork
3  **.trafficSource1.interArrivalTime = exponential(1s)
4  **.scheduler.serviceTime = 0s
5  **.trafficSource1.networkId = 0
6  **.trafficSource1.equipmentId = 0
7  **.trafficSource1.interfaceId = 0
8  **.trafficSource1.virtualLinkId = intuniform(0, 3)
9  **.trafficSource1.seqNum = 0
10 **.trafficSource1.partitionId = 0
11 **.trafficSource1.udpSrcPort = 1234
12 **.trafficSource1.udpDestPort = 5678
13 **.trafficSource1.jobKind =  trafficSource1.virtualLinkId
14 **.trafficSource1.jobPriority = intuniform(0, 1)
```

Figure 3.5 Configuration Parameters and AFDX Message Fields in `*.ini` File

3.3.1.1.1.2 Regulator Logic

`RegulatorLogic` is a block that is designed specifically for the AFDX Model. It should be responsible for introducing BAG into frame sequences of each VL-ID separately. But it is not implemented.

If this block would be implemented, the first part of the configuration latency (2.9) that arises from the BAG regulation, should be introduced by this block. Additionally, it might be needed to be VL-aware because BAG values are VL specific and shall be introduced on a per VL-basis.

3.3.1.1.1.3 Redundancy Controller

This block has two duties in terms of redundancy management. First duty is to increment the sequence number and returns to one when it reaches up to 255. Second duty

is to duplicate of each message to be able to send them to both networks A and B. Redundancy shall be enabled/disabled by the configuration[4]. Thus, this block has some configuration parameters (Table 3.2.) to satisfy that requirement. `RedundancyController` is not VL-aware and does not introduce any delay.

Table 3.2 Redundancy Controller Configuration Parameters

Parameters	Definition
`copyToLinkA` (bool)	Enables network A
`copyToLinkB` (bool)	Enables network B

3.3.1.1.1.4 Tx Queue and MAC

`txQueue` is an object of type `PassiveQueue` from the `queueinglib`. When a `PassiveQueue` receives a message, it looks for an idle `Server` among its connections to direct the message. If one or many servers are available, it selects one and sends the received message. If there aren't any selectable servers i.e., all servers are busy with transmitting previous messages, the received message gets pushed to the queue. If a connected server makes a pull request before a new message reception, the pushed message gets popped and sent to the owner of the request.

`MAC` is inherited from the `Server` of the `queueinglib` library. Due to the inheritance, `MAC` can interact with `PassiveQueue` objects as a `Server`. In addition to that, it handles the interaction with the physical layer. When an "idle" server receives a message from the end-system input, it changes its state from "idle" to "reserved", it sends the message through the Ethernet output port to the physical line if the line is not busy, it waits for an additional IFG time, then changes its state to "idle" again but if the previous message transmission was not concluded yet, simulation stops with an error. Since a server changes its state to "reserved" from the beginning of a transmission until it is concluded successfully, this shouldn't happen in the best practice. If the line is busy, messages are kept in the queue until it becomes available again and thus, a frame-transmission/contention delay will be introduced. This delay is also explained in chapter 2.1.3.1.2 as a part of the technological latency equation (2.8).

These two blocks are working together to fulfil the multiplexing-part of the scheduling mission of an end-system. VL-IDs are not important for `txQueue-MAC` pair because at this point, all frames will be in tandem and must be treated equally. If the ability

of adding multiple traffic sources was implemented for the cases similar to the one shown in the Figure 2.4, incoming parallel frames would be queued by and sent over one by one, by the `txQueue-MAC` pair but that is also not implemented.

3.3.1.1.2 Receiving End

3.3.1.1.2.1 MAC

The duties of the MAC block are much simpler in the receiving-end. If a message is received from the Ethernet input, it gets directed to the end-system by the related port directly. This block doesn't introduce any delay and is not VL-aware.

3.3.1.1.2.2 Integrity Checker

In an AFDX end-system, integrity check shall be done by using the Equation (2.4). The `integrityChecker` in this simulation, examines the sequence number, compares the SN of the received frame with zero, $(PSN + 1)$ and $(PSN + 2)$ and decides whether it is appropriate or not. This block should be VL-aware but it is not. This may cause a mismatch when checking previous sequence numbers. This block doesn't introduce any delay.

3.3.1.1.2.3 Redundancy Checker

This block is responsible for eliminating redundant frames. It has two input ports: one from network A and one from network B. Sequence numbers of successive frames must be increasing all the time. So, if two successive frames have the same sequence numbers, one of them must be the redundant copy and thus it is dropped. This is how the `redundancyChecker` eliminates redundant frames. In addition to that, there is a time control with "skewMax" constant. If sequence number is not incremented as expected but the time difference between the current and the last frames is larger than the "skewMax", then this frame is not treated as a redundant frame and sent over. This block should be VL-aware but it is not. This may cause a mismatch when checking the time difference between successive frames. This block doesn't introduce any delay.

3.3.1.2 Switch

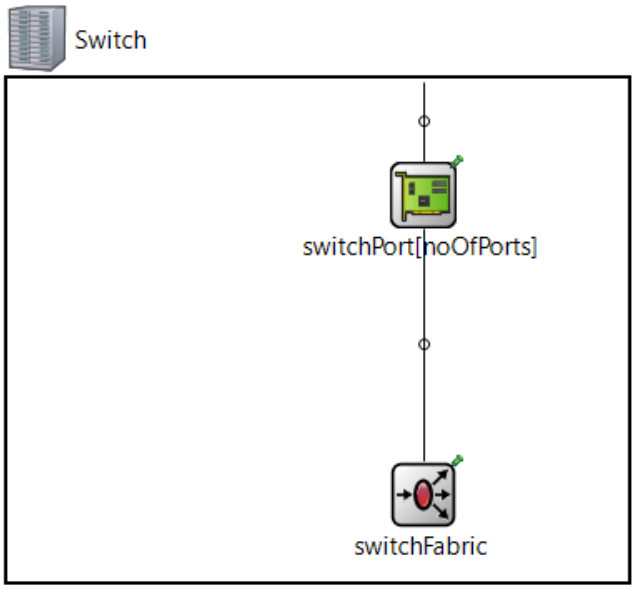

Figure 3.6 Switch Compound Module – Open Form

The Switch module is composed of two main components (Figure 3.6): `SwitchPort` and `SwitchFabric`. There is one `SwitchPort` block for each actual port where there can be up to `noOfPorts` amounts of ports but a typical AFDX switch is expected to have 24 ports. To explain the switch internal structure better, Figure 3.7 is added.

In the switch module of OMNEST AFDX model, the only behavior that VL-awareness is necessary is traffic policing but it is not implemented. Thus, none of the blocks in this module is VL-aware. `txQueue-MAC` blocks are expected to insert queueing delay. But other latencies mentioned in chapter 2.1.4.1 are not modeled in this version such as hardware latency and inter-frame gap.

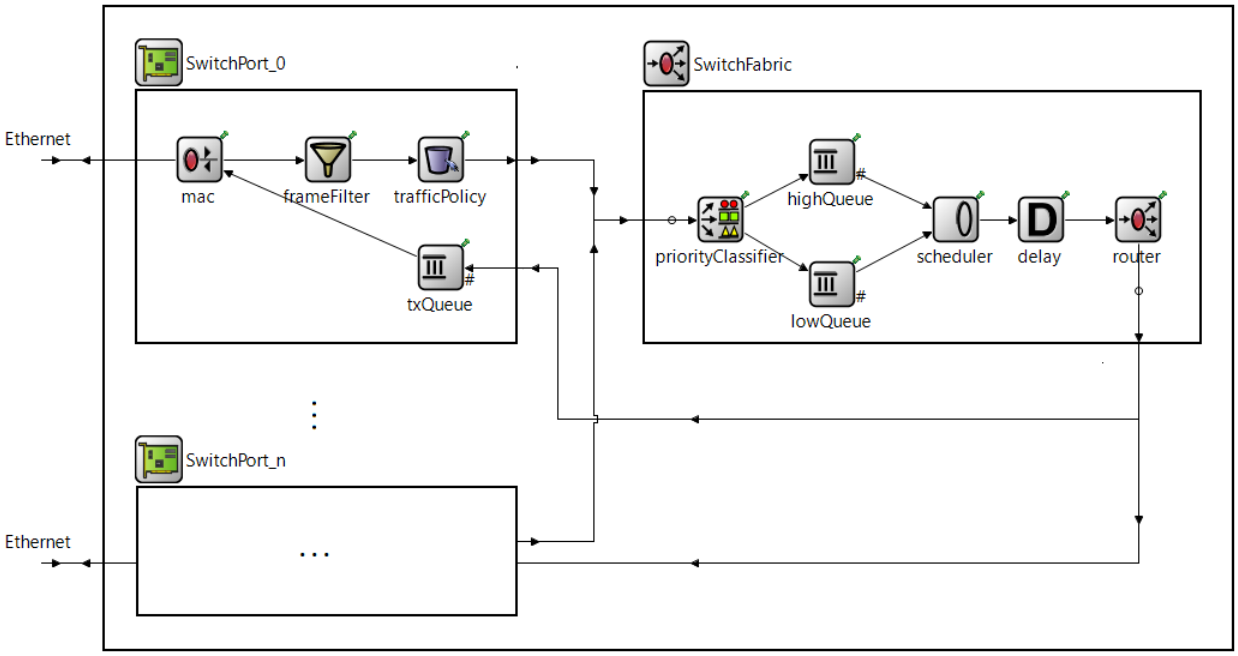

Figure 3.7 Switch Compound Module – Open Form

3.3.1.2.1 Switch Port

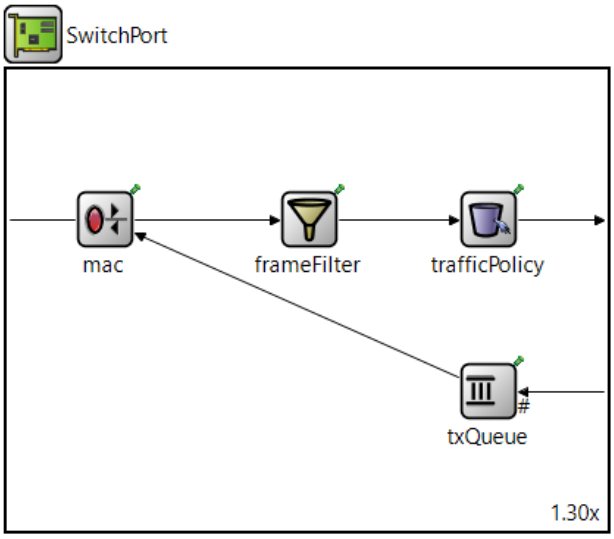

Figure 3.8 Switch Port Compound Module – Open Form

`SwitchPort` compound module is connected to the Ethernet line and `SwitchFabric` by bidirectional ports (Figure 3.7). Switch port handles the interactions with the physical layer, performs frame filtering and traffic policing (Figure 3.8).

3.3.1.2.1.1 TxQueue and mac

These two blocks are the same as the ones mentioned in the end-system (3.3.1.1.1).

3.3.1.2.1.2 Frame Filter

This block is expected to check frames' according to the chapter 2.1.4 to avoid invalid frames to consume credit in later blocks. But it is not implemented in the OMNEST AFDX Model. Hence this block transparently directs the received frames.

3.3.1.2.1.3 Traffic Policy

This block is expected to apply the token bucket algorithm according to chapter 2.1.4 before directing the frames. But it is not implemented in the OMNEST AFDX Model. Hence this block transparently directs the received frames.

3.3.1.2.2 Switch Fabric

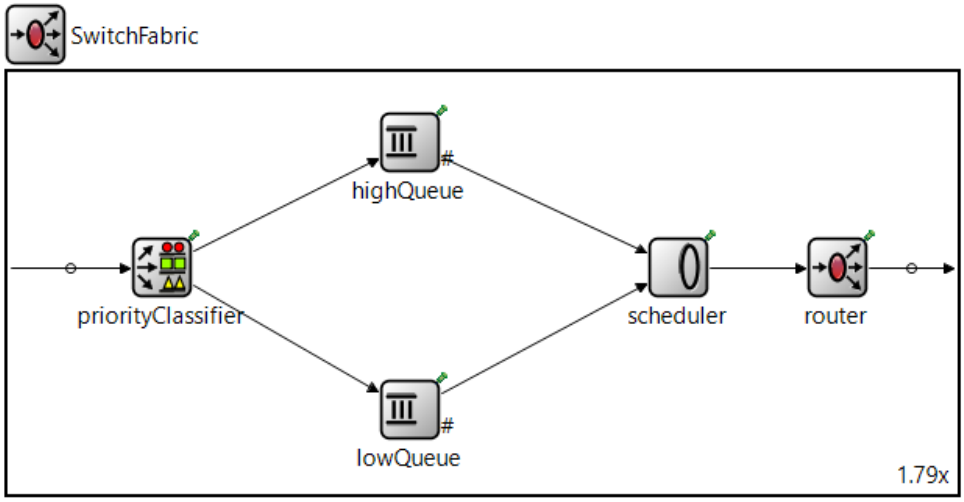

Figure 3.9 Switch Fabric Compound Module – Open Form

`SwitchFabric` compound module has `noOfPort` amount of input and output ports which are connected to each `SwitchPort` block. This block is responsible for priority-based frame classification, scheduling and VL-routing.

3.3.1.2.2.1 Priority Classifier

`priorityClassifier` is a `Classifier` from the `queueinglib`. It evaluates incoming messages by considering the assigned "priority" values and directs them to the appropriate output.

3.3.1.2.2.2 Queues and Scheduler

`scheduler` is a `Server` from the `queueinglib` where `lowQueue` and `highQueue` are `PassiveQueue`s. This server-passive queue pair has the same behavior and responsibility as the ones in the End-System (3.3.1.1.1.4) and Switch Port (3.3.1.2.1.1)

3.3.1.2.2.3 Router

This block is a `VLRouter`. It is expected to read the VL-table and route messages by considering it to the appropriate ports. But it is partially implemented. It doesn't read a configuration table but there is a tiny algorithm that mimics a configuration table and it routes the messages by considering it.

3.3.2. Improvements Over OMNEST Model

OMNEST AFDX model provides a good insight but it has inadequacies. Within the design and evaluation of SQDSR, a new real-time ethernet protocol, this AFDX model is converted into a more realistic one and used for a comparative performance analysis [48]. The improvements made within the scope of the mentioned thesis will be summarized in this chapter.

3.3.2.1 Technological Latencies

Technological latencies are not modeled in the original model although they are quite important when calculating end-system and switch latencies and they contribute to the determinism of the network [4]. In [48], three `queueinglib delay` blocks are added to simulate technological latency in transmitting end-system (2.1.3.1.2), receiving end-system (2.1.3.1.3) and switch (2.1.4.1). Actual delay values are configuration parameters and can be modified by `*.ini` and/or `*.ned` files.

3.3.2.2 Multiple Traffic Source Capability

In a real AFDX network it is very likely for an end-system to have multiple traffic sources. In fact, a multiplexing behavior is incorporated into the nature of AFDX for such a case. Although it was possible to add multiple sources in the previous model, it requires change in the code and thus wasn't a configuration parameter. With the changes in [48], the `trafficSource` object in the end-system compound module is transformed into an array with a configurable size. Thus, both single and multiple traffic sources became available and configurable by the `*.ini/*.ned` file but this contribution comes with a necessity of VL awareness for the blocks that need to distinguish VL-IDs. Due to multiple sources, messages with different VL-IDs can be generated in the same end-system and in some cases such as the time difference between the last frame received matters, each message of different VL-ID must be handled separately. The blocks that are affected by this change are `RegulatorLogic`, `redundancyChecker`, `RedundancyController` and `TrafficPolicy`.

3.3.2.3 BAG Regulation

Although one of the main features of AFDX is BAG regulation as mentioned in chapter 2.1.2, it is not implemented in the original model. In [48], the `handleMessage(..)` function that is called whenever a message is received by the `RegulatorLogic` block is modified and as a result this block gained ability to control the flow by considering BAG values stated in the `*.ini/*.ned` file. When this modification is mixed with the one in chapter 3.3.2.2, a VL-awareness need arises. Because to control the flow, this block keeps the transmission time of previous frames of each VL and won't let new ones to go before at least a BAG time interval is passed.

3.3.2.4 Traffic Policing

AFDX switches, control the bandwidth and shape the traffic with a token bucket algorithm hence traffic policing is very important. But traffic policing functionality is missing in the very first version of AFDX model. In the thesis mentioned [48], the token bucket algorithm is implemented. For that sake, the `handleMessage(..)` function is modified. In the new model, each time a new message is received, message reception time is saved, the time difference between the previous message reception is calculated, already possessed, obtained and spent credits are calculated and a decision is made whether to drop the frame or let it go. Due to the fact that having multiple sources is available now (Chapter 3.3.2.2) all these values are kept in per-VL variables.

3.3.2.5 VL Router

Routing packets according to a routing/VL table was the expected behavior and this was listed as a "to do" in the OMNEST website [18]. As a part of the work in [48], routing tables that used to match VLIDs with switch port numbers are added for each switch to the model and `VLRouter` modified accordingly. With this change, it is possible to specify the message paths without changing the code manually. An example VL-table can be seen in Figure 3.10.

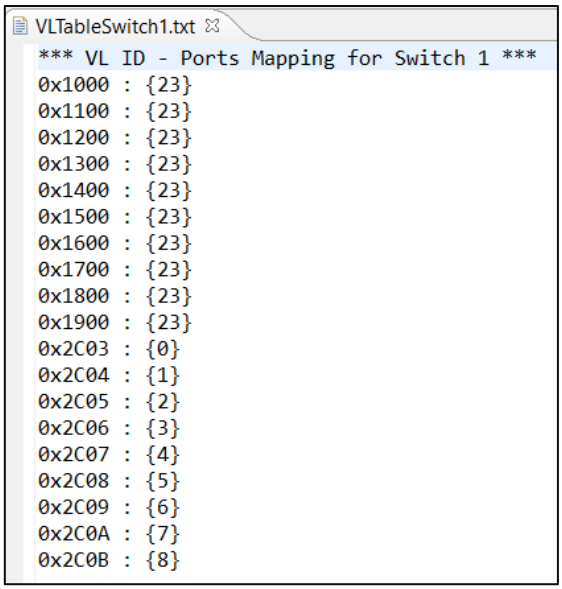

Figure 3.10 Routing Table Example

3.3.2.6 Other Small Changes

Aside from the ones listed below, there are some other small changes. For example, a new parameter called `frameHeaderLength` is added. In the old model, the variable `packetLength` was used to set the length of a message. With this new parameter, summation of both is used to set the message length. This modification enables to set payload length apart from the header length. Additionally, BAG value is added among message parameters and can be set via `*.ini/*.ned` file. Moreover, as a result of the change mentioned in Chapter 3.3.2.2, `redundancyChecker` and `RedundancyController` classes became VL-aware. Because they both need to keep track of sequence numbers and since there are multiple sources, keeping previous sequence numbers in a single variable is not enough. In the scope of this change, the variables that are used to keep previous sequence numbers are changed into arrays.

For the sake of the work in [48], a new class called `NetworkStatistics` is added. It is used to keep necessary measurements and it is called in several places alongside the project such as `PassiveQueue`, `Sink` and `Source` classes in the `queueinglib`.

4. AFDX SIMULATION MODEL

One of the main contributions of this thesis is to develop a better, more realistic, more easily configurable and up to date OMNeT++ AFDX simulation model. The latest AFDX simulation model at hand is developed in [48]. However, since the main concern of that thesis wasn't AFDX and due to the limited time, there were some deficiencies in the represented mode. In addition to that, OMNeT++ and its libraries such as `queueinglib` have faced major updates over time. Therefore, some obligatory and reformative modifications are made to the model at hand.

4.1. New Network Statistics Class

OMNeT++ environment provides users with some functions to keep records of interested metrics. Such as the `record(double value)` function from an internal library `coutvector`. With this function, it is possible to record all of the values that a certain variable takes during the simulation with respect to simulation time. After simulation is finished, it is possible to plot recorded values within the IDE or export in *.csv format.

Changes in the scope of [48], bring to AFDX model a new singleton pattern[50] class called `NetworkStatistics`. This class has some specific functions such as `CollectStatisticsLatency(..)` or `CountGeneratedPackets(..)` which calculates certain statistics by collecting certain values. Although those functions are useful for that work, they are quite application specific.

For the interests of this thesis, `NetworkStatistics` is renewed. The purpose of the modifications in this class is to make record keeping much more generic. Simply, there are two main functions: one is to create a new unique record keeper (`createRecord(recordType, key)`) and the other is to add a new value to the record vector in it (`record(recordType, key, value2Record)`). These functions must be called with an `enum` that defines the record type and a unique `key`. Record types are defined by considering the main metrics to be recorded and given in Figure 4.1. Values with the same unique key and record type will be collected in the same record vector. These two functions use types and functions of the built-in library `coutvector` whose example usage is demonstrated in Figure 4.2.

```
enum RecordType_t
{
    E2E_LATENCY_PER_VL = 0,
    ES_BAG_LATENCY_PER_VL,
    ES_SCHEDULING_LATENCY_PER_VL,
    ES_TOTAL_LATENCY_PER_VL,
    DROPPED_FRAMES_IN_QUEUE_PER_VL,
    DROPPED_FRAMES_TRAFFIC_POLICY_PER_VL,
    TRAFFIC_SOURCE_PER_VL,
    SWITCH_QUEUEING_TIME_PER_SWITCH,
    SWITCH_QUEUE_LENGTH_PER_SWITCH,

    CREDIT_PER_VL_PER_SW,
    E2E_LATENCY_PER_VL_PER_RECEIVER_ES,
    SWITCH_QUEUEING_TIME_PER_VL_PER_SW,
    SWITCH_QUEUEING_TIME_PER_SW_PER_PORT,
    SWITCH_QUEUE_LENGTH_PER_SW_PER_PORT,
    SWITCH_QUEUEING_TIME_PER_SW_PER_VL_PER_PORT,
};
```

Figure 4.1 Record Types in `NetworkStatistics` Class

```
afdx::NetworkStatistics::getInstance()->createRecorder(TRAFFIC_SOURCE_PER_VL,
        afdxMsg->getVirtualLinkId());
afdx::NetworkStatistics::getInstance()->record(TRAFFIC_SOURCE_PER_VL,
        afdxMsg->getVirtualLinkId(),
        afdxMsg->getCreationTime().dbl());
```

Figure 4.2 Call Examples for `NetworkStatistics` Functions

4.2. New Queueing Library

On April 13th, 2022, OMNeT++ 6.0 became available [51]. The `queueinglib` used in the AFDX Model was so old that it wasn't even possible to build the code in the newly released version. Partially due to that difficulty and also due to the fact that a much never version of `queueinglib` was released with the update 6.0 [52], the `queueinglib` used in the AFDX model is replaced with the new one.

In the first AFDX model (3.3.1) `queueinglib` was included in the project under a subfolder inside the project itself, not handled separately. This the case for the work in [48] as well. However, it was an external library that is handled and updated by another party. Thus, in the latest model, `queueinglib` is added to the workspace as a separate library project and referenced in the AFDX project. By separating two projects, AFDX project becomes more resistant to changes in the `queueinglib`.

In previous AFDX models, queueinglib was seen as a part of the project and some changes are made in this external library. For example, application specific record calls are added manually. Since a logical separation is intended in the new model, those changes in the queueinglib are removed. Instead, a recommended method is used. In the samples library provided by OMNeT++, there is a project called queueinglib_ext[53].In this sample project, how to create new classes by extending queueinglib is explained. With this method, it is not only possible to use the functionality of the current functions, but also extend that functionality. For the interests of this thesis, it is needed to extend Source and Sink classes to add NetworkStatistics function calls. Hence Source_ext and Sink_ext are created. There is one exception to this method. There are so many moments needed to be recorded in the PassiveQueue class of the queueinglib. That's why, instead of extending it, PassiveQueue is copied from the queueinglib and modified. This can be fixed in the future.

As described in chapter 3.3.1, in the old queueinglib version that is used in the old AFDX model, Jobs that are derived from raw type cPacket were floating through blocks. But unfortunately, in the latest version the parent class is changed from cPacket to cMessage (Figure 4.3). cPacket was preferable because it consumes time according to the message length when transmitting through Ethernet line unlike cMessage. To solve this problem quickly, Job is changed back to the way it was and derived from cPacket. This is the only change that is made over the external queueinglib library. Although it is intended to keep the original queueinglib untouched, this change was inevitable with current knowledge. This problem could be solved in the future.

Figure 4.3 Old (Top) and Latest (Bottom) Job Classes

4.3. Changes In integrity Checker

The expected behavior from an AFDX integrity checker is to control PSN according to the equation (2.4). Although in the AFDX model that is used in [48], `Integrity Checker` checks only whether the SN is smaller than 255 or bigger than zero, it is fixed in the scope of this thesis and PSN check is added.

4.4. Changes In Traffic Policy

In the previous version of the AFDX model, for the sake of the work in [48], some specific VL-IDs were specially handled in this block. That code blocks are removed. Additionally, rho and sigma values that are explained in chapter 2.1.4.1 were calculated over and over at each term which is unnecessary and inefficient. Thus, rho and sigma are added as configuration parameters and their calculations are retained to the user in the newest version. Finally, when the token bucket algorithm results in a credit shortage, the previous version of the simulation was stopped with an error. It is modified to take record instead of stopping.

4.5. A New Connection Type: Cable

When creating a simulation with OMNeT++, it is possible to connect blocks by using regular channel or `DatarateChannel`. The difference is that simulation time progresses

when cPackets are transmitted through DatarateChannel. In other words, DatarateChannel can be considered as a realistic connection where the regular one is a more symbolic connection with infinite connection speed. Almost all the connections within the end-system and the ones within the switch are regular. On the other hand, the ones that model real, physical connections such as end-system – switch connections are DatarateChannels.

The built-in type DatarateChannel has several parameters such as delay (in sec) that symbolizes the propagation delay and data rate (in bps) which is the bus speed. Propagation delay is calculated with cable length and wave propagation speed ($2e^8$ m/s). It is not very convenient to expect a propagation speed from the user. Thus, in this new connection type that extends DatarateChannel (Figure 4.4), cable length is included as a parameter and propagation delay is calculated internally. A similar approach was also followed in INET type EtherLink [54].

```
channel Cable extends ned.DatarateChannel
{
    @display("ls=blue");
    @labels(Cable);
    double length @unit(m) = default(0m);
    delay = replaceUnit(length / 2e8, "s"); //2e8m/s -> wave propogation speed
}
```

Figure 4.4 New Connection Type Cable

4.6. Changes In Message Types and Source Structure

By considering the industry needs, a major modification is made. In old versions, there was a trafficSource and it was creating AFDXMessages. But in order to be able to run a more realistic scenario, trafficSource is divided into a messageSource and an AFDXMarshall and instead of the regular channel these two blocks are connected with the new connection type Cable that is described in chapter 4.5 . The old and new structures are demonstrated in Figure 4.5.

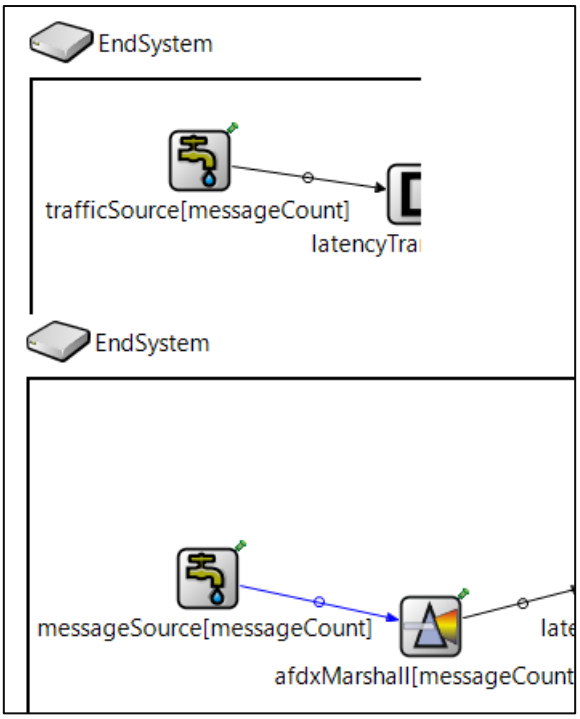

Figure 4.5 Old (Top) and Latest (Bottom) Source Structures

As a result of these changes, `messageSource` became a more generic message source that extends `queueinglib` type `Source`. It sends a message of type called `SubsystemMessage` which does not have any AFDX specific fields and its length is set only by considering payload length. On the other hand, `AFDXMessage` contains all the AFDX specific fields and its length is set by considering payload length plus other AFDX related overheads. For a better understanding, two message fields are shown in Figure 4.6.

SubsystemMessage.msg

```
20 import Job;
21
22 namespace afdx;
23
24 message SubsystemMessage extends queueing::Job
25 {
26     int partitionId;
27     int packetLength;
28 }
```

AFDXMessage.msg

```
27 message AFDXMessage extends queueing::Job
28 {
29     int networkId;
30     int equipmentId;
31     int interfaceId;
32     int virtualLinkId;
33     int seqNum;
34     int partitionId;
35     int udpSrcPort;
36     int udpDestPort;
37     double bagValue;
38     double rho;
39     double sigma;
40
41     simtime_t AFDXMarshallingTime;
42     simtime_t regLogExitTime;
43     simtime_t regLogEntryTime;
```

Figure 4.6 Subsystem (Top) and AFDX (Bottom) Message

After a `SubsystemMessage` is created and sent from `messageSource`, `AFDXMarshall` converts `SubsystemMessage` into a `AFDXMessage` and fills it by using configuration parameters set from `*.ini/*.ned` file. By these changes, `messageSource` and `cable` elements together can mimic a real partition, for instance a sensor sending data from RS485 in any data rate such as 115200 bps.

In addition to these changes, rho, sigma and three other `simtime_t` parameters are added. Rho and sigma are added due to the changes in `TrafficPolicy` and `simtime_t` parameters are added for record purposes.

4.7. A New Type: ConnDef and New Network Definition

Aside from some inadequacies, a major motive behind these improvements is to make this simulation more configurable and make it useful for those who are not familiar with the OMNeT++environment. In this regard, many hard coded parameters converted to configuration parameters and hence almost every characteristic of a network became

configurable. But here is one characteristic that is really hard to alter without changing the code itself, network topology. Network topology is defined with connections between end-systems and switches. In an AFDX model, this is defined by the `connections` section of the network `*.ned` file. In short, to modify the topology, `*.ned` file of the network must be configured. Although OMNeT++ is a really flexible environment and it allows certain parameters of `*.ned` file to be configured via `*.ini` file, it is not as easy to alter connections through in the same way. This chapter explains the key enhancement that enables network topology to be configurable.

A new type called `ConnDef` is added which is an acronym for "Connection Definition". This type is used to define a connection between blocks therefore it contains parameters to define each end of the connection. There can only be two ends for a connection and the blocks at each end can either be a switch or end-system. As a bonus, cable length of the connection is also added among other fields. `ConnDef` is demonstrated in Figure 4.7.

```
simple ConnDef
{
    bool isEntryAnEndsystem;
    bool isExitAnEndsystem;
    int entryIndex;
    int exitIndex;
    double cableLength @unit(m) = default(0m);
}
```

Figure 4.7 A New Type: `ConnDef`

In order to make network topology configurable, the `*.ned` file that defines the AFDX network is modified accordingly. Two new configuration parameters are added for the number of switches and number end-systems. End-systems and Switches are converted into arrays of variable sizes. For each end-system – Switch connection, a `ConnDef` block is added, which is also an array with size of [number of switches + number of end-systems - 1]. When defining the connections, parameters of `ConnDef` are used which is the actual tricky part. Finally, all the configuration parameters mentioned and fields of `ConnDefs` are filled from `*.ini` file. By this way, network topology became configurable by `*.ini` file only. The resulting network is named as "Auto Network" and described with `AutoNetwork.ned` file. It can be reviewed from the GitHub repository [20].

Not to forget, to make the simulation run properly, routing tables shall also be filled by considering the topology. Although it is easier than changing the code and recompiling it every time, there are still some works to do for the user such as creating an appropriate `*.ini` file and routing tables. To ease this procedure as well, a new network configuration tool named as ANCAT is proposed in this thesis. ANCAT will be explained in upcoming chapters is proposed.

4.7.1. Other Small Changes

Major improvements are listed above. Here, remaining smaller changes are summarized.

1. The per-VL queue inside the `RegulatorLogic` was unlimited. By considering the industry needs, this queue became upper-bounded.
2. After the latest update, unconnected ends started to give compile-time errors. There was a loose end in the connections of `priorityClassifier` in the `SwitchFabric`. The Classifier block of the `queueinglib` has multiple ports such as inputs (`in[]`) for entering packets, outputs (`out[]`) for classified output packets and a `rest` output for the packets there are not able to be classified. `priorityClassifier`'s inputs were connected to each `SwitchPort` and outputs were connected to `lowQueue` and `highQueue`. Which remains the port `rest` unconnected. To solve the issue, `allowunconnected` keyword is added to the `connections` section of `SwitchFabric.ned`.
3. VLRouter.cc class is used to contain a code section that drops certain kinds of messages with a certain possibility. This was added for a specific application about the work in [48] and it is removed since there is not a generic approach. Addition to that, code is slightly optimized and the routing table name became a configuration parameter.
4. `redundancyChecker` is modified by considering code readability.
5. Some redundant parameters and code blocks are removed all over the project and comments are added.
6. Deprecated functions are either removed or modified.
7. To keep record of a pre-defined set of metrics, `NetworkStatistics` function calls are added all over the project. Such as, `MAC` (For ES scheduling time and

switch entry/exit moments), `PassiveQueue` (For switch queueing metrics), `Source_ext` (For creation times), `RedundancyController` and `RegulatorLogic` (For BAG latency)

5. PROPOSED NETWORK CONFIGURATION AND ANALYSIS TOOL FOR AFDX (ANCAT)

In this thesis, a new network configuration and analysis tool for AFDX is proposed. A strong motivation of this thesis is to present an easily configurable simulation. The modifications made in the AFDX simulation model makes it easier to configure the simulation by using `*.ini` file. But changing `*.ini` file still requires some OMNeT++ experience. Reviewing simulation results also requires familiarity to OMNeT++ environment similar to the input configuration.

To save the user from that burden, a python-based tool ANCAT is proposed. It simply takes simulation configuration in a generic format such as `*.xlsx`, runs the simulation from the command shell and creates a report by processing simulation results. All users need to do is to prepare an "input.xlsx" file and run the batch file after specifying some important paths such as the location of the AFDX Simulation or OMNeT++ files.

ANCAT is composed of one batch file named "ANCAT_run.bat" and three python scripts that are called "PreProcessor.py", "SimProcessor.py" and "PostProcessor.py" (Figure 5.1).

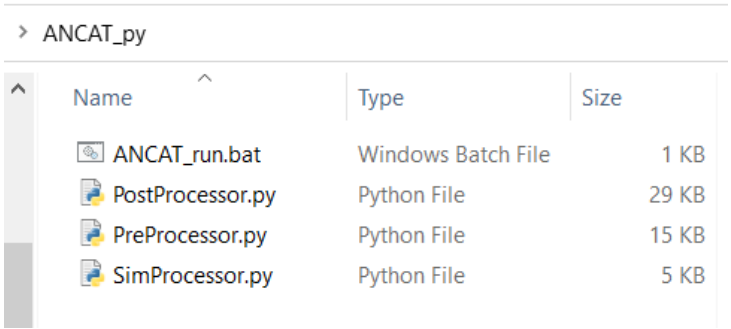

Figure 5.1 ANCAT components

The batch file named as ANCAT_run.bat is responsible for calling python scripts by specifying required options/paths. PreProcessor reads the input file then creates `*.ini` file

and routing table(s). Outputs of PreProcessor are created under the AFDX simulation folders. SimProcessor runs the simulation via command line interface and recording files (*.vci, *.vec) are created as a result. Finally, PostProcessor creates a report by using recording files. This process is demonstrated with a block diagram in Figure 5.2.

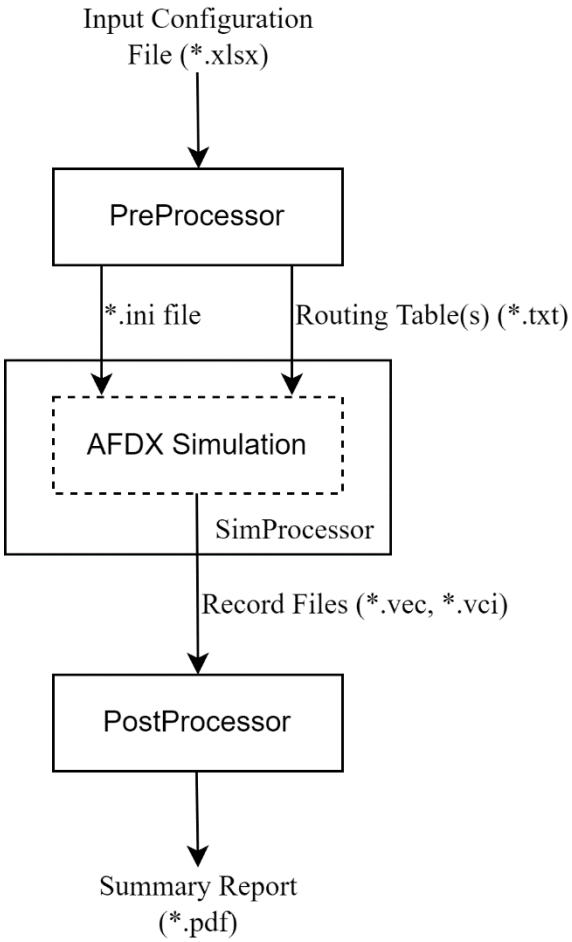

Figure 5.2 ANCAT Logical Block Diagram

5.1. PreProcessor and Input File

The AutoNetwork.ini file is the backbone of the simulation. PreProcessor creates the `*.ini` file and the routing table(s) and put them under the specified folder which is expected to be the “simulations” folder in the AFDX Simulation (Figure 5.3). Number of routing tables depends on the number of switches in the network configuration and its format is demonstrated in Figure 3.10.

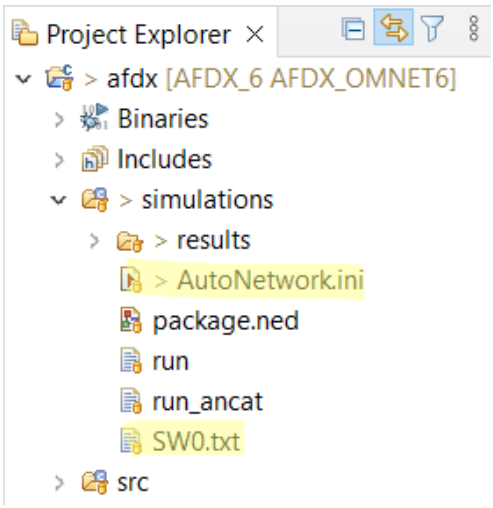

Figure 5.3 AFDX Model "simulations" Folder

Preprocessor fills the `*.ini` file partially with the information provided in the input configuration file, partially with some constant data. For example, some lines are added to enable/disable certain records or indicate the network name. In addition, there are some variables that don't affect the simulation behavior but needed to be specified. Lines for those variables are also created by the Preprocessor itself. The constant lines of both types that are created for an example `*.ini` files are demonstrated in Figure 5.4 and Figure 5.5.

AutoNetwork.ini

```
[General]
print-undisposed = false
**.vector-recording = true
**.scalar-recording = true
**.queueLength.statistic-recording = false
**.lifeTime.statistic-recording = false

#Network Params
network = afdx.AutoNetwork
```

Figure 5.4 An Example Ini File – General Network and Record Settings Section

AutoNetwork.ini

```
**.scheduler.serviceTime = 0s
**.afdxMarshall[*].networkId = 0x99
**.afdxMarshall[*].equipmentId = 0x66
**.afdxMarshall[*].interfaceId = 0
**.afdxMarshall[*].seqNum = 0
**.afdxMarshall[*].udpSrcPort = 0x1234
**.afdxMarshall[*].udpDestPort = 0x5678
**.afdxMarshall[*].frameHeaderLength = 47
```

Figure 5.5 An Example Ini File – Simulation Constants

For the other lines, the input `*.xlsx` file that is composed of four pages is used. The detail of each page is summarized below:

1. Instructions: Includes summarized explanations about each page of the input file and an example table demonstrating page 4 (Message Set).
2. Topology: The information contained in this page are used to define the lines about the connections between end-systems and switches in the `*.ini` file. Each end of the connections in the network shall be defined with cable lengths here. In addition to that, each end is supposed to be named according to their types such as “ES<n>” for end-systems and “SW<n>” for switches where “n” indicates the index and cable length shall be stated with a unit such as “m”. An example topology page can be seen in Figure 5.6 and resulting `*.ini` file section is demonstrated in Figure 5.7.

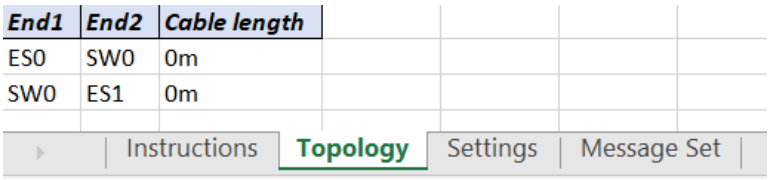

Figure 5.6 ANCAT Input File – Topology Page

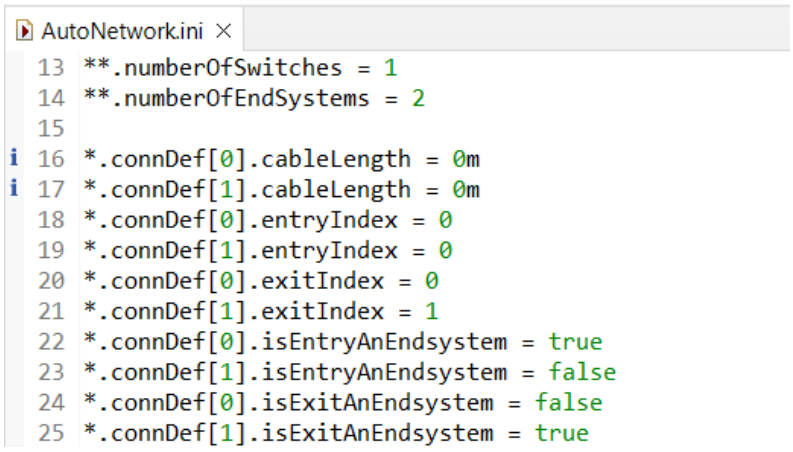

Figure 5.7 An Example Ini File – Connection Definitions Section

3. Settings: This page contains general settings and constants such as, technological delays, ethernet speed, skew max and the size of the per-VL queue in `RegulatorLogic`. Figure 5.8 shows a setting page of an example input file and Figure 5.9 demonstrates the resulting `*.ini` file lines.

Setting	Value
Switch Tech Delay	4us
ES Tx Tech Delay	32us
ES Rx Tech Delay	32us
Ethernet Speed	100Mbps
Skew Max	10ms
VL Queue size	1000

Instructions | Topology | **Settings** | Message Set

Figure 5.8 ANCAT Input File – Settings Page

AutoNetwork.ini

```
36
37 **.switchFabric.delay.delay = 4us
38 **.ESGroup[*].latencyTechTx.delay = 32us
39 **.ESGroup[*].latencyTechRx.delay = 32us
40 *.ethSpeed = 100Mbps
41 **.redundancyChecker.skewMax = 10ms
42 **.regulatorLogic.maxVLIDQueueSize = 1000
```

Figure 5.9 An Example Ini File – AFDX General Settings Section

4. Message Set: This is the most crowded page and contains an entry for each intended message source. By using the values stated in this page, the newly created `SubsystemMessages` and `AFDXMessages` are filled with. Hence this page includes detailed AFDX message information such as VL-ID, Partition ID, BAG, payload length, rho and sigma. In addition to those, some message source specific information is provided here. Simulation metrics such as start and stop times and cable length between `messageSource` and `AFDXMarshall` can be given as examples. Moreover, the information about source and destination end systems (Source ES and Destination ES) are combined with the ones in the Topology page. With this data, graphs are generated and the shortest paths between end-systems are obtained by using the Dijkstra Algorithm [55] to create VL-Routing tables. An example Message Set page is demonstrated in Figure 5.10. The resulting `*.ini` file lines are more than one page long. A small portion of them are shown in Figure 5.11.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	***Source ES***	***Destination ES***	***VLID***	***Partition ID***	***Start Time***	***Stop Time***	***BAG***	***Period***	***Payload Length***	***Rho***	***Sigma***	***Datarate of source***	***Cable length***
2	ES0	ES1	0x1	1	0s	1s	1ms	1ms	1183	10Mbps	15000	0Mbps	0m
3	ES0	ES1	0x2	1	0s	1s	1ms	1ms	1183	10Mbps	15000	0Mbps	0m
4	ES0	ES1	0x3	1	0s	1s	1ms	1ms	1183	10Mbps	15000	0Mbps	0m
5	ES0	ES1	0x4	1	0s	1s	1ms	1ms	1183	10Mbps	15000	0Mbps	0m

Instructions | Topology | Settings | **Message Set**

Figure 5.10 ANCAT Input File – Message Set Page

AutoNetwork.ini

```
47
48 **.ESGroup[0].messageCount = 4
49 **.ESGroup[1].messageCount = 0
50 **.ESGroup[0].afdxMarshall[0].rho = 10Mbps
51 **.ESGroup[0].afdxMarshall[0].sigma = 15000
52 **.ESGroup[0].afdxMarshall[0].BAG = 1ms
53 **.ESGroup[0].afdxMarshall[0].virtualLinkId = 0x1
54 **.ESGroup[0].messageSource[0].partitionId = 1
55 **.ESGroup[0].messageSource[0].startTime = 0s
56 **.ESGroup[0].messageSource[0].stopTime = 1s
57 **.ESGroup[0].messageSource[0].interArrivalTime = 1ms
58 **.ESGroup[0].messageSource[0].packetLength = 1183.0
59 **.ESGroup[0].messageSource[0].baudrate = 0Mbps
60 **.ESGroup[0].messageSource[0].cableLength = 0m
61
62 **.ESGroup[0].afdxMarshall[1].rho = 10Mbps
63 **.ESGroup[0].afdxMarshall[1].sigma = 15000
64 **.ESGroup[0].afdxMarshall[1].BAG = 1ms
65 **.ESGroup[0].afdxMarshall[1].virtualLinkId = 0x2
66 **.ESGroup[0].messageSource[1].partitionId = 1
67 **.ESGroup[0].messageSource[1].startTime = 0s
68 **.ESGroup[0].messageSource[1].stopTime = 1s
69 **.ESGroup[0].messageSource[1].interArrivalTime = 1ms
70 **.ESGroup[0].messageSource[1].packetLength = 1183.0
71 **.ESGroup[0].messageSource[1].baudrate = 0Mbps
72 **.ESGroup[0].messageSource[1].cableLength = 0m
73
74 **.ESGroup[0].afdxMarshall[2].rho = 10Mbps
```

Figure 5.11 An Example Ini File –AFDX Message Settings Section (Cropped)

5.2. Python Script Options and Batch File

The three python scripts require some paths or file names to be given as inputs. These paths and file names are used in the scripts to get the input files or to put the outputs in. Detailed information about these options is reachable to the user by “-h” command line option which is demonstrated in Figure 5.12, Figure 5.13 and Figure 5.14. As can be seen in the figures, PreProcessor needs input configuration file location and output file location to be specified. SimProcessor requires OMNeT++ installation and AFDX Simulation folders. Finally, PostProcessor needs the location of recording files, location and name for the output report. In addition to the listed options, it is possible to affect the behavior of the

script with specified flags which is not mandatory. Detailed information about these flags is demonstrated in Figure 5.14.

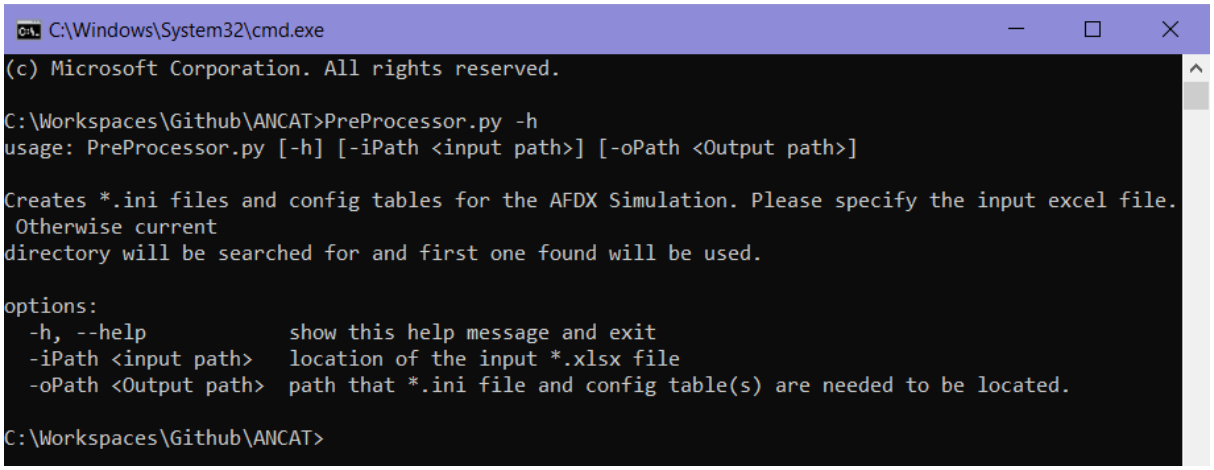

Figure 5.12 PreProcessor Help ("-h") Printout

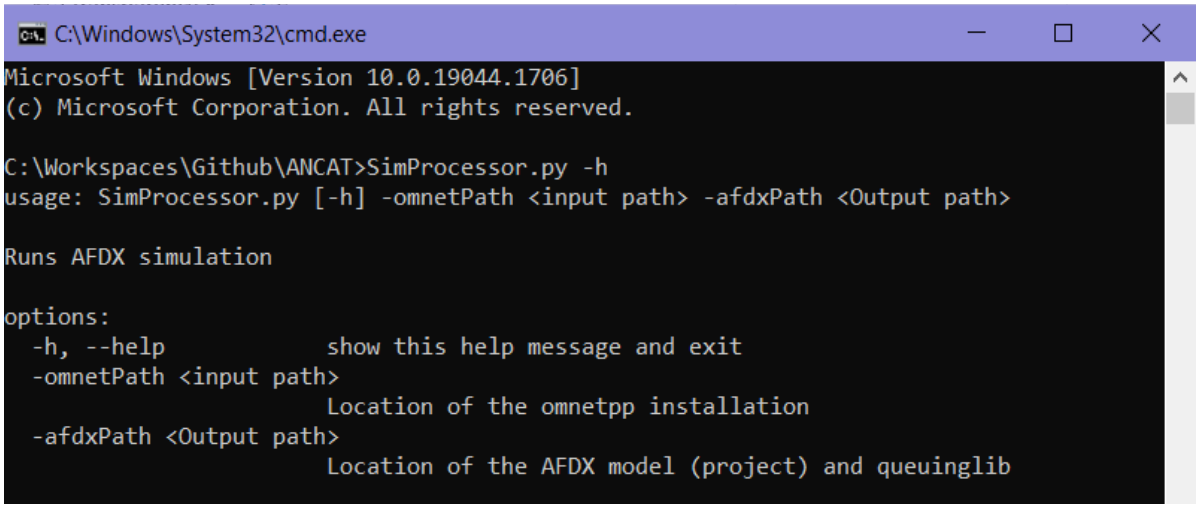

Figure 5.13 SimProcessor Help ("-h") Printout

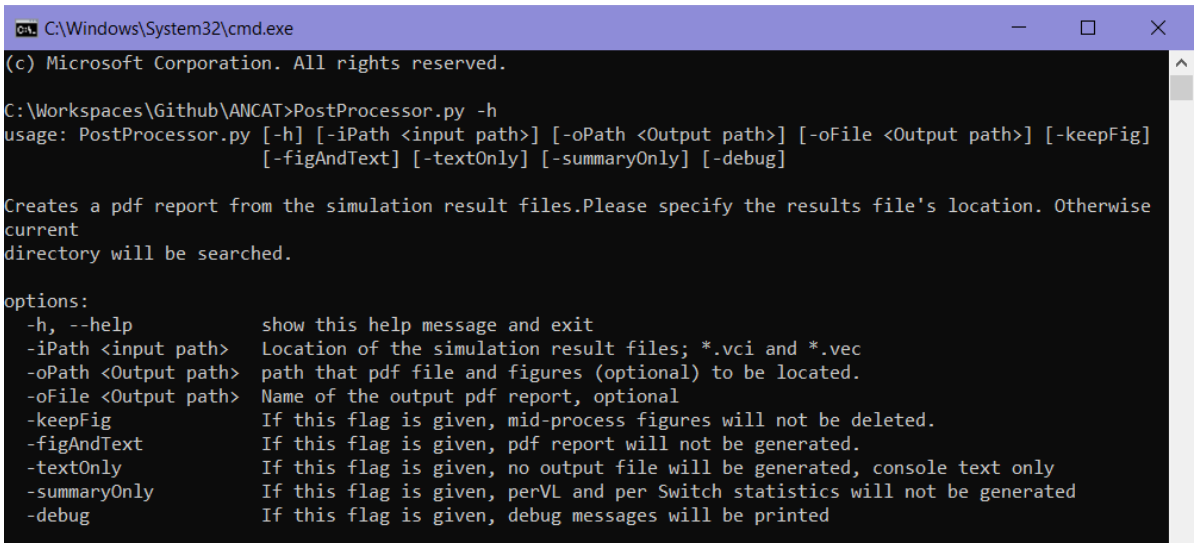

Figure 5.14 PostProcessor Help "-h" Printout

These needed paths must be provided by the user but to make the process easier, place holders are used in the batch file. The first six lines of the batch file are added to get the required location of certain files that depend on the user settings and assign them to relevant place holders. After that, the batch file runs the python scripts with place holders. An example batch file content is shown in Figure 5.15.

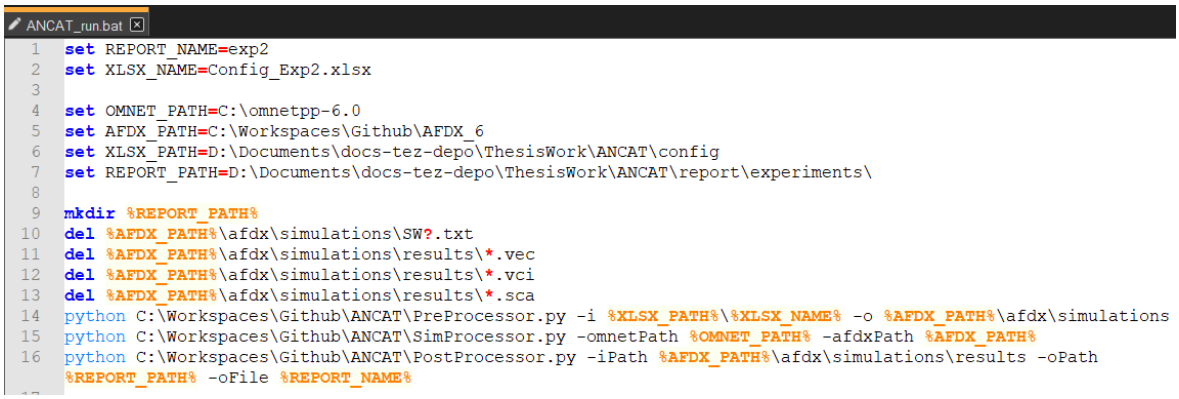

Figure 5.15 Example Batch File

5.3. Output

After simulation is completed, data that is recorded during the session are saved in record files in `*.vec` and `*.vci` formats and put under the folder "results" automatically (Figure 5.16). Finally, ANCAT takes these record files, processes them and creates a report.

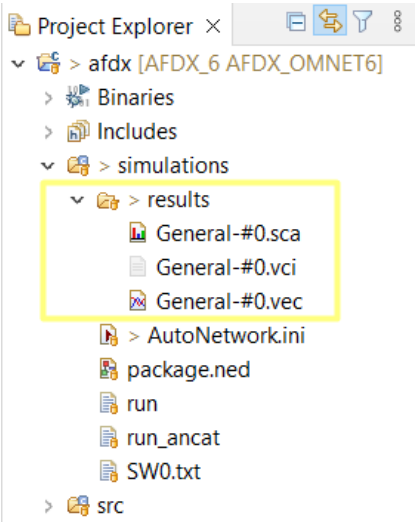

Figure 5.16 AFDX Model “results” Folder

There are three types of records with respect to *key* values that they are grouped under, per-VL, per-Switch and combined. The *keys* for per-VL records are VL-IDs and the *keys* for the per-Switch records are switch indexes. But the combined ones are a little more complicated. There are some records that are needed to be grouped under more than one characteristic. For example, token-bucket credits being recorded only per-VL doesn’t tell much. Because this record will show combinations of credit values in all switches i.e., without separating each switch. Let’s say, there is a message from a certain VL-ID and it passes through more than one switches. Thus, token-bucket credit records will be taken each time this message enters a switch and these records will be combined together since this record doesn’t distinguish switches. Finally, the resulting record will be showing credit management of different switches in the same batch which is meaningless. Therefore, "the *key* for credit records is defined as a combination of VL-ID and Switch index. Another example is the end-to-end latency. There is one per-VL end-to-end latency record. But this record groups only the end-to-end latencies with respect to the VL-IDs. If one VL is directed to multiple destination end-systems, this record will be showing a combination of end-to-end latencies recorded in different destinations. Thus, another record is added to get end-to-end latencies of messages arriving at each destination end-system per VL-ID and to do that, the key is defined as a combination of VL-ID and destination end-system index.

The information that is contained in the recording files can listed as below:

Per-VL Records:

- End-to-End Latency

- ES BAG Latency
- ES Scheduling Latency
- ES Total Latency
- Switch Queueing Time
- Dropped Frames in Queue
- Dropped Frames Count at TrafficPolicy

Per-Switch Records:

- Switch Queueing Time
- Switch Queueing Length

Combined records:

- Token-Bucket Credit (Per-VL + Per-Switch)
- End-to-End Latency (Per-VL + Per-Receiver ES)
- Switch Queueing Time (Per-VL + Per-Switch)
- Switch Queueing Time (Per-SW + Per-Port)
- Switch Queueing Time (Per-SW + Per-Port + Per-VL)
- Switch Queue Length (Per-VL + Per-Switch)

Output results report is composed of three main parts. “Overall Statistics”, “Per-Switch Statistics” and “Per-VL Statistics”. Each statistics is explained in detail below.

1. Overall Statistics: This part summarizes all results. The first page shows some general quantities such as “Overall Frame Count”, “Overall Simulation Time”, “Overall Dropped Frame Count” and “Overall Dropped Frame Percentage”. Aside from these, it gives maximum and mean values and confidence intervals (Chapter 2.1.7) of all the metrics that are recorded. The remaining pages of this section show overall metrics. For example, where per-VL metrics are plotted for all VL-IDs together per-switch metrics are plotted for all switches together.
2. Per Switch Statistics: This part demonstrates records per-switch and some of the combined records separately for each switch
3. Per VL Statistics: This part demonstrates records per-VL and some of the combined records separately for each VL-ID.

6. AFDX MODEL VERIFICATION TESTS

Presented AFDX model is verified based on the experiment results and their theoretical results computed-on-paper to see if they comply with them or not. In each experiment, a network topology is designed and one or more scenarios are defined with message characteristics (BAG, period, message length, sigma, rho, etc.). To evaluate the simulation, certain aspects (time, packet count, etc.) are recorded at some points marked in Figure 6.1. The experiments are executed with the contributions of ANCAT. The metrics and plots that are mentioned in the following chapters are extracted from the records and interpreted by the tool itself.

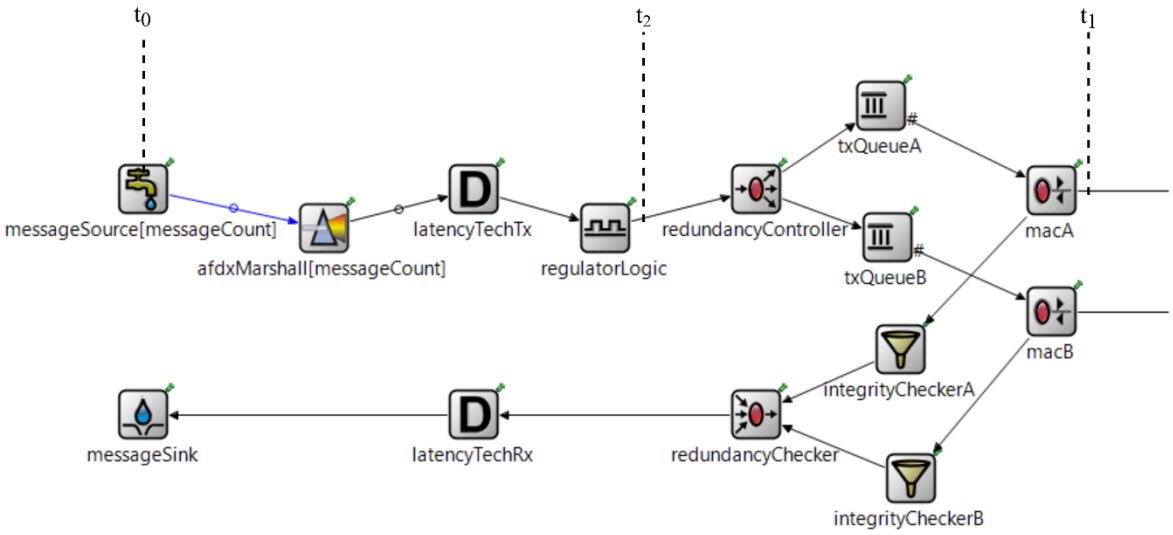

Figure 6.1 Record Points

6.1. Experiment 1: Regulator BAG Enforcement

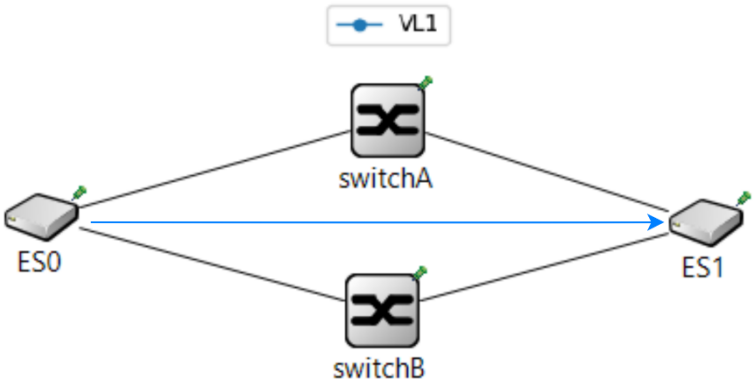

Figure 6.2 Experiment 1 – Topology

This experiment's objective is to demonstrate the BAG regulation behavior of the model. There are two end-systems in this network. ES0 is sending messages while ES1 is

receiving them. Three scenarios will be held in this experiment and the main distinction among them is the relations between inter-arrival times and BAG values. The values that are given in Table 6.1 are simulation constants and Table 6.2 presents scenario specific parameters.

Table 6.1 Experiment 1 – Simulation Constants

Definition	Value
Start time of the simulation	0 s
Stop time of the simulation	1 s
End-system technological latency	32 us
Switch technological latency	4 us

Table 6.2 Experiment 1 – Scenario Characteristics

Scenario	# Of VLs	Inter-arrival time (msec)	BAG (msec)
1	1	1	0.5
2		0.5	1
3		$1 \pm rand^{1}(0,0.2)$	1

To assess the results, two aspects will be investigated. One is the time difference between end-system output (t_1 in Figure 6.1) and message creation time (t_0 in Figure 6.1) which is the total end-system latency. The other is the frequencies of specified ranges i.e., histograms for inter-arrival time values that are measured at message creation (t_0 in Figure 6.1) and after BAG regulation (t_2 in Figure 6.1). Since there is only one VL in this experiment, messages will not be facing any jitter. Only delaying element will be the BAG regulation.

To interpret the simulation results, latencies and inter-arrival times will be reviewed for each scenario. In terms of end-system latency, the difference "$t_1 - t_0$" i.e., total end-system latency will be investigated and for the inter-arrival times, measured time differences at `MessageSource` (Δt_0) and `RegulatorLogic` (Δt_2) will either be represented textually or with tables.

[1] rand: random. It depicts a random number generator function that creates random numbers between 0 and 0.2.

6.1.1. Scenario 1

In this scenario, message creation period is bigger than the BAG. Therefore, it is expected to see that messages are leaving the end-system in the period that they are created. Regulator will not be introducing any additional delay. Thus, messages will not be facing BAG latency and total end-system latency will be equal to end-system technological latency.

In fact, simulation results are conforming these expectations. At the end of the simulation, one thousand packets are created and then reached to the sink. When the difference between successive measurements of t_0 and t_2 values are calculated separately, they appeared to be equal as expected. Moreover, measured end-system latency values are equal to the technological latency as expected. Results can be seen in Table 6.3.

Table 6.3 Time Difference Measurements

	Time differences (Δt, msec)		**Frame Count**
	Expected	**Measured**	
Creation (Δt_0)	1	1	1000
After BAG Regulation (Δt_2)	1	1	1000
Overall End-System Latency ($t_1 - t_0$)	32 (= tech. latency)	32	1000

6.1.2. Scenario 2

In this scenario, the message creation period is smaller than the BAG, one half of it to be exact (Period = BAG/2). Even though this approach does not comply with the best practice, it is used to assess the behavior of the simulation model in a data burst situation. It is expected to see that for every two messages created in each BAG slot, only one message will be sent. This will result in an overload and this experiment will last twice of the expected duration i.e., two seconds.

At the end of this simulation, 2000 packets are created and then reached to the sink. Even though the difference between successive measurements of t_0(at source) is 0.5 msec, the difference between successive t_2(after BAG regulation) measurements is 1 msec, i.e., BAG, as expected. This difference is due to the BAG regulation; every one out of two messages are delayed until BAG in the `RegulatorLogic` block.

Table 6.4 Time Difference Measurements

	Time differences (Δt, msec)		Frame Count
	Expected	**Measured**	
Creation (Δt_0)	0.5 msec	0.5 msec	2000
After BAG Regulation (Δt_2)	1 msec	1 msec	2000

Figure 6.3 shows the total end-system latency and Figure 6.4 provides a closer look. Since the messages are created in a higher pace than the BAG, after the first message, each one must wait for the BAG regulator to allow them to go. While the first message is delayed by 32 µs (as technological latency), the succeeding messages are delayed more and more, cumulatively. This behavior can be seen in the Figure 6.3, the latest frame is facing with a latency of one second. As opposed to the Scenario 1; the simulation is concluded at t = 2s instead of t = 1s.

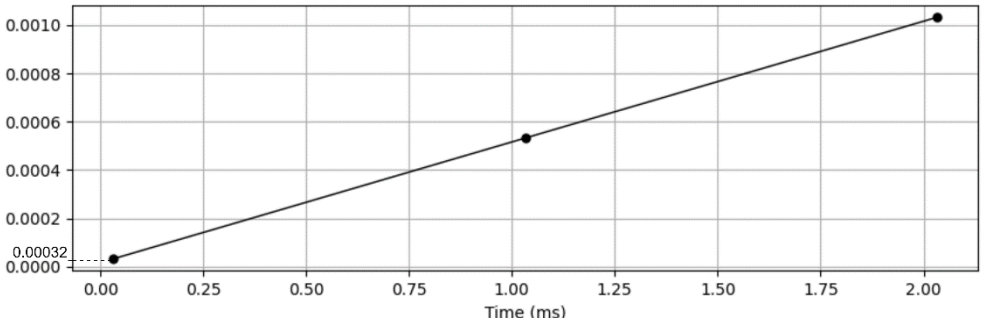

Figure 6.3 Scenario 2 - Total End-System Latency – Close Up

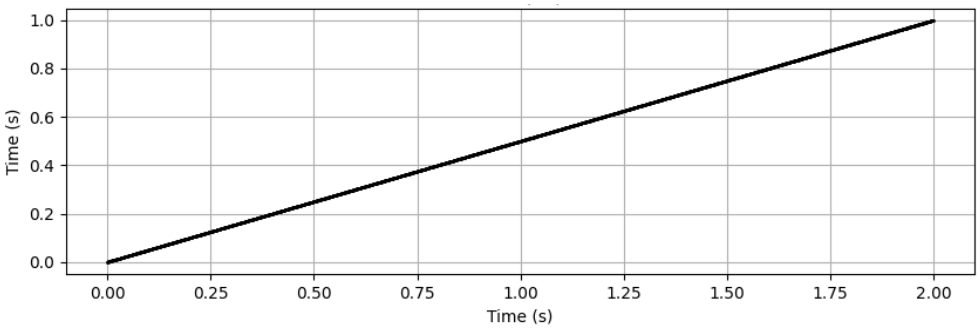

Figure 6.4 Scenario 2 - Total End-System Latency

6.1.3. Scenario 3

In this scenario, inter-arrival times of each message are varying around the BAG with a uniform random distribution. Accordingly, inter-arrival time will be both bigger and smaller than the BAG occasionally during the simulation. This may result in momentary overloads but the simulation is expected to be balanced and does not take additional time.

Figure 6.5 clearly indicates that messages are created in the expected paces. Whereas in the Figure 6.6, all frames are regulated according to the BAG and the smallest time difference between successive frames becomes 1 msec as expected. Remaining two tiny bins in Figure 6.6 show the frames with inter-arrival times higher than BAG.

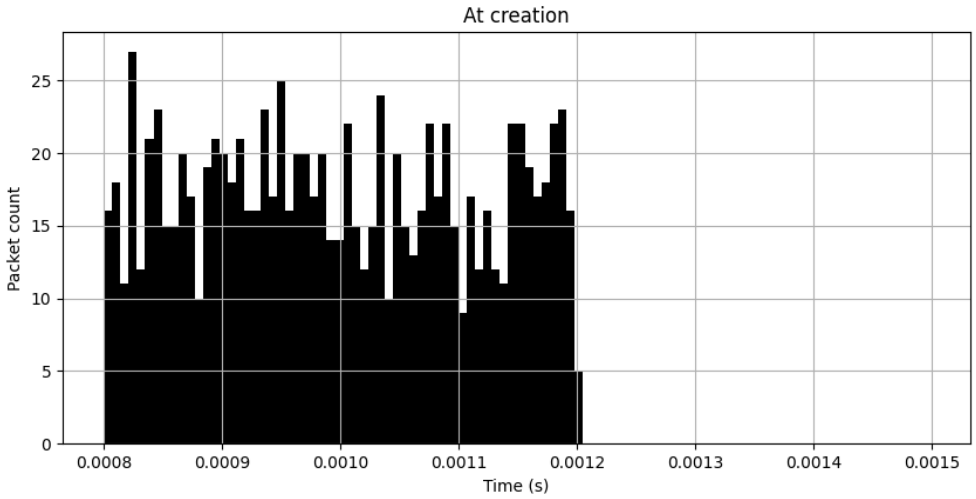

Figure 6.5 Scenario 3 – Inter-arrival Time Histogram at Creation

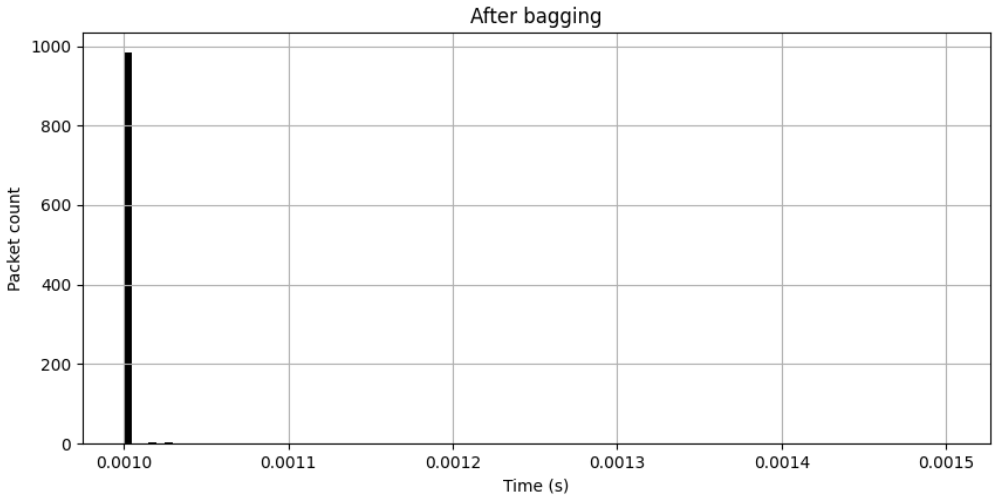

Figure 6.6 Scenario 3 – Inter-Arrival Time Histogram After BAG Regulation

Since there are frames faster than BAG, this topology presents a behavior similar to the one in Scenario 2 in the moments where inter-arrival times are smaller than the BAG. This attitude shows up in Figure 6.7 with increasing values of latency. On the other hand, the frames with inter-arrival times bigger than or equal to the BAG, behave in a fashion similar to the ones in Scenario 1 and drain-out the overloaded frames. That is why, simulation is still finished at about one second unlike Scenario 2 and Figure 6.7 has a lot of ups and downs.

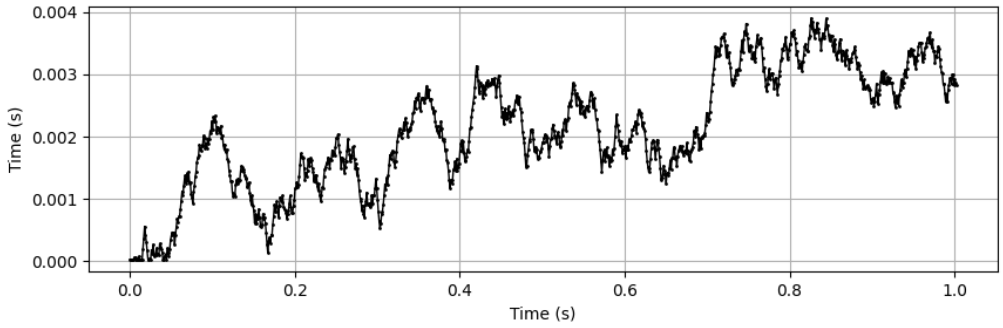

Figure 6.7 Scenario 3 – Total End-System Latency

6.2. Experiment 2: End-System Jitter

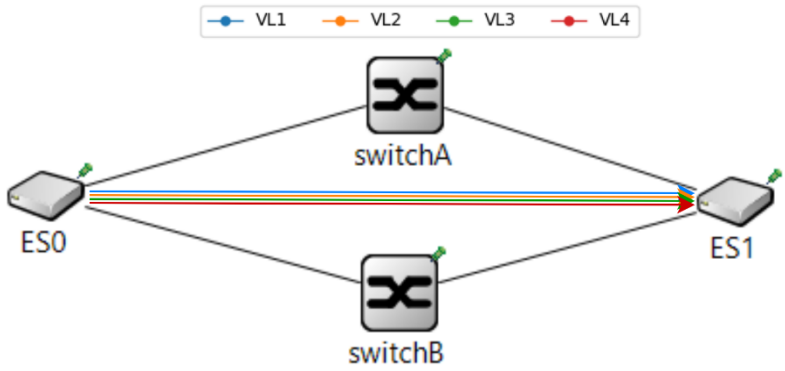

Figure 6.8 Experiment 2 – Topology

The purpose of this experiment is to examine the end-system jitter that is explained in chapter 2.1.3.1.1. The topology is given in Figure 6.8 and the simulation constants in Table 6.1 are binding for this experiment as well. As for scenario characteristics, Table 6.5 is added. In the table, the value given under the transmission time can also be viewed as contention delay which is calculated by using Equation (2.5) where C is taken 100 Mbps.

Table 6.5 Experiment 2 – Scenario Characteristics

# of VLs	Inter-arrival time (msec)	BAG (msec)	S (bytes)	Transmission Time (µs)
4	1	1	1250	100

In this experiment, there are messages of four different VLs that are leaving their source at the same time. Since inter-arrival time is equal to the BAG and they have different VL-IDs, each batch of messages will pass through the regulator block without getting queued. As a result, messages of four VLs will arrive at the queue of the "MAC block at the same time. The MAC block which is responsible for handing over the messages to the physical line, must wait before the previous message is completely transmitted.

Thus, even though messages from four VLs arrive at the `MAC`'s queue simultaneously, they will wait for each other and jitter with an amount of transmission time will be introduced.

For jitter measurement, the time difference between the message creation (t_0 in Figure 6.1) and end-system output (t_1 in Figure 6.1) are recorded when they are about to leave the end-system. The total end-system latency plot is given in Figure 6.9 and the actual values are listed in Table 6.6 for a more readable demonstration.

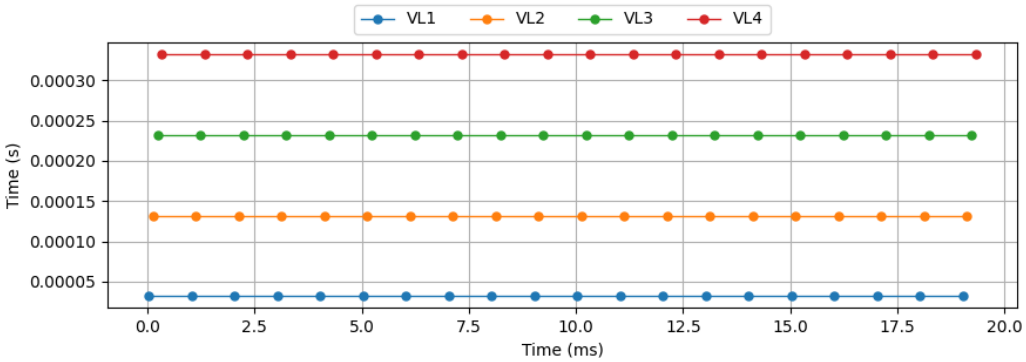

Figure 6.9 Experiment 2 – Total End-System Latencies

Table 6.6 Experiment 2 – Total End-System Latencies

VL-ID	**ES Latency**
1	0.000032
2	0.000132
3	0.000232
4	0.000332

As can be seen in Figure 6.9 and Table 6.6 while the first message is facing with a technological latency only, the other are delayed as technological latency plus total transmission time of the all previous messages as denoted in the equation (2.6).

6.3. Experiment 3: Account Management

The purpose of this experiment is to monitor the behavior of the `TrafficPolicy` block, i.e., the token-bucket algorithm by playing with the switch jitter. As described in chapter 2.1.4.1, the token-bucket algorithm is used to control the bandwidth. If two successive frames of the same VL are intended to be sent too close to each other, the second one couldn't gain enough credit and get dropped as in the equation (2.14). Normally, the time difference between two messages of the same VL is controlled in the end-system with BAG. However, sometimes due to additional delays caused by multiple end-systems, VLs

or switches, two messages might get closer than they should. In that case, it is expected to switch to police that traffic.

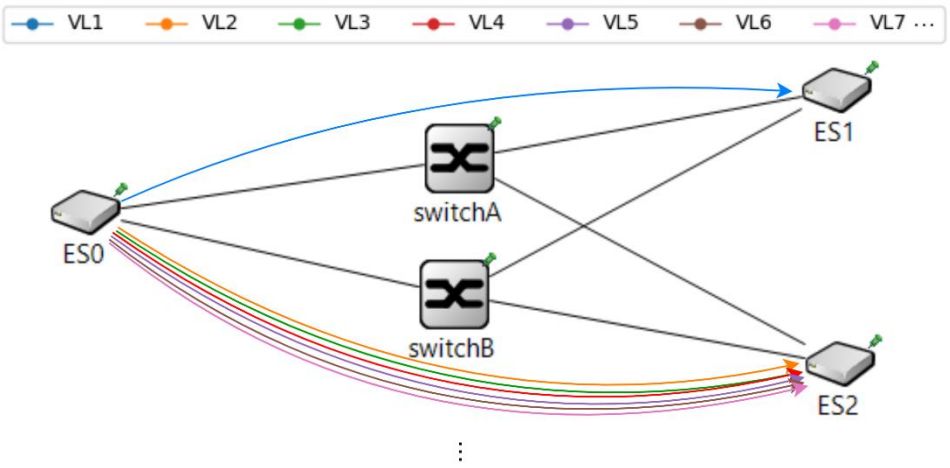

Figure 6.10 Demonstration – Network Topology

To demonstrate this issue, let us assume that there are three end-systems in a network, one is transmitter (ES0), and the other two are receivers (ES1 and ES2). There are 'n' VLs leaving the ES0 and while one of them is addressed to ES1, the other 'n-1' ones are addressed to ES2. Network and VLs are denoted in Figure 6.10. For the sake of the demonstration, inter-arrival time (t_i) for VL1 is selected as 1 msec and 3 msec for all the other VLs.

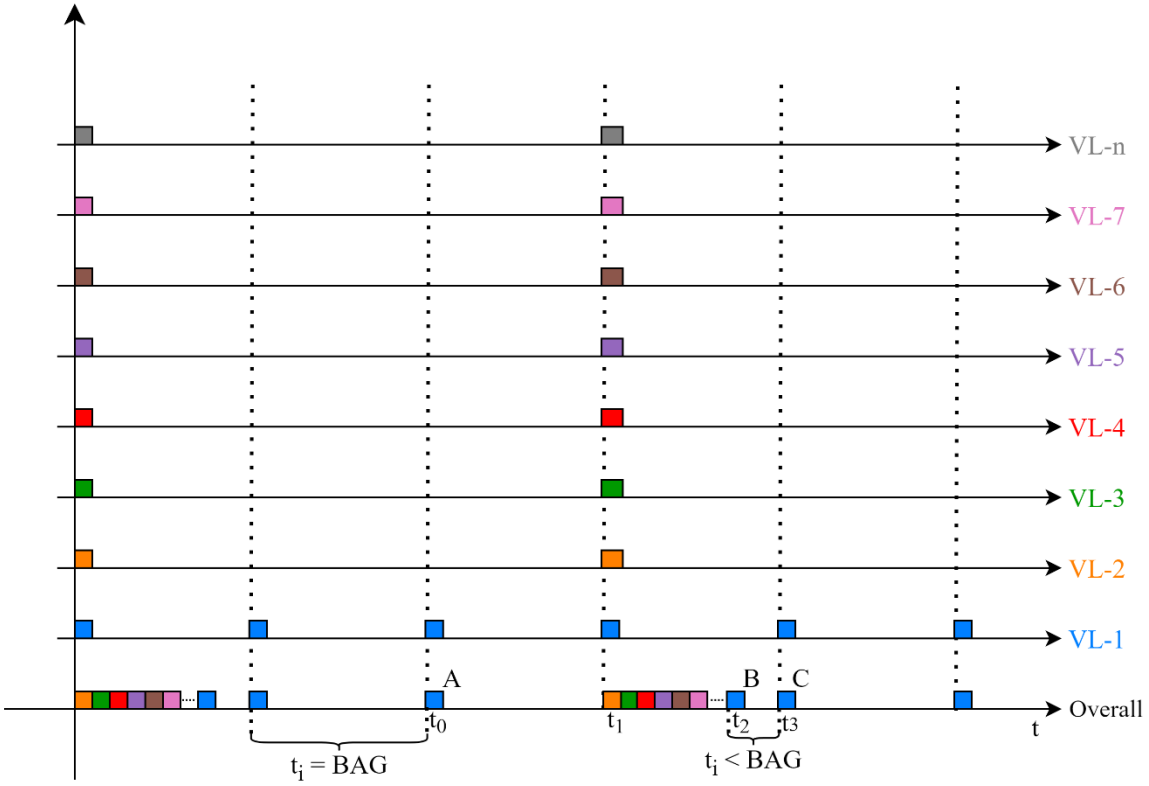

Figure 6.11 Demonstration – Message Traffic at Switch Input

In every three msecs, ‘n-1’ messages of other VL-IDs in addition to the one message of VL1 (frame B) will be entering the switch and causing a delay to VL1. The next message after this turn will be a message of VL1 (frame C) and the time difference between two VL1 messages will be shorter than BAG due to the contention. By considering the amount of credit, frame B will be dropped or not. This expected behavior is visualized in Figure 6.11.

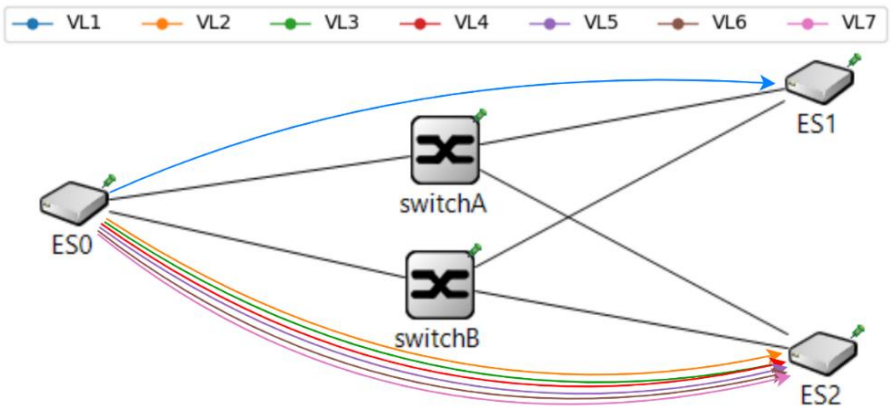

Figure 6.12 Experiment 3 – Topology

Table 6.7 Experiment 3 – Scenario Characteristics

# of VLs	Inter-Arrival Time (msec)	Transmission Time (µs)	BAG (msec)	S (bits)	Rho (Mbps)	Sigma (bits)
1	1	100	1	10000	10	15000, 20000
6	3					15000

In this experiment, the previously explained demonstration will be run by the AFDX simulation with 7 VLs (n = 7), the characteristics of which are given in Table 6.7 and the topology given in Figure 6.12. If the scenario given in Figure 6.11 is considered together with these characteristics, the transmission of successive frames from each VL is expected to take 700 usec in total which coincides with the time interval $[t_2\text{-}t_1]$. The BAG value is also marked in the Figure 6.11 with time interval $[t_3\text{-}t_1]$ and it is equal to 1000 usec. Therefore, the time difference between two successive VL1 frames, i.e., $[t_3\text{-}t_2]$, must be 300 usec. The derivations are given with equation (6.1).

$$\text{Total transmission time} = t_2 - t_1 = 7 * 100 = 700 \text{ usec}$$
$$BAG = t_3 - t_1 = 1\ msec = 1000\ usec$$
$$t_3 - t_2 = 1000 - 700 = 300 \text{ usec} \qquad (6.1)$$

Token-bucket algorithm is computed with these timings for the frames that are denoted as A, B, C and D and for two different sigma values that are given Table 6.7. The expected results for sigma = 15000 and sigma = 20000 are given in Table 6.8 and respectively. The moment that packet drop is expected to happen is marked with red in Table 6.8. In a regular token-bucket computation, token value cannot be negative and in case of obtained token is less than the token that is needed to be spent, the frame is dropped. But for simplicity, packet drop is represented with a negative remaining credit.

Table 6.8 Experiment 3 – Credits when Sigma = 15000 bits

Tokens (bits)	Frame A (t = 1 msec)	Frame B (t = 0.3 msec)	Frame C (t = 1.7 msec)	Frame D (t = 1 msec)
Initial	15000	5000	8000	10000
Obtain-ed	10000	3000	17000	10000
Total	15000	8000	15000	15000
Remaining	5000	-2000	5000	5000

Table 6.9 Experiment 3 – Credits when Sigma = 20000 bits

Tokens (bits)	Frame A (t = 1 msec)	Frame B (t = 0.3 msec)	Frame C (t = 1.7 msec)	Frame D (t = 1 msec)
Initial	20000	10000	3000	10000
Obtained	10000	3000	17000	10000
Total	20000	13000	20000	20000
Remaining	10000	3000	10000	10000

After the simulation is executed with the mentioned characteristics, results are processed and following figures are obtained (Figure 6.13 and Figure 6.14). There might be insignificant differences between computations and simulation outputs because small time differences like IFG are not considered in the theoretical derivations for simplicity. But even so, simulation results are fully satisfying the expectations. The frame drop moments are shown with negative spikes in the simulation results and as expected they are only seen in the (sigma = 15000) scenario. Graphs are very close to the theoretical traffic graph (Figure 6.11) and the values are almost the same as the computations (Table 6.8).

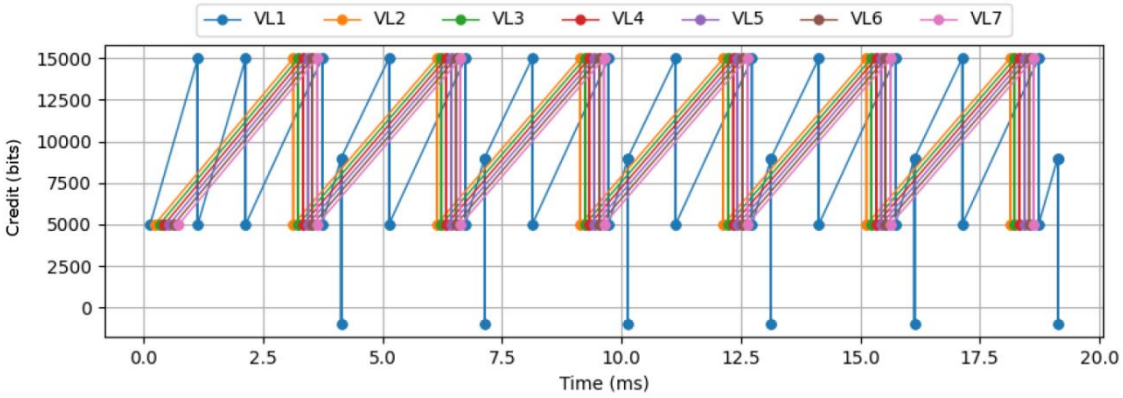

Figure 6.13 Experiment 3 – Change in Credit for Sigma = 15000 bits

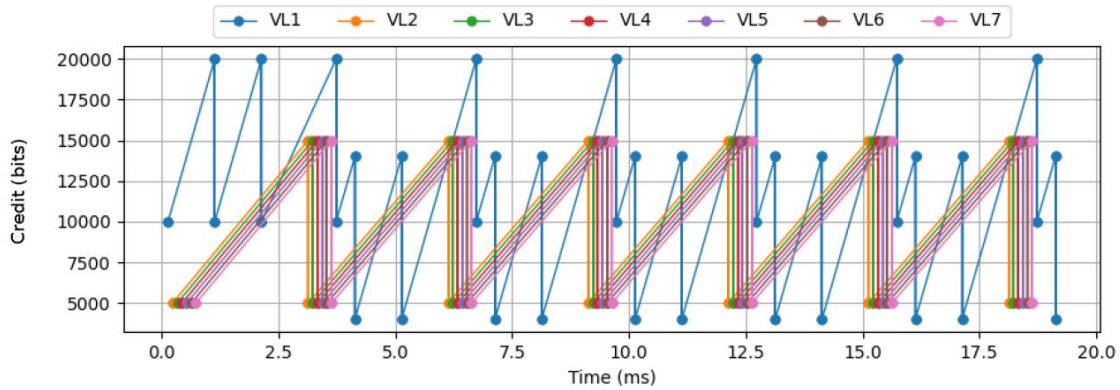

Figure 6.14 Experiment 3 – Change in Credit for Sigma = 20000 bits

6.4. Experiment 4: Switch Latency and Queue Management

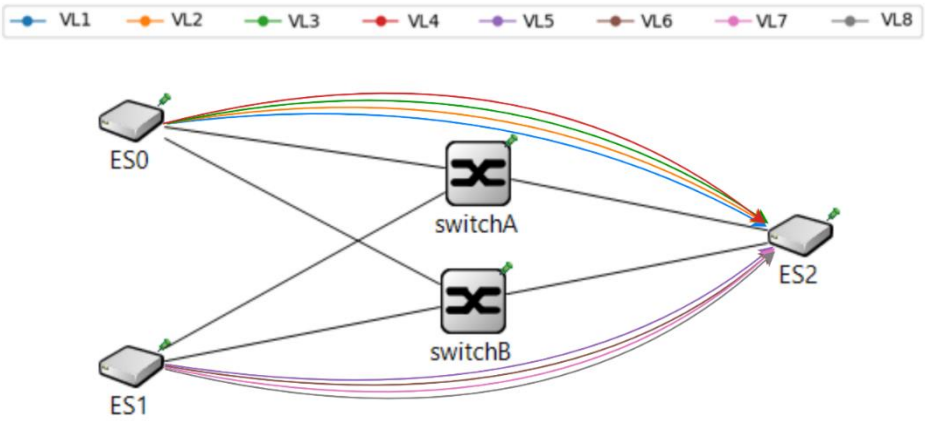

Figure 6.15 Experiment 4 – Topology

The purpose of this experiment is to investigate the behavior of the AFDX switch by using Little's Law (2.1.6). The topology is given in Figure 6.15 and the simulation constants in Table 6.1 are binding for this experiment as well. As for scenario characteristics, Table 6.10 is added. To be able to apply the Little's Law, packet drops must be hindered. Since the purpose of this experiment is to investigate the queueing behavior not traffic policing, sigma is intentionally set to a value that is bigger than necessary to prevent any packet drops.

Table 6.10 Experiment 4 – Scenario Characteristics

# of VLs	Inter-arrival time (msec)	BAG (msec)	S (bytes)	Sigma
4	$1 \pm rand(0,0.2)$	1	1250	5*S
4				

Three metrics are needed to apply Little's Law. First one is the average number of items in a queue (L). This can also be expressed as the average queue length of the switch and measured in the `txQueue` block by counting entering/exiting frames which is marked in the Figure 6.16 with "L" in red. The second parameter is the average queueing time (W) which can also be expressed as the time difference for a frame between entry and exit moments in and out of the queue. In this experiment, the time difference between entry and exit points are calculated and recorded at the `txQueue` block when frames are leaving the queue. The measurement points are marked with labels "t_{in}" and "t_{out}" in Figure 6.16. The third and final parameter is the average frame rate (λ). Since this simulation setup is specifically designed to assess the switch behavior and there are no packet drops, all frames follow the same path through the switch until the `Sink` block. Hence average

frame rate is calculated with the total number of frames arriving at the `Sink` divided by the total simulation time which is the time of the latest frame entering to the `Sink`.

In the current setup, since there are 8 different VLs with 1msec of BAG, it is expected to see an average frame rate of 8 frames/msec. Hence average queue length per average queueing time shall give approximately 8 frames/msec to satisfy the equation (2.17).

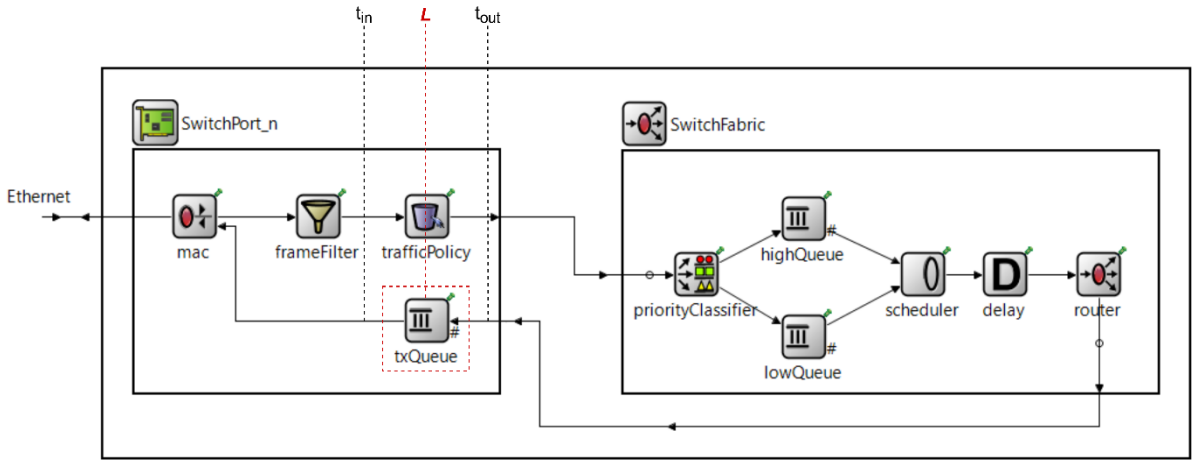

Figure 6.16 Experiment 4 – Measurement Points

The simulation results given in Table 6.11 presents the essential metrics for Little's Law. With total frame count and total simulation time, the average frame rate (λ) is evaluated as 7945.32 frame/second or 7.94 frame/msec which is approximately equal to 8 frame/msec as anticipated. Moreover, the calculations given in (6.2) clearly show that the modeled AFDX switch satisfies Little's Law as expected. The results also proves that the measurements recorded at the switch are valid and accurate.

Table 6.11 Experiment 4 – Simulation Results

Total Frame Count	8019
Total Simulation Time	1.0092 sec
Average Queue Length (L)	0.3035 frames
Average Queueing Time (W)	38.202 usec

$$0.3035 \stackrel{?}{=} 38.202e^{-6} * \frac{8019}{1.0092}$$
$$= 0.3035 \blacksquare \qquad (6.2)$$

6.5. Experiment 5: Skew Max Control

The purpose of this experiment is to show the behavior of AFDX Model when a redundant frame is delayed more than `skewMax`. The expected behavior is explained in chapter 2.1.3. In this experiment the topology (Figure 6.15) and characteristics (Table 6.10) used in Experiment 4 (Chapter 6.4) will be used. Additionally, `skewMax` is selected as 10msec.

To be able to demonstrate that case, a test block is added to the end-system module between `integrityCheckers` and `redundancyChecker` that is called `skewMaxTester`. This block has two inputs and two outputs. It manipulates the traffic of frames with a certain VL-ID but for all other frames, it is transparent. It directs frames received from `integrityCheckerA` to `redundancyChecker`'s `inA` port and `integrityCheckerB` to `redundancyChecker`'s `inB` port. Connections and placement can be seen in Figure 6.17.

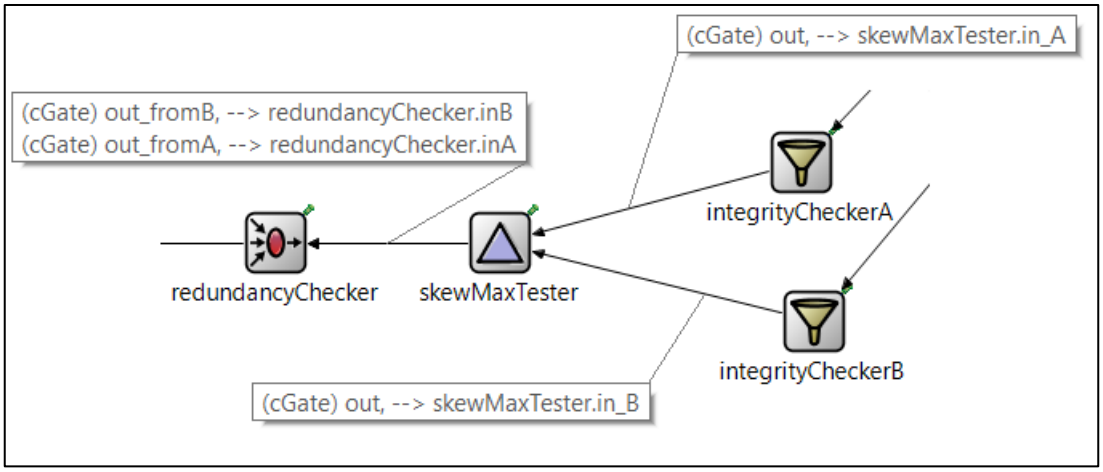

Figure 6.17 Experiment 5 – Skew Max Tester Block in End-System

If the received frame's VL-ID is equal to one, `skewMaxTester` will operate. It will delete certain messages in order to widen the time difference between successive frames. After more than 10 msec passes, it will resend the latest frame and it is expected from `redundancyChecker` to accept this frame one more time. Because messages received after a time difference more than `skewMax` should not be considered as a redundant frame even though they are the same.

```
0.0505521091|SkewTester|Forwarding (SN: 49):NW-A
0.0505521091|SkewTester|Forwarding (SN: 49):NW-B
0.0505521091|redChecker|Received (SN: 49):NW-A
0.0505521091|redChecker|Accepted (SN: 49):NW-A
0.0505521091|redChecker|Received (SN: 49):NW-B
0.0505521091|redChecker|Deleted(redundant) (SN: 49):NW-B
0.0515521091|SkewTester|Forwarding (SN: 50):NW-A
0.0515521091|SkewTester|Deleted (SN: 50):NW-B
0.0515521091|redChecker|Received (SN: 50):NW-A
0.0515521091|redChecker|Accepted (SN: 50):NW-A
0.0525521091|SkewTester|Deleted (SN: 51)
0.0525521091|SkewTester|Deleted (SN: 51)
0.0535521091|SkewTester|Deleted (SN: 52)
0.0535521091|SkewTester|Deleted (SN: 52)
0.0545521091|SkewTester|Deleted (SN: 53)
0.0545521091|SkewTester|Deleted (SN: 53)
0.0555521091|SkewTester|Deleted (SN: 54)
0.0555521091|SkewTester|Deleted (SN: 54)
0.0567521091|SkewTester|Deleted (SN: 55)
0.0567521091|SkewTester|Deleted (SN: 55)
0.0577521091|SkewTester|Deleted (SN: 56)
0.0577521091|SkewTester|Deleted (SN: 56)
0.0587521091|SkewTester|Deleted (SN: 57)
0.0587521091|SkewTester|Deleted (SN: 57)
0.0597521091|SkewTester|Deleted (SN: 58)
0.0597521091|SkewTester|Deleted (SN: 58)
0.0607521091|SkewTester|Deleted (SN: 59)
0.0607521091|SkewTester|Deleted (SN: 59)
0.0619859181|SkewTester|Deleted (SN: 60)
0.0619859181|SkewTester|Deleted (SN: 60)
0.0629859181|SkewTester|Deleted (SN: 61)
0.0629859181|SkewTester|Deleted (SN: 61)
0.0639859181|SkewTester|Deleted (SN: 62)
0.0639859181|SkewTester|Deleted (SN: 62)
0.0649859181|SkewTester|Deleted (SN: 63)
0.0649859181|SkewTester|Deleted (SN: 63)
0.0659859181|SkewTester|Deleted (SN: 64)
0.0659859181|SkewTester|Deleted (SN: 64)
0.0669859181|SkewTester|Changed (SN: 65 -> 50):NW-B
0.0669859181|SkewTester|Changed (SN: 65 -> 50):NW-A
0.0669859181|redChecker|Received (SN: 50):NW-B
0.0669859181|redChecker|Accepted(skewMax!!) (SN: 50):NW-B
0.0669859181|redChecker|Received (SN: 50):NW-A
0.0669859181|redChecker|Deleted(redundant) (SN: 50):NW-A
0.0679859181|SkewTester|Forwarding (SN: 66):NW-A
0.0679859181|SkewTester|Forwarding (SN: 66):NW-B
0.0679859181|redChecker|Received (SN: 66):NW-A
0.0679859181|redChecker|Accepted (SN: 66):NW-A
0.0679859181|redChecker|Received (SN: 66):NW-B
0.0679859181|redChecker|Deleted(redundant) (SN: 66):NW-B
```

1 2.1 2.2 2.3 3 1

Figure 6.18 Experiment 5 – Simulation Logs

The simulation logs listed in Figure 6.18 can be investigated under three main sectors. First sector is in green and the third sector is in red. Second sector is in orange and composed of three parts which are marked with 2.1, 2.2 and 2.3 respectively.

First sector shows the regular flow. `skewMaxTester` forwards two frames with SN-49 directly to `redundancyChecker` where they are either accepted or deleted. This sequence is repeated for all the frames other than the ones in between SN-50 and SN-65.

Second sector shows the sequence that is added to trigger `skewMax` behavior. In the part marked with 2.1, the first frame of SN-50 is forwarded which is coming from network-A and the one from network-B is deleted. In the part marked with 2.2, the following 14 frames from both network-A and network-B are also deleted and finally, in the part marked with 2.3 the latest frames' sequence numbers are changed from 65 to 50. It can be seen that `redundancyChecker` accepted its last frame at t = 0.054 (s) and after all the deletes, the next frame will be sent in t = 0.067 (s). Thus, a time difference of more than 10 msec is being created.

In the final sector in red, `redundancyChecker` receives the frame with SN-50 one more time after 13 msec from network-B and it accepts after the skewMax control and it got back to the regular flow.

7. MODEL PERFORMANCE EVALUATION IN REALISTIC CONDITIONS

7.1. Flight Management System Experiment

Integrated modular avionics (IMA) is investigated over a realistic AFDX case study in [56]. A flight management system (FMS) is presented and certain metrics are procured both theoretically and by modelling. After that, the same network is modeled and studied via OPNET in [27].

In the scope of this experiment, a network is created, run and documented with the help of ANCAT by using topology and characteristics used in [27] and [56]. The network created in OMNeT++ is demonstrated in Figure 7.1. ES0 block represents a module containing Keyboard Unit (KU) and Multi-Function Display (MFD) subsystems inside as well as ES1. ES2 and ES3 are Flight Manager (FM) modules. ES4 and ES5 modules contain Air Data Inertial Reference Units (ADIRU) which are gathering data from ES6 and ES7 namely Remote Data Centers (RDC). Not to mention, RDCs are connected with appropriate sensors. Finally, ES8 is Navigation Database (NDB) and it sends latitude/longitude data when requested.

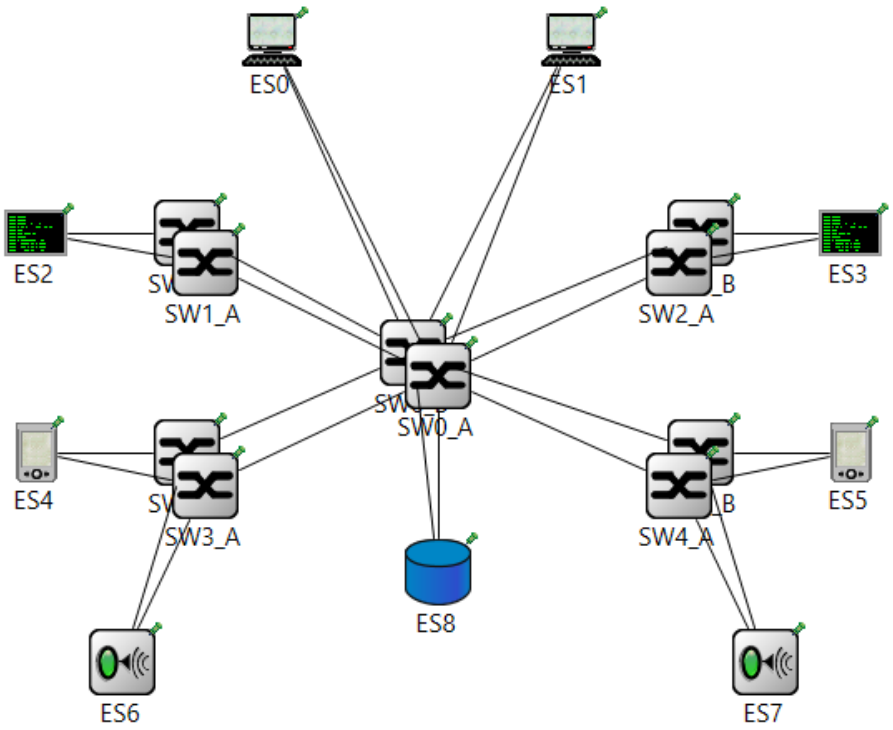

Figure 7.1 Flight Management System Network

Network characteristics that are given in Table 7.1 are gathered from the referenced works. In [56] data length is denoted with S however length of the smallest message in the mentioned thesis is 512 bits/64 bytes which is the smallest possible value of L (L and S concepts are explained in chapter 2.1.1). Since S is 20 more than L, this is not possible. Therefore, data lengths are taken as L instead of S. Moreover, length of one of the messages in [56] is 700 bits, which doesn't seem sensible because 700 bits of length means 87.5 bytes. To solve that problem, the latency calculations in [27] are investigated and it is discovered that in those calculations, message length of that particular message was taken as 800 instead of 700.

After the topology and characteristics extracted from the papers, ANCAT input excel is prepared and simulation is executed. In the paper [27], end-to-end latencies for three selected VL's and for two different types of switches were presented. The difference between switches is denoted by switch technological latency. For SW-type1 it is given as 140 usec where for SW-type2 16 usec. In Table 7.2 the results obtained from OMNeT++ AFDX model are compared to the ones obtained from OPNET in [27]. As expected, results are corroborative on behalf of AFDX OMNeT++ Model.

Table 7.1 Flight Management System Characteristics

VL-ID	Source ES	Destination ES	S (byte)	BAG (msec)	Period (msec)	Rho (Mbps)	Sigma (bits)
0x1	ES0	ES2, ES3	95	32	50	0.02375	761.1875
0x2	ES1	ES2, ES3	95	32	50	0.02375	761.1875
0x3	ES2	ES0	[145-645]	8	60	0.645	5192.25
0x4	ES2	ES8	145	16	60	0.0725	1163.625
0x5	ES3	ES1	[145-645]	8	60	0.645	5192.25
0x6	ES3	ES8	145	16	60	0.0725	1163.625
0x7	ES8	ES2	520	64	100	0.065	4163.25
0x8	ES8	ES3	520	64	100	0.065	4163.25
0x9	ES6	ES4	84	32	60	0.021	673.05
0xA	ES7	ES5	84	32	60	0.021	673.05
0xB	ES4	ES2, ES3	120	32	60	0.03	961.5
0xC	ES5	ES2, ES3	120	32	60	0.03	961.5

Table 7.2 Flight Management System Comparison of Results

VL-ID	End-to-End Latency (usec)			
	SW-type1		SW-type2	
	Safwat et al.[27]	Proposed Model	Safwat et al.[27]	Proposed Model
0x7	477	442	194	194
0x9	154	151	33	27
0xB	492	454	92	82

7.2. Commercial Avionics Architecture Experiment

A real network architecture and message set that is supplied from a commercial avionics company is investigated in [48]**.** In the scope of that thesis, mentioned architecture is implemented with both AFDX and the proposed protocol SQSDR (Shared Queue based Dynamic Slot Reservation). Implementations are conducted over OMNeT++ simulations. While SQSDR is modeled with INET framework [17], for AFDX, an improved version of the existing AFDX OMNEST model (Chapter 3.3.1) is used. The AFDX model that is used in [48] and improvements over OMNEST model for that sake are explained in Chapter 3.3.2. After the simulations are conducted, resulting end-to-end latencies and queue metrics are compared to assess the behavior of SQSDR compared to AFDX.

In this chapter, the same network is implemented with the same configurations mentioned in [48] to verify the behavior of the proposed AFDX model. Moreover, after reviewing the simulation results, network configuration is revised to improve outcomes.

The proposed network contains many avionic subsystems such as sensors, actuators, controllers and data loggers. It has 23 end-systems and 2 switches and it is modeled in Figure 7.2. First 10 end-systems are connected to Switch-0 and the remaining 13 are connected to Switch-1. Technological latencies of switch and end-systems are taken as 50 usec. Simulation is executed for 10 secs and the time that messages are started to be sent are specified randomly (`rand(0, 5ms)`). The connection assignments are determined by considering the physical placements of end-systems in the actual architecture. Similarly, all message characteristics that are listed in Table 7.3 are inherited from the real-world set-up.

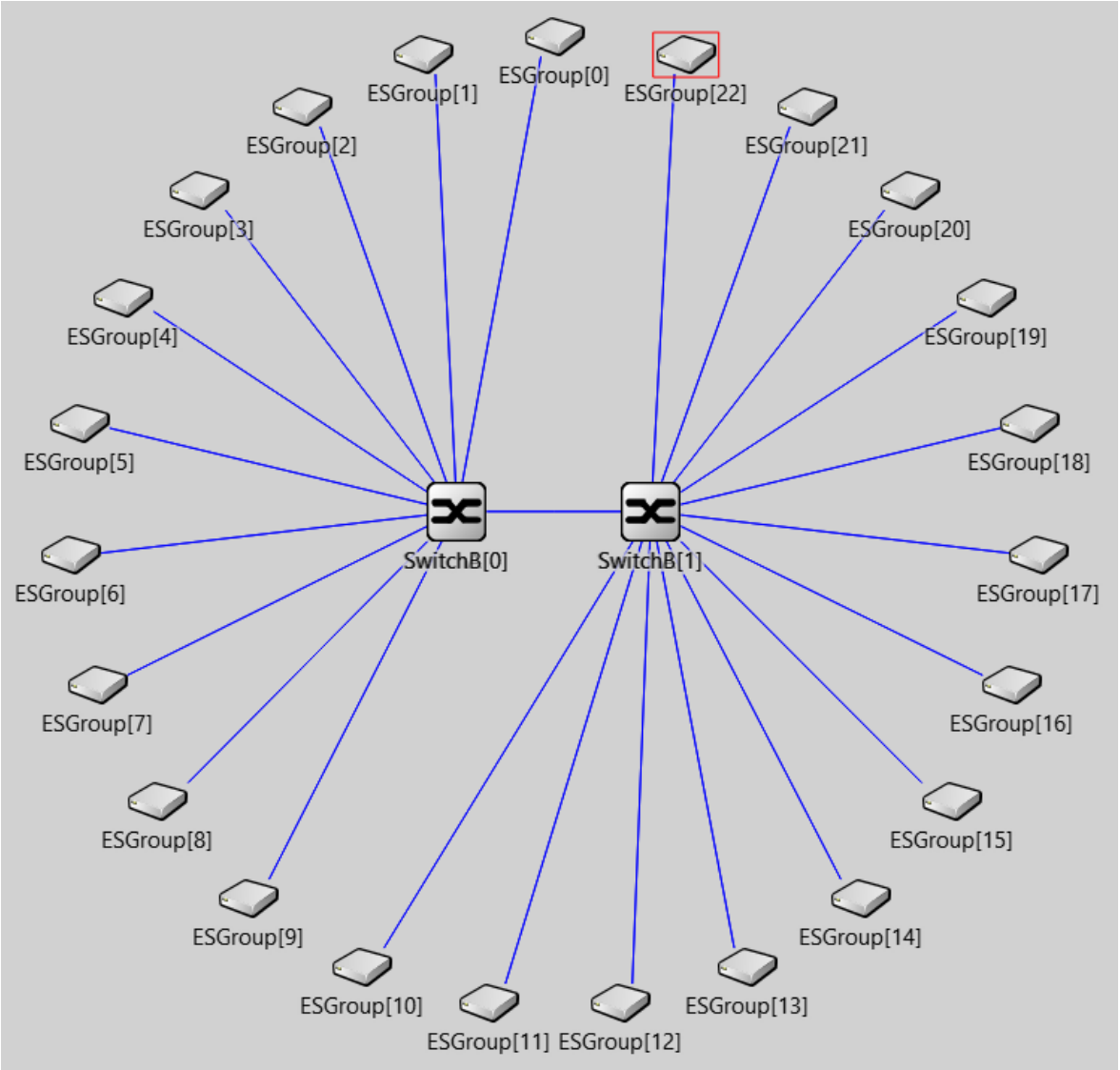

Figure 7.2 Proposed Network in [48]

Table 7.3 Message Characteristics of [48]

Periodic (P) or Sporadic (S)	VLID	Source ES	Destination ES	BAG	Period	Payload Length
P	0x1000	ES0	ES14, ES19	4ms	5ms	100
P	0x1100	ES1	ES14, ES19	4ms	5ms	200
P	0x1200	ES2	ES14, ES19	4ms	5ms	100
P	0x1300	ES3	ES14, ES19	4ms	5ms	200
P	0x1400	ES4	ES14, ES19	4ms	5ms	200
P	0x1500	ES5	ES14, ES19	4ms	5ms	200
P	0x1600	ES6	ES14, ES19	4ms	5ms	100
P	0x1700	ES7	ES14, ES19	4ms	5ms	100
P	0x1800	ES8	ES14, ES19	4ms	5ms	100
P	0x1900	ES9	ES19	1ms	5ms	1471
P	0x1900	ES9	ES19	1ms	5ms	1471
P	0x1900	ES9	ES19	1ms	5ms	1471
P	0x1900	ES9	ES19	1ms	5ms	587
P	0x2000	ES10	ES14, ES19, ES22	1ms	1ms	250
P	0x2100	ES11	ES14, ES19, ES22	1ms	1ms	750
P	0x2200	ES12	ES14, ES19, ES22	32ms	50ms	750
P	0x2300	ES13	ES14, ES19, ES22	32ms	50ms	750
S	0x2400	ES14	ES19	4ms	rand(0s, 2*5ms)	100
P	0x2500	ES15	ES14, ES19, ES22	4ms	5ms	200
P	0x2600	ES16	ES14, ES19, ES22	4ms	5ms	200
P	0x2700	ES17	ES14, ES19, ES22	4ms	5ms	100
P	0x2800	ES18	ES14, ES19, ES22	64ms	100ms	1000
S	0x2A00	ES20	ES19	1ms	rand(0s, 3*2*4.21ms)	1316
S	0x2A01	ES20	ES19	1ms	rand(0s, 3*2*4.21ms)	1316
S	0x2A02	ES20	ES19	1ms	rand(0s, 3*2*4.21ms)	1316
S	0x2B00	ES21	ES19	1ms	rand(0s, 3*2*4.21ms)	1316
S	0x2B01	ES21	ES19	1ms	rand(0s, 3*2*4.21ms)	1316
S	0x2B02	ES21	ES19	1ms	rand(0s, 3*2*4.21ms)	1316
P	0x2C00	ES22	ES14, ES15, ES19	4ms	5ms	200
P	0x2C01	ES22	ES14, ES16, ES19	4ms	5ms	200
P	0x2C02	ES22	ES14, ES17, ES19	4ms	5ms	100
P	0x2C03	ES22	ES14, ES19, ES0	4ms	5ms	100
P	0x2C04	ES22	ES14, ES19, ES1	4ms	5ms	200

Periodic (P) or Sporadic (S)	VLID	Source ES	Destination ES	BAG	Period	Payload Length
P	0x2C05	ES22	ES14, ES19, ES2	4ms	5ms	100
P	0x2C06	ES22	ES14, ES19, ES3	4ms	5ms	200
P	0x2C07	ES22	ES14, ES19, ES4	4ms	5ms	200
P	0x2C08	ES22	ES14, ES19, ES5	4ms	5ms	200
P	0x2C09	ES22	ES14, ES19, ES6	4ms	5ms	100
P	0x2C0A	ES22	ES14, ES19, ES7	4ms	5ms	100
P	0x2C0B	ES22	ES14, ES19, ES8	4ms	5ms	100
P	0x2C0C	ES22	ES14, ES18, ES19	64ms	100ms	1000

To verify the proposed AFDX model, the simulation is executed with the given topology (Figure 7.2) and message characteristics (Table 7.3). The results are compared with the ones presented in [48]. As can be seen in Table 7.4, results are consistent.

Table 7.4 End-to-End Latencies for Sporadic and Periodic Messages

		End-to-End Latencies	
		Atik[48] **Model**	**Proposed Model**
Sporadic	**Mean**	1.21 msec	1.426 msec
	Max	24.118 msec	21.133 msec
Periodic	**Mean**	0.335 msec	0.309 msec
	Max	3.72 msec	2.634 msec

In this network, there are both periodic and sporadic messages. Sporadic messages are implemented with random inter-arrival times in the previous configuration as can be seen in Table 7.3. But when looked closely, it can be noticed that the range of the random function starts with ‘0s’, which means messages can be generated at a rate faster than BAG. This is not only contradicting with the nature of sporadic messages, but also damaging the AFDX performance. A similar case is demonstrated in Scenario 3 of Experiment 1 (Chapter 6.1.3). It is explained in that chapter that messages with a period faster than BAG will overload the system. Hence the time that elapses for messages to leave the end-system will be increased due to BAG regulation. In other words, this is a design mistake. Besides, having an end-to-end latency around 20 msecs when BAG is 1 or 4 msec shows the seriousness of the problem. Although the network is quite loaded, it can be seen in the simulation results that, the queueing times in the switch for sporadic

messages are much lower than the elapsed time in the end-system (Table 7.5) which shows that end-system latency is the one having a major share in the end-to-end latency instead of the switch latency. Therefore, it is fair to say that the end-to-end latency that these messages are facing are caused by the faulty design. In addition to that, sporadic messages shall not come faster than their period. Thus, they are generated with inter-arrival times varying between the specified period and a bigger value.

Table 7.5 ES and Switch-1 Latencies for Sporadic Messages (Old Configuration)

VL-ID	**ES Latency (msec)**		**SW Latency (msec)**	
	Mean	**Max**	**Mean**	**Max**
0x2A00	0.707	5.93	0.069	0.478
0x2A01	0.659	5.586	0.069	0.449
0x2A02	0.807	8.525	0.067	0.443
0x2B00	0.742	5.993	0.067	0.443
0x2B01	0.829	7.613	0.068	0.459
0x2B02	0.774	7.661	0.069	0.43
0x2400	2.923	21.008	0.095	0.526

To make the architecture better, inter-arrival times are renewed by considering the sporadic messages and relationship between BAGs and inter-arrival times as demonstrated in Table 7.6. Moreover, small adjustments are done in message set. After this modification, end-to-end latencies in sporadic messages that are given with Table 7.8 became much smaller. Moreover, even though the queueing latencies in the switch did not get affected significantly from this improvement, the total end-system latencies decreased around 10 times (Table 7.9). For clarity, the network with these improved characteristics is denoted as "New Configuration" and the original characteristics that are used in Atik's model [48] is denoted as "Old Configuration".

Table 7.6 Modified Message Characteristics (New Configuration)

VL-ID	**BAG (msec)**	**Inter-arrival Times**	
		Old (msec)	**New (msec)**
0x2400	4	rand(0, 2*5)	rand(**5, 10**)
0x2A00, 0x2A01, 0x2A02, 0x2B00, 0x2B01, 0x2B02	1	rand(0, 3*2*4.21)	rand(**1.263, 5**)

Table 7.7 Modified Message Destination Nodes (New Configuration)

VLID	Source ES	Destination ES
0x1000 – 0x1008	ES0 – ES8	ES14, ES19, **ES22**
0x1900	ES9	ES19, **ES14**

Table 7.8 Comparison of End-to-End Latencies

		End-to-End Latencies in Proposed Model	
		Old Configuration	**New Configuration**
Sporadic	**Mean**	1.426 msec	0.419 msec
	Max	21.133 msec	1.444 msec
Periodic	**Mean**	0.309 msec	0.414 msec
	Max	2.634 msec	3.298 msec

Table 7.9 ES and Switch-1 Latencies for Sporadic Messages (New Configuration)

VL-ID	ES Latency (msec)		SW Latency (msec)	
	Mean	**Max**	**Mean**	**Max**
0x2A00	0.054	0.267	0.033	0.368
0x2A01	0.054	0.197	0.027	0.28
0x2A02	0.054	0.235	0.027	0.339
0x2B00	0.054	0.229	0.026	0.284
0x2B01	0.054	0.244	0.026	0.291
0x2B02	0.054	0.16	0.026	0.34
0x2400	0.05	0.05	0.027	0.3

In terms of switch characteristics, when the message paths are reviewed, one switch port which is connected to the ES19 attracts the attention. It represents a data logger that is designed to get all messages from all VLs which results in a highly loaded traffic and hence queueing in the switch. To get an idea about the busyness in that switch port, checking the actual bandwidth usage and maximum usable bandwidth for each message that is directed to that port (i.e., all messages) can be useful which are calculated as S (bits) per period and L (bits) per BAG respectively. For each switch port, total values are calculated by summing up individual values for each message that is using the port. Most loaded ports are listed for both new and old configurations in Table 7.10 and Table 7.11. The small difference between two configurations is resulted from newly added destination nodes. As can be seen in the tables, for both configurations maximum usable bandwidth exceeds the bandwidth that the bus can provide which is 100 Mbps. Thus, this network can

be considered as highly loaded, which should be considered in the design phase. However, the concern in this thesis is not investigation the original design but reviewing its performance. That's why, the actual bandwidths are checked which are also showing a highly loaded (almost 80%) network.

Table 7.10 Bandwidth Requirements of Most Loaded VLs (Old Configuration)

VL-ID	Ports	Actual BW Usage (Mbps)	Max Usable BW (Mbps)
All	SW1-ES19	79.09	129.02
All except sporadic	SW1-ES14	17.83	20.16
0x2500-0x2800	SW1-ES22	10.54	11.01
0x1000-0x1900, 0x2C03-0x2C0B	SW1-ES19	14.52	49.75

Table 7.11 Bandwidth Requirements of Most Loaded VLs (New Configuration)

VL-ID	Ports	Actual BW Usage (Mbps)	Max Usable BW (Mbps)
All	SW1-ES19	79.09	128.9
All except sporadic	SW1-ES14	26.26	73.37
0x2500-0x2800	SW1-ES22	13.58	14.82
0x1000-0x1900, 0x2C03-0x2C0B	SW1-ES14	14.52	49.75

When queueing times are investigated, it can be seen that they are insignificant when compared to the periods. Queueing latency plots for the mentioned end-systems are given in Figure 7.3, Figure 7.4, Figure 7.5 and Figure 7.6 and a summary containing maximum and average values are given in Table 7.12.

Table 7.12 SW Queueing Latencies for Highly Loaded Ports (New Configuration)

VL-ID	Ports	Queueing Time (msec)	
		Max	Mean
All	SW1-ES19	0.388	0.034
All except sporadic	SW1-ES14	0.146	0.012
0x2500-0x2800	SW1-ES22	0.022	0.004
0x1000-0x1900, 0x2C03-0x2C0B	SW0-SW1	0.041	0.005

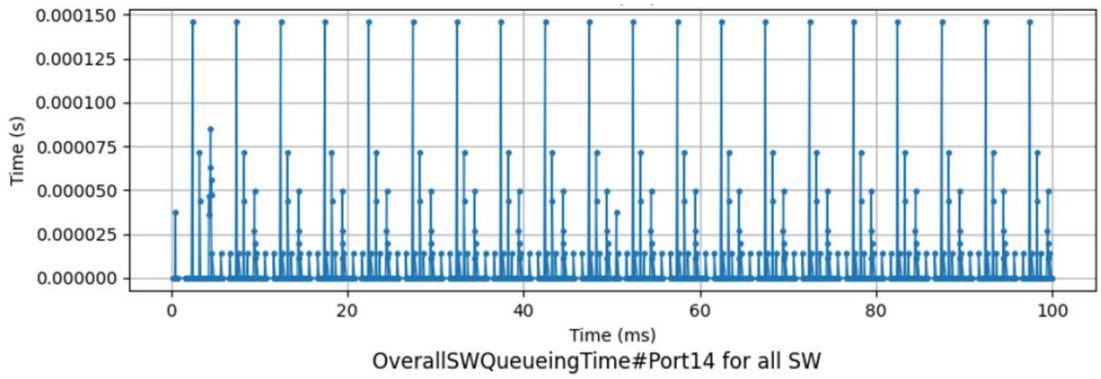

Figure 7.3 Queueing Time for SW0-ES14

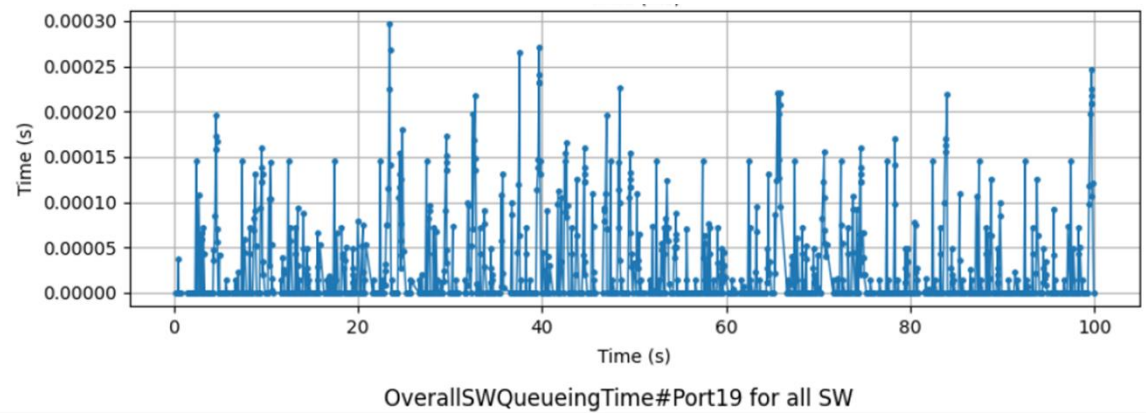

Figure 7.4 Queueing Time for SW0-ES19

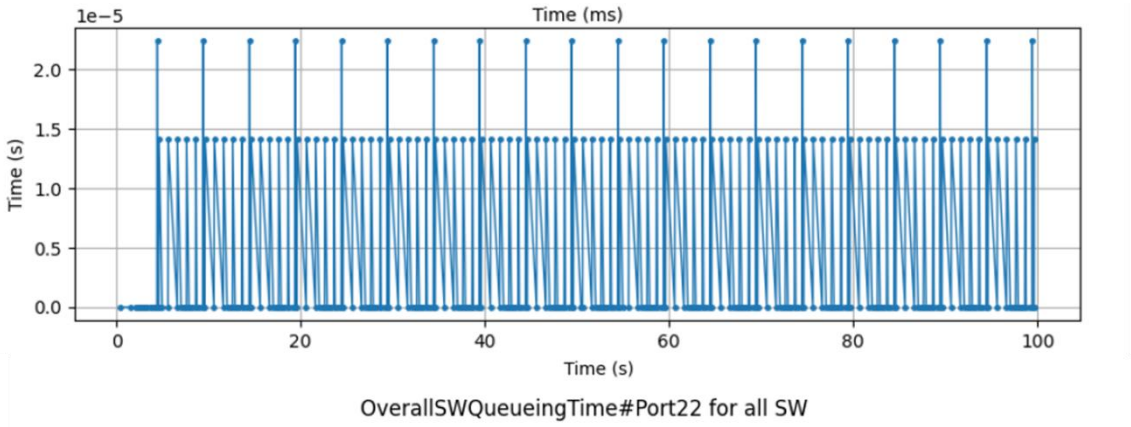

Figure 7.5 Queueing Time for SW0-ES22

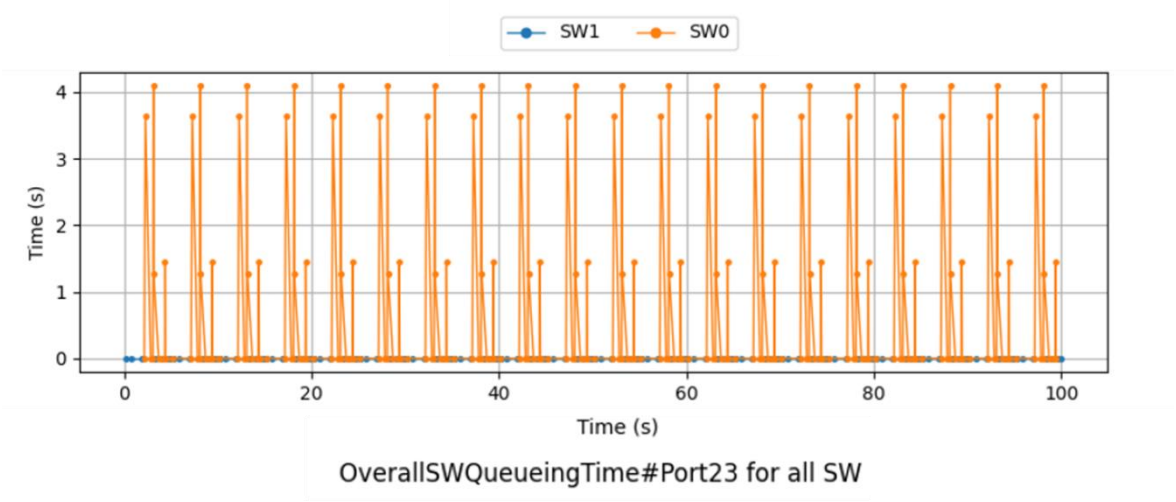

Figure 7.6 Queueing Time for SW0-SW1

Furthermore, to construct this network (Figure 7.2) in OMNeT++, a long `*.ini` file composed of almost 400 lines, is required. But with the help of the proposed network configuration and analysis tool ANCAT (Chapter 5), it is much simpler to build this network. Instead of dealing with an `*.ini` file, filling certain columns of a standard *.`xlsx` file saves a lot of time especially when repeating the tests with different characteristics.

Results show that, presented AFDX Model is consistent with the previous work, which is a real-life avionics application. In addition to that, with an improvement in the network configuration, simulation results are changed in the positive direction. With the proposed AFDX Model and ANCAT, it takes seconds to try different configurations and obtain detailed results.

7.3. Custom Network Experiment

In both realistic experiments, new network configuration and analysis tool ANCAT is used with the improved OMNeT++ AFDX Model to configure, run and analyze the network and obtained results are compared with previous works. However, in this chapter, a new network is established by combining topology of Flight Management System (Chapter 7.1) and message structure of Commercial Avionics Architecture (Chapter 7.2). By doing so, the network in Flight Management System became more loaded similar the one in the Commercial Avionics Architecture. The intention in creating such a network is to investigate a complicated, realistic AFDX network and to show capabilities of ANCAT while doing so.

In the original FMS (Flight Management System) network, there were two user interface end-systems containing keyboard and monitor, two flight managers (FM), two remote data center (RDC), two inertial reference units (ADIRU) and one navigation database. In this network, when some end-systems are sending periodic data to each other, some end-systems are working in a command-response fashion. According to the sequence, RDCs sends the data that is gathered from sensors to the ADIRUs and ADIRUs direct those data to the FM after making additional calculations. On the other hand, when a user request estimated arrival of time and distance to the target from the user interface, the responsible end-systems sends a request to the FM which results in another request from FM to the navigation database. Finally, FM calculates the requested information by combining the navigational data with sensor data that are already gathered periodically and sends to the end-system that is responsible of user interface, again periodically.

In custom network, five new end-systems are added to the existing FMS network. These are, two cameras, one data logger and two actuators. Cameras are sending sporadic video data that is split into three VLs each in order to not exceed the dedicated bandwidth of a VL. Actuators are expecting some time-critical data from FMs. Lastly, the data logger listens all messages from all VLs except from the cameras. In addition to the new end-systems, to be able to compute required time-critical data for the actuators, 4 new fast VLs that are transmitting data from RDCs to FMs are added. Moreover, again to be able to compute the data that will be sent to the actuators in time, BAG and period of the VLs between RDCs to ADIRUs are decreased. Finally, the end-system technological latency wasn’t included in the original FMS network, thus it is set to 40 us where the original switch technological latency that is equal to140 us is kept. The custom network that is created by modifying the flight management system that is given in Figure 7.1.

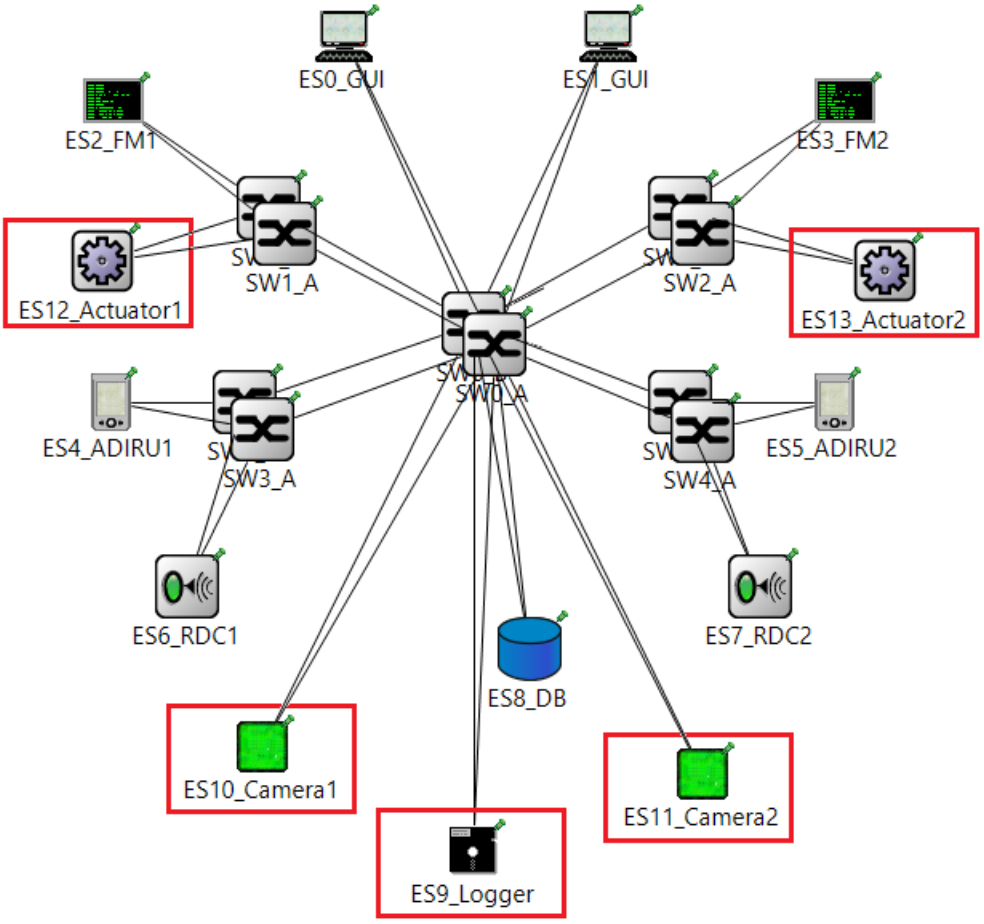

Figure 7.7 Custom Network

Table 7.13 Message Characteristics of Custom Network

Periodic (P) or Sporadic (S)	VLID	Source ES	Destination ES	BAG (msec)	Period (msec)	Payload Length
S	0x1	ES0	ES2, ES9	32	rand (50,100)	28
S	0x2	ES0	ES3, ES9	32	rand (50,100)	28
S	0x3	ES1	ES3, ES9	32	rand (50,100)	28
S	0x4	ES1	ES2, ES9	32	rand (50,100)	28
S	0x5	ES2	ES8, ES9	16	rand (60,100)	78
S	0x6	ES3	ES8, ES9	16	rand (60,100)	78
S	0x7	ES8	ES2, ES9	64	rand (100,150)	453
S	0x8	ES8	ES3, ES9	64	rand (100,150)	453
P	0x9	ES6	ES4, ES9	4	4	17
P	0xA	ES7	ES5, ES9	4	4	17
S	0x10	ES6	ES2, ES9	1	rand (1,5)	1471
S	0x11	ES6	ES3, ES9	1	rand (1,5)	1471
S	0x12	ES7	ES2, ES9	1	rand (1,5)	1471
S	0x13	ES7	ES3, ES9	1	rand (1,5)	1471
P	0x14	ES4	ES2, ES9	32	40	53
P	0x15	ES4	ES3, ES9	32	40	53
P	0x16	ES5	ES3, ES9	32	40	53
P	0x17	ES5	ES2, ES9	32	40	53
P	0x18	ES2	ES0, ES9	8	40	rand(683, 1183)
P	0x19	ES3	ES1, ES9	8	40	rand(683, 1183)
S	0x20	ES10	ES0	1	rand(1.632, 5)	1316
S	0x21	ES10	ES0	1	rand(1.632, 5)	1316
S	0x22	ES10	ES0	1	rand(1.632, 5)	1316
S	0x30	ES11	ES1	1	rand(1.632, 5)	1316
S	0x31	ES11	ES1	1	rand(1.632, 5)	1316
S	0x32	ES11	ES1	1	rand(1.632, 5)	1316
P	0xB	ES2	ES12, ES9	2	2	1471
P	0xC	ES3	ES13, ES9	2	2	1471
P	0xD	ES2	ES13, ES9	2	2	1471
P	0xE	ES3	ES12, ES9	2	2	1471

Table 7.14 Per-Port Bandwidth Requirements of Switches

Switch		VL-IDs	Actual BW Usage (Mbps)	Max Usable BW (Mbps)
ID	Port			
0	0	0x18, 0x20, 0x21, 0x22	26.53	34.44
	1	0x19, 0x30, 0x31, 0x32	26.53	34.44
	2	0x5, 0x6	0.038	0.145
	3	0x1, 0x2, 0x3, 0x4, 0x7, 0x8, 0x9, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0xB, 0xC, 0xD, 0xE	74.90	77.01
	4	0x1, 0x4, 0x7, 0x10, 0x12, 0x14, 0x17, 0xE	30.88	30.93
	5	0x2, 0x8, 0x11, 0x13, 0x15, 0x16, 0xD	30.84	30.87
1	0	0x1, 0x4, 0x7, 0x10, 0x12, 0x14, 0x17	24.73	24.78
	1	0xB, 0xE	12.30	12.30
	2	0x5, 0x18, 0xD	6.421	7.474
2	0	0x2, 0x3, 0x8, 0x11, 0x13, 0x15, 0x16	24.73	24.78
	1	0xC, 0xD	12.30	12.30
	2	0x6, 0x19, 0xE	6.421	7.474
3	0	0x9	0.168	0.168
	1	0x9, 0x10, 0x11, 0x14, 0x15	24.82	24.84
4	0	0xA	0.168	0.168
	1	0xA, 0x12, 0x13, 0x16, 0x17	24.82	24.83

In the, Table 7.13 source and destination end-systems of each VL are given with BAG and period values. Judging by this table and the scenario, the busiest destination end-system is expected to be ES9 which is the logger. Additionally, Table 7.14 contains bandwidth requirements that is binding for each port of each switch. Port-2 of SW0, which is also the one that logger is connected, it requires the maximum bandwidth due to its overload.

When simulation is executed with these inputs and final records are interpreted by ANCAT, obtained results are as follows:

No packet drops occur in the switch. As can be seen in the

1. Table 7.14, all bandwidth requirements are withing the acceptable range (smaller than100Mbps).
2. Within all five switches, SW0 is the busiest one since it is handling majority of the VLs. As can be seen in the Table 7.15, queueing latency of SW0 is significantly high than other switches.

Table 7.15 Switch Queuing Latencies

Switch	Queueing Latency (msec)		
	Mean	Min	Max
0	0.052	0	0.848
1	0.001	0	0.33
2	0.001	0	0.384
3	0.003	0	0.136
4	0.003	0	0.136

Table 7.16 Queueing Latencies for All Switches Per Port

Switch		VL-IDs	Queueing Latency (msec)		
ID	Port		Mean	Min	Max
0	0	0x18, 0x20, 0x21, 0x22	0	0	0.146
	1	0x19, 0x30, 0x31, 0x32	0	0	0.145
	8	0x5, 0x6	0	0	0.122
	9	0x1, 0x2, 0x3, 0x4, 0x7, 0x8, 0x9, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0xB, 0xC, 0xD, 0xE	0.106	0	0.848
	14	0x1, 0x4, 0x7, 0x10, 0x12, 0x14, 0x17, 0xE	0.006	0	0.238
	16	0x2, 0x8, 0x11, 0x13, 0x15, 0x16, 0xD	0.007	0	0.271
1	2	0x1, 0x4, 0x7, 0x10, 0x12, 0x14, 0x17	0.002	0	0.33
	12	0xB, 0xE	0	0	0
	14	0x5, 0x18, 0xD	0.001	0	0.111
2	3	0x2, 0x3, 0x8, 0x11, 0x13, 0x15, 0x16	0.003	0	0.384
	13	0xC, 0xD	0	0	0
	16	0x6, 0x19, 0xE	0.001	0	0.111
3	4	0x9	0	0	0
	15	0x9, 0x10, 0x11, 0x14, 0x15	0.004	0	0.136
4	5	0xA	0	0	0
	17	0xA, 0x12, 0x13, 0x16, 0x17	0.004	0	0.136

3. Maximum end-to-end latency in the system is 1.6 msec which is in 0.3% of the true mean with %95 confidence. Maximum, minimum and mean values for each VL are given in Table 7.17.

Table 7.17 End-to-end Latencies for All VLs

VL-ID	Mean (msec)	Min (msec)	Max (msec)
0x1	0.348	0.233	0.711
0x2	0.367	0.233	0.801
0x3	0.356	0.233	0.728
0x4	0.353	0.233	0.764
0x5	0.486	0.392	1.209
0x6	0.482	0.392	1.3
0x7	0.438	0.301	0.737
0x8	0.441	0.301	0.943
0x9	0.334	0.232	1.343
0xA	0.897	0.726	1.319
0x10	0.895	0.726	1.576
0x11	0.893	0.726	1.402
0x12	0.897	0.726	1.466
0x13	0.511	0.395	0.873
0x14	0.487	0.386	0.919
0x15	0.507	0.397	0.89
0x16	0.493	0.386	0.847
0x17	1.067	0.845	1.681
0x18	0.864	0.722	1.215
0x19	0.444	0.439	0.593
0x20	0.443	0.439	0.551
0x21	0.443	0.439	0.658
0x22	0.444	0.439	0.657
0x30	0.443	0.439	0.712
0x31	0.444	0.439	0.63
0x32	0.331	0.232	1.136
0xB	0.662	0.464	1.068
0xC	0.803	0.586	1.805
0xD	1.119	0.971	1.589
0xE	0.866	0.726	1.192

8. CONCLUSION AND FUTURE WORK

Due to deficiencies of previous avionics communication protocols a deterministic, dual-redundant, full-duplex, high speed ethernet protocol that provides guaranteed bandwidth, is presented and standardized as ARINC664 p7 or namely AFDX. AFDX keeps the end-to-end latency under control with the help of Bandwidth Allocation Gap (BAG) regulation and token-bucket algorithm and thus bring determinism to the system. The avionic systems using AFDX are mostly safety-critical systems with very strict timing requirements and hard real-time control loops. Thus, these systems must be investigated thoroughly in terms of latencies, utilization rates etc. to avoid unexpected setbacks later on. To make a performance analysis, the actual system can be used but it would not be possible to produce different scenarios with real subsystems. Mathematical computations can be used to foresee worst-case scenarios but they may remain incapable of giving average results. When these two options are eliminated, using a network simulation seems like the most realistic and comprehensive approach. For all these mentioned reasons, a fully functional AFDX model running in the environment is proposed in this thesis.

In order to execute an AFDX simulation, a network simulation tool is necessary. When selecting a network simulation tool, different papers that are comparing certain tools are reviewed. These papers are comparing simulation tools such as NS2, NS3, OPNET, OMNeT++, MATLAB/SIMULINK and so much more by taking price, community, language, complexity, testability, integrability, flexibility, CPU and memory usages and some other similar aspects into consideration. Although each tool has its own benefits, OMNeT++ comes forward in multiple reviews. The advantages of using OMNeT++ is that It is free for non-commercial purposes, easy to develop models, supports object-oriented programming, has an active community, has built-in libraries/example projects and open source. By considering all these aspects, OMNeT++ is preferred.

To develop the mentioned AFDX model, an existing one is used as a base. By fixing missing aspects and running numerous test scenarios, the model becomes fully AFDX compliant. The contribution of this thesis is not just producing a good simulation tool but also making it easy to configure without the need of re-compiling the code. Therefore, every feature and configuration parameter that may be needed to be changed by the user become accessible from one `*.ini` file. This includes parameters like cable length, BAG,

period, message length, data rate of message source etc. but also aspects like network topology.

In addition to the ready-to-use, realistic and easy to configure AFDX model, a network configuration and analysis tool named ANCAT is proposed. For those who are not familiar with OMNeT++ or for those who does not want to deal with the process of changing the `*.ini` file, running the simulation and reviewing the results in the OMNeT++ environment, ANCAT is a game changer. It expects an excel file in a certain format containing information about the network topology, general system settings and message VL configurations. It creates an `*.ini` file and VL-routing table(s) accordingly. After that, it runs the simulation with generated files. Finally, it creates a report by analyzing the simulation output files (`*.vci`, `*.vec`). In order to complete its missions, ANCAT only needs a configuration excel and certain file paths such as OMNeT++ setup folder or AFDX simulation folder.

By using AFDX simulation model and ANCAT, many experiments are conducted. Some of them are artificial experiment with easily predictable results such as, BAG regulation assessment, jitter measurement, account and queue management and latencies in the switch, skew max control. For those experiments, the expectations and outcomes are explained and compared clearly. In addition to that, realistic experiments are executed. For those, some realistic topologies and message sets are gathered from other thesis and papers then they are executed with the proposed AFDX model. After that, obtained simulation results are compared with the original works. Both artificial and realistic experiments fulfill the expectations. Thus proposed AFDX model and ANCAT are verified.

To expend this thesis, followings can be done:

- All queues in the simulation model are FIFO (First In, First Out) queues. Thus, scheduling in the end-system and switch port are FIFO scheduling. In fact, different type of scheduling algorithms are provided in the queueinglib. Option to select different scheduling algorithms can be provided to the user. [8]
- Frame Filtering functionality of an AFDX end-system was not implemented in the legacy project and not handled in the scope of this work. The test scenarios are executed with valid frame sequences hence not having frame filtering doesn't

affect realistic features of this simulation. But it can be added to filter-out invalid frames.

- Ethernet layers other than MAC are not handled in the presented simulation because the only concern was AFDX protocol itself. But it can be useful for community so it should be added in the future. In addition to that merging the proposed AFDX simulation with INET framework, can make adding other layers much easier and can also provide more to the community.
- ANCAT is not covering faulty scenarios and does not handle the errors occurred during the simulation run. It can check the excel file if all inputs are written correctly and report the errors thrown from the OMNeT++.
- ANCAT is not capable of identifying design errors. It can be edited to detect some design errors and prompt warning.
- ANCAT can be modified to take inputs from user interface instead of an xlsx file.
- Simulation results can be verified further. Setting up a real AFDX network and comparing its outcomes with simulation results would justify this simulation for sure.

REFERENCES

[1] H. Xu, “Design of Avionics Integration Architecture and Data Network for A-17 ”ZEPHYR",” 2017. doi: 10.13140/RG.2.2.26673.12649.

[2] T. Gaska, C. Watkin, and Y. Chen, “Integrated modular avionics - Past, present, and future,” *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 9, pp. 12–23, 2015, doi: 10.1109/MAES.2015.150014.

[3] R. L. Alena, J. P. Ossenfort IV, K. I. Laws, A. Goforth, and F. Figueroa, “Communications for Integrated Modular Avionics,” in *IEEE Aerospace Conference Proceedings*, 2007, pp. 1–18. doi: 10.1109/AERO.2007.352639.

[4] Airlines Electronic Engineering Committee, “ARINC 664 P7-1: Aircraft Data Network Part 7 Avionics Full-Duplex Switched Ethernet Network,” *ARINC 664 Specification*. 2009.

[5] “AFDX®/ARINC664P7 Tutorial - What is AFDX - AIM Online.” https://www.aim-online.com/products-overview/tutorials/afdx-arinc664p7-tutorial/ (accessed Apr. 01, 2022).

[6] Q. Xu and X. Yang, “Performance evaluation on packet transmission for distributed real-time avionics networks using forward end-to-end delay analysis,” *Trans Jpn Soc Aeronaut Space Sci*, vol. 64, no. 1, pp. 1–12, 2021, doi: 10.2322/TJSASS.64.1.

[7] H. Bauer, J. L. Scharbarg, and C. Fraboul, “Improving the worst-case delay analysis of an AFDX network using an optimized Trajectory approach,” *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 521–533, 2010, doi: 10.1109/TII.2010.2055877.

[8] Y. Tian, Z. Ma, and S. Zhou, “Analysis of AFDX network delay based on NS2,” *Journal of Physics: Conference Series*, vol. 2026, no. 1, p. 012010, 2021, doi: 10.1088/1742-6596/2026/1/012010.

[9] D. Song, X. Zeng, L. Ding, and Q. Hu, “The design and implementation of the AFDX network simulation system,” in *2010 International Conference on Multimedia Technology, ICMT 2010*, 2010, vol. 2, pp. 2–5. doi: 10.1109/ICMULT.2010.5629728.

[10] X. Jiqiang, “Study on Real-time Performance of AFDX Using OPNET,” in *2011 International Conference on Control, Automation and Systems Engineering (CASE)*, 2011, pp. 1–5. doi: 10.1109/ICCASE.2011.5997784.

[11] E. Weingärtner, H. vom Lehn, and K. Wehrle, “Performance Comparison of Recent Network Simulators,” in *IEEE International Conference on Communications*, 2009, pp. 1–5. doi: 10.1109/ICC.2009.5198657.

[12] A. Zarrad and I. Alsmadi, “Evaluating network test scenarios for network simulators systems,” *International Journal of Distributed Sensor Networks*, vol. 13, no. 10, pp. 1–17, 2017, doi: 10.1177/1550147717738216.

[13] Q. Yang, H. Lu, and X. Tu, “Simulation and Experiment of AFDX Network Based on OMNeT++,” in *Proceedings - 2020 Chinese Automation Congress, CAC 2020*, 2020, pp. 5849–5854. doi: 10.1109/CAC51589.2020.9326633.

[14] “OMNeT++ Discrete Event Simulator.” https://omnetpp.org/ (accessed Apr. 02, 2022).

[15] T. Steinbach, H. Dieumo Kenfack, F. Korf, and T. Schmidt, “An Extension of the OMNeT++ INET Framework for Simulating Real-time Ethernet with High Accuracy,” in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, 2011, pp. 375–382. doi: 10.4108/icst.simutools.2011.245510.

[16] N. Rejeb, A. K. ben Salem, and S. ben Saoud, “AFDX simulation based on TTEthernet model under OMNeT++,” in *Proceedings of International Conference on Advanced Systems and Electric Technologies, IC_ASET 2017*, 2017, pp. 423–429. doi: 10.1109/ASET.2017.7983731.

[17] “inet-framework/inet: INET Framework for the OMNeT++ discrete event simulator,” 2022. https://github.com/inet-framework/inet (accessed Apr. 01, 2022).

[18] “OMNEST - Performance Modeling Case Studies.” https://omnest.com/casestudy-afdx.php (accessed Apr. 19, 2022).

[19] “omnetpp-models/afdx: Avionics Full-Duplex Switched Ethernet model for OMNeT++,” 2022. https://github.com/omnetpp-models/afdx (accessed Apr. 01, 2022).

[20] “badapplexx/AFDX,” 2022. https://github.com/badapplexx/AFDX (accessed Apr. 02, 2022).

[21] F. Molina, P. Corral, M. Aljaro, G. de Scals, and A. Rodriguez, “Implementation of an AFDX interface with Zynq SoC Board in FPGA,” *Elektronika ir Elektrotechnika*, vol. 26, no. 5, pp. 11–15, 2020, doi: 10.5755/J01.EIE.26.5.26008.

[22] H. Charara, J. L. Scharbarg, J. Ermont, and C. Fraboul, “Methods for bounding end-to-end delays on an AFDX network,” *Proceedings - Euromicro Conference on Real-Time Systems*, vol. 2006, no. August, pp. 193–202, 2006, doi: 10.1109/ECRTS.2006.15.

[23] F. He, L. Zhao, and E. Li, “Impact analysis of flow shaping in ethernet-AVB/TSN and AFDX from network calculus and simulation perspective,” *Sensors (Switzerland)*, vol. 17, no. 5, 2017, doi: 10.3390/s17051181.

[24] P. M. Vdovin and V. A. Kostenko, “Organizing message transmission in AFDX networks,” *Programming and Computer Software*, vol. 43, no. 1, pp. 1–12, 2017, doi: 10.1134/S0361768817010078.

[25] X. Liu, Z. Du, and K. Lu, “Modeling and Simulation of Avionics Full Duplex Switched Ethernet(AFDX Network) Based on OPNET,” *Atlatntis Highlights in Enginnering*, vol. 3, no. Jimec 2018, pp. 1–4, 2018, doi: 10.2991/jimec-18.2018.65.

[26] T. Ricker, “Avionics Bus Technology: Which Bus Should I Get On?,” in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, 2017, pp. 1–12. doi: 10.1109/DASC.2017.8102152.

[27] N. E. D. Safwat, A. Zekry, and M. Abouelatta, “Avionics Full-duplex switched Ethernet (AFDX): Modeling and simulation,” in *National Radio Science Conference, NRSC, Proceedings*, 2015, vol. 2015-June, pp. 286–296. doi: 10.1109/NRSC.2015.7117841.

[28] N. El-Din Safwat, M. A. El-Dakroury, and A. Zekry, “The Evolution of Aircraft Data Networks,” *International Journal of Computer Applications*, vol. 94, no. 11, pp. 27–32, 2014, doi: 10.5120/16389-5968.

[29] C. Suthaputchakun, Z. Sun, C. Kavadias, and P. Ricco, “Performance analysis of AFDX switch for space onboard data networks,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 4, pp. 1714–1727, 2016, doi: 10.1109/TAES.2016.150304.

[30] Actel, “Developing AFDX Solutions,” no. March. pp. 1–18, 2005.

[31] B. Annighoefer, H. Ihle, and F. Thielecke, “An easy-to-use real-time AFDX simulation framework,” *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, vol. 2016-Decem, pp. 1–9, 2016, doi: 10.1109/DASC.2016.7778027.

[32] T. Hamza, J. L. Scharbarg, and C. Fraboul, “Priority assignment on an avionics switched Ethernet Network (QoS AFDX),” *IEEE International Workshop on Factory Communication Systems - Proceedings, WFCS*, 2014, doi: 10.1109/WFCS.2014.6837580.

[33] Y. Hua and X. Liu, “Scheduling design and analysis for end-to-end heterogeneous flows in an avionics network,” *Proceedings - IEEE INFOCOM*, pp. 2417–2425, 2011, doi: 10.1109/INFCOM.2011.5935062.

[34] J. D. C. Little, “A Proof for the Queuing Formula: L= λ W,” *Operations Research*, vol. 9, no. 3, pp. 383–387, May 1961, [Online]. Available: http://www.jstor.org/stable/167570

[35] D. Simchi-Levi and M. A. Trick, “Introduction to ‘little’s law as viewed on its 50th anniversary,’” *Operations Research*, vol. 59, no. 3, p. 535, 2011, doi: 10.1287/opre.1110.0941.

[36] F. M. Dekking, C. Kraaikamp, H. P. Lopuhaä, and L. E. Meester, “A Modern Introduction to Probability and Statistics,” 2005, doi: 10.1007/1-84628-168-7.

[37] A. Varga and R. Hornig, “An overview of the OMNeT++ simulation environment,” *SIMUTools 2008 - 1st International ICST Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, 2008, doi: 10.4108/ICST.SIMUTOOLS2008.3027.

[38] “lidongming/mf-opp4: Mobility Framework for OMNeT++ 4.” https://github.com/lidongming/mf-opp4 (accessed Apr. 09, 2022).

[39] X. Xian, W. Shi, and H. Huang, “Comparison of OMNET++ and other simulator for WSN simulation,” *2008 3rd IEEE Conference on Industrial Electronics and Applications, ICIEA 2008*, pp. 1439–1443, 2008, doi: 10.1109/ICIEA.2008.4582757.

[40] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley, “ns-3 Project Goals.” 2006.

[41] H. Charara and C. Fraboul, “Modelling and simulation of an avionics full duplex Switched Ethernet,” in *Proceedings - Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/E-Learning on Telecommunications Workshop AICT/SAPIR/ELETE 2005*, 2005, pp. 207–212. doi: 10.1109/AICT.2005.58.

[42] M. Veran and D. Potier, “QNAP 2: A Portable environment for Queueing Systems Modelling,” *Modelling Techniques and Tools for Performance Analysis*, pp. 5–24, 1984.

[43] P. T. Jean-Yves Boudec, *Network Calculus*, vol. 2050. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. doi: 10.1007/3-540-45318-0.

[44] L. Fernandez-Olmos, F. Burrull, and P. Pavon-Marino, “Net2Plan-AFDX: An open-source tool for optimization and performance evaluation of AFDX networks,” *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, vol. 2016-Decem, pp. 1–7, 2016, doi: 10.1109/DASC.2016.7778026.

[45] H. Bauer, J.-L. Scharbarg, and C. Fraboul, “Applying Trajectory approach to AFDX avionics network,” 2009.

[46] H. Charara, J. L. Scharbarg, J. Ermont, and C. Fraboul, “Methods for bounding end-to-end delays on an AFDX network,” *Proceedings - Euromicro Conference on Real-Time Systems*, vol. 2006, pp. 193–202, 2006, doi: 10.1109/ECRTS.2006.15.

[47] H. Bauer, J. L. Scharbarg, and C. Fraboul, “Applying and optimizing trajectory approach for performance evaluation of AFDX avionics network,” *ETFA 2009 - 2009 IEEE Conference on Emerging Technologies and Factory Automation*, 2009, doi: 10.1109/ETFA.2009.5347083.

[48] E. Atik, “A new fault tolerant real-time ethernet protocol: desing and evaluation,” 2021. [Online]. Available: https://open.metu.edu.tr/handle/11511/89745

[49] “omnetpp-models/queueing: A general queueing library for the OMNeT++ simulator.” https://github.com/omnetpp-models/queueing (accessed Apr. 19, 2022).

[50] E. Gamma, R. Helm, R. Johnson, and V. John, *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995.

[51] “OMNeT++ 6.0 Available,” 2022. https://omnetpp.org/software/2022/04/13/omnet-6-released.html (accessed May 14, 2022).

[52] “omnetpp/samples/queueinglib at master · omnetpp/omnetpp.” https://github.com/omnetpp/omnetpp/tree/master/samples/queueinglib (accessed May 16, 2022).

[53] “omnetpp/samples/queueinglibext at master · omnetpp/omnetpp.” https://github.com/omnetpp/omnetpp/tree/master/samples/queueinglibext (accessed May 15, 2022).

[54] “EtherLink.” https://doc.omnetpp.org/inet/api-current/neddoc/inet.node.ethernet.EtherLink.html (accessed May 15, 2022).

[55] C. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, “Dijkstra’s algorithm,” in *Introduction to Algorithms*, 2001, pp. 595–601.

[56] M. Lauer, “Une méthode globale pour la vérification d’exigences temps réel : application à l’Avionique Modulaire Intégrée,” Institut National Polytechnique de Toulouse, 2012.