ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

OMNET++ SIMULATION MODEL FOR INTEGRATED MODULAR AVIONICS

M.Sc. THESIS

Mümin Göker GAYRETLİ

Department of Defence Technologies

Defence Technologies Programme

June 2023

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

OMNET++ SIMULATION MODEL FOR INTEGRATED MODULAR AVIONICS

M.Sc. THESIS

Mümin Göker GAYRETLİ
(514201018)

Department of Defence Technologies

Defence Technologies Programme

Thesis Advisor: Asst. Prof. Ramazan YENİÇERİ

June 2023

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

ENTEGRE MODÜLER AVİYONİKLER İÇİN OMNET++ SİMULASYON MODELİ

YÜKSEK LİSANS TEZİ

Mümin Göker GAYRETLİ
(514201018)

Savunma Teknolojileri Anabilim Dalı

Savunma Teknolojileri Programı

Tez Danışmanı: Asst. Prof. Ramazan YENİÇERİ

Haziran 2023

Mümin Göker GAYRETLİ, a M.Sc. student of ITU Graduate School student ID 514201018, successfully defended the thesis entitled "OMNET++ SIMULATION MODEL FOR INTEGRATED MODULAR AVIONICS", which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Asst. Prof. Ramazan YENİÇERİ** İstanbul Technical University	.....
Jury Members :	**Prof. Dr. Sıddıka Berna ÖRS YALÇIN** Istanbul Technical University	.....
	Dr. Muhammet Selim DEMİR TÜBİTAK BİLGEM	.....

Date of Submission : 23 May 2023
Date of Defense : 14 June 2023

To my spouse and family,

FOREWORD

During the initial stages of my thesis, I dedicated considerable time to reading and exploring various subjects. It was challenging to find a topic that aligned with my interests and expertise. Fortunately, with the invaluable assistance of my colleagues at TUBITAK and Assist. Prof. Ramazan YENİÇERİ, we identified a thesis topic that revolves around captivating domains such as simulation, operating systems, and networking. Once I delved into the subject and began my work, I was captivated by the interconnections between the three seemingly disparate fields of simulation, operating systems, and networking. This realization not only piqued my curiosity but also intensified my motivation to explore further and unravel the intricate relationships among them. In addition to the technical contributions of the thesis process, I have gained valuable management skills on how to plan, execute, and conclude an academic study.

Firstly, I have been deeply grateful to Assist. Prof. Ramazan YENİÇERİ for their continuous support and guidance in my academic journey since my undergraduate years. His significant contributions have been instrumental in the completion of this thesis. Also, I would like to extend my gratitude to the TUBITAK institution and all my colleagues for their support and collaboration throughout this thesis. However, I would like to extend my gratitude to my colleagues, Dr. M. Selim Demir, Dr. İbrahim Hökelek, and Yunus Yılmazer, for being there for me throughout the thesis process, providing assistance and guidance whenever I needed it. Furthermore, I would like to thank my family, who have always made me feel their special support and care throughout my life. Finally, I would like to express my heartfelt gratitude to my spouse who, throughout the thesis process, has been there to hold my hand and lift me up whenever I felt tired and demoralized. Her unwavering support and constant presence by my side have been invaluable.

June 2023

Mümin Göker GAYRETLİ
(Computer Engineer)

TABLE OF CONTENTS

ABBREVIATIONS

IMA	**:** Integrated Modular Avionic
JIAWG	**:** Joint Integrated Avionics Working Group
ARINC	**:** Aeronautical Radio, Incorporated
AFDX	**:** Avionics Full-Duplex Switched Ethernet
APEX	**:** Application/Executive
MTF	**:** Major Time Frame
O/S	**:** Operating System
QoS	**:** Quality of Service
OOP	**:** Object-Oriented Programming
CIRA	**:** Italian Aerospace Research Centre
NS	**:** Network Simulator
INET	**:** Station
TCP	**:** Transmission Control Protocol
IP	**:** Internet Protocol
ES	**:** End System
VL	**:** Virtual Link
DES	**:** Discrete Event Simulator
FES	**:** Future Event Set
TTEthernet	**:** Time-Triggered Ethernet
TSN	**:** Time-Sensitive Networking

SYMBOLS

UPPALL	**:** Integrated Tool Environment
SIMULINK	**:** A commercial simulator
OPNet	**:** Commercial network simulator
NS-2	**:** Open-source network simulator
NS-3	**:** Open-source network simulator
OMNET++	**:** Open-source network simulator
INET	**:** A framework developed for OMNET++
ms	**:** millisecond

LIST OF TABLES

Page

LIST OF FIGURES

OMNET++ SIMULATION MODEL FOR INTEGRATED MODULAR AVIONICS

SUMMARY

As the number and variety of electronic devices in aircraft continue to grow, the traditional federated architecture needs to be revised to meet these vehicles' size, weight, and power (SWaP) constraints. Integrated Modular Avionics (IMA) architecture has emerged as a promising solution for the SWaP problems. The IMA architecture optimizes the utilization of size, weight, and power by centralizing multiple application tasks onto a single hardware platform. When developing an IMA system, it is crucial to consider the relevant standards. ARINC 653 and ARINC 664 P7 (also known as AFDX) are two prominent standards that have garnered considerable attention and recognition within this context. However, these standards have numerous configuration parameters and offer various design options for engineers and designers. Therefore, determining the system configuration for optimal network performance is complex. In this regard, performing a significant portion of the IMA system design process in a simulation environment can efficiently conserve limited resources, including time and finances. This thesis proposes a simulation model of the IMA system to solve these issues.

It is neither logical nor necessary to simulate all the rules defined by ARINC 653 and AFDX standards to measure the network performance of applications. Therefore, the first step of the thesis is to develop a system model of ARINC 653 concepts and AFDX devices to identify the necessary components used to measure communication performance. It is necessary to design components such as *partition*, *partition manager*, *process*, and *process manager* to manage avionic tasks according to ARINC 653 standard. The role of the *partition manager* component is to handle the initiation and termination operations of the required *partition* components based on the Major Time Frame (MTF). On the other hand, the *partition* component encompasses sub-components, including the *process manager* and *process*. Furthermore, the *partition* should relay the initiation and termination requests it receives to the *process manager*. As for the *process manager*, it executes the operations of stopping or starting *process* according to the received requests. In addition to managing avionic tasks, it is also necessary to have communication between the *partitions* to measure the system's communication performance. Therefore, it is essential to develop components that perform the sampling and queuing communication modes defined by the ARINC 653 standard.

Two devices must be modeled for the AFDX standard: the End System (ES) and the Switch. The ES serves as the network device for communication between *processes* in the network. When a *process* wants to send a message, it writes the message to the sender communication ports within the ES. After writing the message, the

device applies techniques such as BAG, data packet size compatibility, and redundancy management mechanisms to the packet before sending it to the network. On the receiving side, the ES receives the message from the physical link and performs operations like integrity checking and redundancy management. Then, the packet is written to the appropriate receiver communication port. When the receiver *process* is activated, it can read the message from the port. The switch device connects the ES in the network and performs *filtering*, *policing*, and *switching* tasks. While *filtering* ensures that packets comply with data packet size limitations, *policing* checks adherence to BAG rules. Also, *switching* determines the appropriate output ports for incoming packets.

The devices and structures developed in the system model must be converted to the simulation model in a simulation environment. OMNET++ offers better scalability and extensibility compared to other simulation environments. Additionally, we can benefit from an active community, open-source code, and frameworks like INET that provide a detailed implementation of the OSI layer. That is why OMNET++ has been chosen as the environment for implementing a simulation model of the developed system model. In addition to the *partition*, *partition manager*, *process*, and *process manager* components specified in the system model, two additional components have also been developed for the simulation model of the ARINC 653 concepts: *port channel* and *network transmitter driver*. The *port channel* connects the receiver *process* to the receiver communication port of the ES, while the *network transmitter driver* is responsible for writing the data packets sent by the sender *process* to the correct sender communication port within the ES.

For the AFDX device, a previously developed model [1] has been utilized. The same model from the paper has been used for the switch device. However, additional enhancements are required for the ES. In the existing model, the ES was designed solely for performing measurements at the device level, so no sender and receiver *communication ports* were designed. In addition to *communication ports*, the design does not include the *demux* component, which is responsible for writing messages to the appropriate receiver communication ports. The newly developed simulation model can handle packet reception and transmission operations by adding these two components to the previous simulation model.

The developed simulation model is tested to determine their capability to handle packet reception and transmission tasks successfully. For this purpose, a scenario was created in the network model, consisting of two IMA modules referred to as sender and receiver. The sender module generates data packets and transmits them to the receiver module. The message integrity of the received data by the *process* of the receiver module is observed on the simulation console, confirming the successful execution of packet reception and transmission operations.

It is essential to test not only the packet reception and transmission operations but also the timing of component executions to ensure proper functioning. A more advanced scenario from a previous study has been utilized [2]. Initially, the theoretical delays of the Virtual Link (VL) in the scenario are calculated using the application delay formula that includes software-based overheads. Then, the application delays are obtained through simulation, and it is observed that they converge with the theoretical

values. Furthermore, the sender application in VL 1 of the scenario sends two data packets without any time gaps. However, since the design of the receiver and sender applications did not consider this, the current BAG value of the VL cannot prevent packet loss. The expectation is that reducing packet loss can be achieved by increasing the BAG value to match the period of the receiver application. This relation has been confirmed through testing with various BAG values. The consistency between the theoretical and simulated application delays and the expected relationship between BAG and packet loss indicates that the components are executed at the correct timing.

The system design has been accurately transferred to the simulation environment, allowing for performance measurement in various IMA scenarios. Future developments can explore advanced technologies, such as using devices that support Time-Triggered Ethernet (TTEthernet) and Time-Sensitive Networking (TSN) instead of AFDX-compliant devices. Additionally, the ARINC 653 standard model has been designed to model a single processor system, but it can be further enhanced to enable parallel execution of applications and conduct complex measurements.

ENTEGRE MODÜLER AVİYONİKLER İÇİN
OMNET++ SİMULASYON MODELİ

ÖZET

İlk icat edilen transistörlerin boyutları oldukça büyük olduğu için bu transistörler ile oluşturulan entegre devreler ve bilgisayarlar hava araçlarının boyut ve ağırlık gibi gereksinimlerini karşılayamamaktaydılar. Ancak gün geçtikçe bir entgere devre üzerine yerleştirilebilecek tümdevre sayısının sürekli artması hava araçlarında kullanılabilecek avionik sistemlerin çeşitliliği arttırmaktadır. Ancak avionik sistemlerin çeşitliliğinin artması bölümün başında bahsedilen boyut ve ağırlık gibi gereksinimlerin sağlanmasında tekrardan sıkıntılara neden olmuştur. Bu noktada Entgre Modüler Aviyonik (Integrated Modular Avionics - IMA) mimarisi bu probleme bir çözüm olarak Ortak Entegre Aviyonik Çalışma Grubu (Joint Integrated Avionics Working Group - JIAWG) tarafından ilk olarak F-22 savaş uçağı programı kapsamında önerilmiştir. IMA farklı kritikliğe sahip aviyonik görev bilgisayarlarının yaptığı işleri tek bir bilgisayar tarafından yapılmasına olanak sağlayan bir mimaridir.

IMA mimarisinin bir uçakta tam anlamıyla sağlanabilmesi bir çok farklı standard geliştirilmiştir. Bunlardan ikisi ARINC 653 ve ARINC 664 P7 (Avionics Full-Duplex Switched Ethernet : AFDX) standardlarıdır. ARINC 653 standardı, aviyonik görev bilgisayar üzerinde kullanılmak üzere tasarlanan Gerçek Zamanlı İşletim Sisteminin (GZIS) kurallarını ve servislerini tanımlar. GZIS sayesinde de donanım üzerindeki hesaplama birimleri ve Giriş/Çıkış (G/Ç) cihazları gibi kaynaklar aviyonik uygulamalara paylaştırılabilir. Bu paylaşım *bölümleme (partitioning)* adındaki bir yöntem ile gerçekleştirilir. ARINC 653, donanımın ortak kaynakları kullanan her bir uygulamayı *bölme (partition)* olarak adlandırır. *Bölmeler* Ana Zaman Yapısı (Major Time Frame : MTF) olarak adlandırılan zaman akışı üzerine yerleştirilir. Her bir *bölme*, MTF üzerinde Bölme Zaman Çerçevesi (PTW : Partition Time Window) olarak isimlendirilen bir zaman aralığına sahiptir ve donanım üzerindeki kaynaklara sadece bu zaman aralığında erişebilir. *Bölme*, sadece sanal bir entiti olduğu için sensör verisi okuma ve motor kontrolü gibi görevleri gerçekleştiremez. Bu tarz görevler *işlem (process)* adı verilen çalışma birimleri tarafından gerçekleştirilir. Her *bölmenin* kendine ait *işlemleri* mevcuttur ve bu *işlemler* farklı kritiklik değerlerine sahiptir. Zaman, PTW'nun MTF üzerindeki başlangıç noktasına ulaştığında, *bölmedeki* en yüksek kritikliğe sahip *işlem* donanımın kaynaklarına erişmek üzere *işlem yöneticisi* tarafından seçilir ve çalıştırılır.

AFDX standardı temel olarak, IMA sistemindeki ağ cihazlarının band genişliğinin uygulamalara paylaştırılmasında kullanılan yöntem ve kuralları tanımlar. Bir gönderici aviyonik uygulama, bir veya birden fazla alıcı aviyonik uygulamaya aynı veriyi iletebilir. Bu veri iletiminde kullanılan uygulamaların oluşturduğu alt ağ, sanal hat (VL : Virtual Link) olarak isimlendirilir. AFDX, VL'de gönderici uygulamanın zaman

farkı olmaksızın art arda paket yollamasını önleyen Band genişliği Tahsis Aralığı (Bandwidth Allocation Gap : BAG) kuralını tanımlar. Bu kurala göre bir VL kaynak uygulamasının gönderdiği iki veri paketi arasında geçmesi gereken minimum bir süre vardır. Aviyonik işlem bu kurala uymadan veri paketi gönderse dahi Uç Sistem (End System : ES) ve *Anahatar* olarak adlandırılan ve AFDX'in kurallarının uygulanmasını sağlayan ağ cihazları bu veri paketlerini donanım üzerinde depolar ve BAG kuralını sağlayana kadar bekletir. Bu kural sağlandıktan sonra paketler alıcı uygulamalara gönderilmek üzere ağa gönderilir. Bunun yanı sıra ağda iletilen her bir veri paketi boyutunun maksimum (S_{max}) ve minimum (S_{min}) değerleride belirlenmelidir. Bu veri boyutu aralığını ihlal eden paketler ES ve Anahtar tarafından düşürülmektedir. Bu iki kural, bir VL'in band genişliğini tanımlar.

Bahsedilenlere ek olarak, ARINC 653 ve AFDX standardları kullanıldığı sistemlerde daha bir çok kuralın ve parametrenin tanımlanmasına neden olur. Bu durum sistem tasarımındaki değişken sayısı arttırmakta ve optimum ağ performansını verecek sistem tasarımını ve parametreleri bulmayı zorlaştırmaktadır. Fiziksel testlerin fazlalığı ise projelerin maliyetini arttırır ve teslim tarihlerinde gecikmelere neden olur. Bu sorunları çözmek adına tasarlanan IMA modeli simulasyon ortamında gerçeklenebilir. Böylelikle mühendisler daha fazla denemeyi daha kısa bir zamanda gerçekleştirerek ürünlerin pazara sürüm süresini kısaltabilirler. Bunlara ek olarak sisteme yeni bir donanım eklemek istediklerinde donanımı satın almak yerine simulasyon ortamındaki sisteme donanımın modelini ekleyebilirler. Simulasyon ortamında yeterli bir gelişme görülürse donanım fiziksel olarak alınabilir. Bu tezde, ARINC 653 ve AFDX standardlarını sağlayan IMA sisteminin bir benztimi OMNET++ simulatoru üzerinde gerçeklenmiştir.

Uygulamaların ağ performansını ölçmek adına, ARINC 653 ve AFDX standardlarının tanımladığı bütün kuralları simulasyon ortamında sağlamak hem mantıklı hem de gerekli değildir. Bu yüzden ilk olarak bir sistem modeli geliştirilerek haberleşme performansını ölçmek için gerekli olan bileşenler belirlenmiştir. Haberleşme performans ölçümü yapmak için temelde sistemde bulunan *işlemlerin* standarda uygun olarak seçilip çalıştırılması gerekmektedir. Bu görevi yapmak adına *bölme*, *bölme yöneticisi*, *işlem* ve *işlem yöneticisi* bileşeleri tasarlanmalıdır. *Bölme yöneticisi* modülü, MTF'nin içeriğine göre gerekli olan *bölme* modüllerinin başlatılması ve durdurulması operasyonlarını gerçekleştirmelidir. *Bölme* modülü ise içinde alt modül olarak *işlem yöneticisi* ve *işlem* modüllerini barındırmalıdır. Ayrıca *bölme* modülü, kendisine gelen başlatma ve durdurma isteklerini *işlem yöneticisi* modülüne aktarmalıdır. *İşlem yöneticisi* modülüde kendisine gelen bu isteklere göre *işlemleri* durdurma veya koşturma operasyonlarını gerçekleştirir. Bu bileşenler, aviyonik görevlerin ARINC 653 standardına uygun olarak yönetilmesini sağlarlar. Ancak buna ek olarak *bölmeler* arasındaki haberleşme de sistemin haberleşme performansını ölçmek adına gereklidir. Bunun için standard tarafından tanımlanan *örnekleme* ve *kuyruklama* haberleşme modlarını gerçekleyen bileşenlerde geliştirilmelidir.

AFDX standardı için modellenmesi gereken iki adet cihaz vardır: ES ve Anahtar. ES, *işlemlerin* ağda iletişim kurmalarını sağlayan cihazdır. Bir *işlem*, mesaj göndermek istediği zaman, ilk olarak oluşturulan mesajı ES içinde bulunan gönderici haberleşme portlarına yazar. Mesaj yazımının ardından cihaz; BAG, veri paket boyutu uyumluluğu

ve fazlalık güzergah mekanizması gibi yöntemleri paket üzerinde uygular ve paketi ağa gönderir. Daha sonra alıcı tarafta mesajı alan ES, paketi bütünlük denetimi ve fazlalık denetimi gibi işlemlerden geçirerek uygun alıcı haberleşme portuna yazar. Son olarak alıcı *işlem* aktive edildiğinde mesajı bu porttan okur. Anahtar ise ağdaki ES'leri birbirine bağlamada kullanılır ve temelde 3 adet görevleri vardır: *süzgeçleme*, *ilkeleme* ve *anahtarlama*. *Süzgeçleme*, anahtara gelen paketlerin VL veri paketi boyutu kısıtının sağlanıp sağlanmadığını kontrol eder. Öte yandan BAG aralığı kuralının kontrol edilmesi ise *ilkeleme* olarak adlandırılır. *Anahtarlama* özelliği ise gelen paketlerin hangi çıkış portlarına yönlendirilmesi gerektiğini belirleyen kuraldır. Haberleşmenin AFDX standardına göre gerçekleştirilmesi için modellenmesi gereken cihazlar ve yöntemler bu şekilde özetlenebilir.

Sistem modeli geliştirilen cihazların ve yapıların simulasyon ortamında aktarılması gerekmektedir. OMNET++, diğer simulasyon ortamlarına kıyasla daha iyi ölçeklenebilme ve genişletilebilme özelliklerine sahiptir. Ayrıca, OMNET++ aktif bir topluluğa sahiptir ve açık kaynak kodlu bir yazılımdır. Bunlara ek olarak ortamda INET gibi OSI katmanının ayrıntılı gerçeklenmesini sağlayan bir çalışma mevcuttur. Bu sebeplerden ötürü sistem modellerinin simulasyon üzerinde gerçeklenmesi için OMNET++ ortamı seçilmiştir. ARINC 653 standardı için sistem modelinde belirlenen *bölme*, *bölme yöneticisi*, *işlem* ve *işlem yöneticisi* bileşenlerine ek olarak *port kanalı* ve *ağ gönderici sürücüsü* olmak üzere iki adet daha bileşen geliştirilmiştir. *Port kanalı*, alıcı *bileşenler* ile ES'nin alıcı haberleşme portunu birbirine bağlarken, *ağ gönderici sürücüsü* ise gönderici *işlemin* yolladığı veri paketlerinin ES'de bulunan doğru gönderici haberleşme portuna yazılmasından sorumludur.

AFDX standardı için, daha önceki bir çalışmada geliştirilen bir model temel alınmıştır [1]. Anahtar cihazı için çalışmadaki model aynı şekilde kullanılabilirken ES cihazı ise ek bileşen geliştirmelerine ihtiyaç duymaktadır. Çalışmadaki modelde ES sadece cihaz seviyesinde ölçüm yapmak adına tasarlandığı için gönderici ve alıcı *haberleşme portları* tasarlanmamıştır. Dolayısıyla alıcı haberleşme portlarına mesajın yazılmasından sorumlu olan *yönlendirici* bileşeni de tasarlanmamıştır. Önceki çalışmadaki ES simulasyon modeli üzerine bu iki bileşen daha eklenerek paket alma ve verme işlemlerini gerçekleştirebilecek simulasyon modeli oluşturulmuştur.

Oluşturulan simulasyon modellerinin ilk olarak paket alma ve verme işlemlerini gerçekleştirip gerçekleştiremediği test edilmiştir. Bunu test etmek adına ağ modelinde gönderici ve alıcı olarak isimledirilen iki adet IMA modülü bulunduran bir senaryo oluşturulmuştur. Gönderici modülü, veri paketi oluşturulması ve bu paketin alıcı modele gönderilmesinden sorumludur. Senaryoda iki modül arasında herhangi bir anahtar cihazı bulunmamaktadır. Bu yüzden gönderilen paket doğrudan alıcı modülünün haberleşme portlarına yazılır. Alıcı modüldeki *işlem* başlatıldığında haberleşme portundaki veriyi alır ve içindeki mesajı simulasyon konsoluna yazdırır. *İşlem* bileşenin aldığı paket verilerinin doğruluğu simulasyon konsulunda gözlemlenerek paket alma ve verme işlemlerinin doğru bir şekilde yapıldığı anlaşılmıştır.

Paket alma ve verme işlemlerine ek olarak bileşenlerin doğru zamanlarda çalıştırılıp çalıştırılmadığı da test edilmelidir. Bunun için başka bir çalışmada bulunan daha gelişmiş bir senaryo kullanılmıştır [2]. Bu senaryoda toplamda 4 adet IMA modülü

ve 5 adet uygulama mevcuttur. Sadece bir modülde 2 adet uygulama mevcut iken diğer modüllerde birer adet uygulama vardır. İlk olarak geliştirilen yazılım-tabanlı uygulama gecikmesi formülü ile senaryoda bulunan sanal hatların teorik gecikmesi hesaplanmışıtır. Daha sonra her bir VL'in uygulama gecikmesi simulasyon ortamında elde edilmşitir. Elde edilen bu iki verinin birbirleri ile uyumlu olduğu gözlenmiştir. Ayrıca senaryodaki 1 numaralı sanal hattındaki gönderici uygulama 2 adet veri paketini aralarında zaman farkı olmaksızın göndermektedir. Alıcı ve gönderici uygulamaların periyot değerleri de bu durum göz önünde bulundurulmadan tasarlandığı için VL'in güncel BAG değeri de paket kaybını engelleyememektedir. Bu durumu çözmek adına BAG değeri alıcı uygulamanın periyoduna yaklaştırıldığında paket kabının azalması beklenmektedir. Bu beklenti farklı BTA değerleri için doğrulanmıştır. Teorik ve simulasyon uygulama gecikme değerleri ve BAG-Paket kaybı ilişkisinin beklenildiği gibi olması bileşenlerin doğru zamanlarda çalıştırılabildiğini göstermektedir.

Sistem modeli oluşturulan tasarım, simulasyon ortamına aktarılmış ve aktarılan modelin doğruluğu farklı parametreler ve senaryolar kullanılarak test edilmiştir. Bu model ile daha bir çok senaryonun performansı farklı metrikler kullanılarak ölçülebilir. Tezde yapılan çalışmaya daha gelişmiş teknolojiler eklenebilir. Örneğin, cihaz seviyesinde AFDX standardı ile uyumlu cihazlar yerine Zaman Tetiklemeli Ethernet (Time-Triggered Ethernet : TTE) ve Zaman Duyarlı Ağ (Time-Sensitive Networking : TSN) gibi teknolojileri destekleyen cihazlar kullanılabilir. Ayrıca tasarlanan ARINC 653 standardı modeli tek işlemcide çalışacak şekilde modellenmiştir. Bu model uygulamaların paralel olarak çalışıtırılabileceği bir hale getirilip daha farklı ölçümler yapılabilir.

1. INTRODUCTION

Advancements in electronic devices, particularly their form factors becoming smaller, have made it possible to integrate various avionic sub-systems and components into aerial vehicles. The diversity of the avionic sub-systems has led to their utilization in different areas such as military, agriculture, and surveillance through electronic devices such as radar and cameras. However, the increasing number of these devices harms the flight duration of aerial vehicles and the power demands of the vehicles, which increase with the use of a diverse and increasing number of avionic systems. Therefore, the traditional federated architecture, which utilizes each system in a different hardware component, cannot meet the needs of current large-scale avionics system development due to its portability, security, and scalability [3].

A system architecture, namely IMA, is designed to solve these issues. Around three decades ago, the IMA concept first emerged in the United States through the F-22 Joint Integrated Avionics Working Group (JIAWG). Subsequently, it spread to commercial aircraft and business jets during the late 1990s [4]. The IMA architecture combines various avionics functions with different levels of criticality on a shared computing and networking platform. The aerospace industry introduced two primary standards, ARINC 653 and AFDX, to satisfy the growing requirements for safety-critical IMA systems. These standards were developed to ensure that the integration of multiple avionics functions is secure and reliable.

ARINC 653 is a software specification that outlines a set of services known as APEX (Application/Executive) services for implementing communication between an avionic computer's application and operating system (O/S). Applying APEX services can be quickly executed on different O/Ss without significant modification. For example, there may be an application code that implements Kalman filter for controlling helicopter motors in VxWorks-653 [5] Real-Time O/S (RTOS). Suppose the company of the helicopter motor's producer decided to switch their O/S to GzIS [6]

RTOS, and the filter code is written by using APEX services. In that case, engineers do not have to modify their application codes. Also, the ARINC 653 standard establishes rules where different software applications can run on a shared hardware platform while isolated temporally and spatially. This isolation ensures that a problem in one avionics function does not impact the others. Moreover, the partitioning method is utilized to overcome issues related to size, weight, and power consumption [7].

In the past, bus standards such as ARINC-429, ARINC 629, and MILSTD-1553 were designed to communicate between avionic systems in aerial vehicles. However, these standards need to be revised to meet the requirements of modern aerial vehicles in terms of duplexity, bandwidth, speed, latency, and isolation [8]. AFDX is a newer standard that offers a high-speed, isolated, and deterministic network for avionics applications in aircraft systems. AFDX is based on the Ethernet protocol and provides features such as quality of service (QoS) guarantees and redundancy management. By combining ARINC 653 and AFDX, developers can create reliable, efficient, robust, and safe avionics systems that meet the rigorous demands of aerospace applications.

In the IMA system design, it is difficult to determine the system architecture which gives optimum network performance using manual methods, especially for large aircraft, because the degree of freedom in choosing the system architecture designed by two standards, ARINC 653 and AFDX, is high [9]. Thus, it can be challenging to predict the overall system behavior without running the entire system physically, which can be very costly during development. Airbus, a leading aviation industry company, proposes a concept *Open IMA* that defines standards for developing IMA technology on the A380 program. The primary purpose of the *Open IMA* is applying open avionic and commercial communication standards (ARINC 600 norms) for the avionic modules to obtain interoperability between the product owners and third-party avionic suppliers [10]. Also, the *Open IMA* introduces the AFDX network topology used in the A380 program. Figure 1.1 shows the content of the network topology.

The network topology comprises two redundant networks (Network A&B), 9x2 AFDX switches, and 123 End Systems. Performing tests on a network of this magnitude using a physically implemented setup would be costly and time-consuming. Hence,

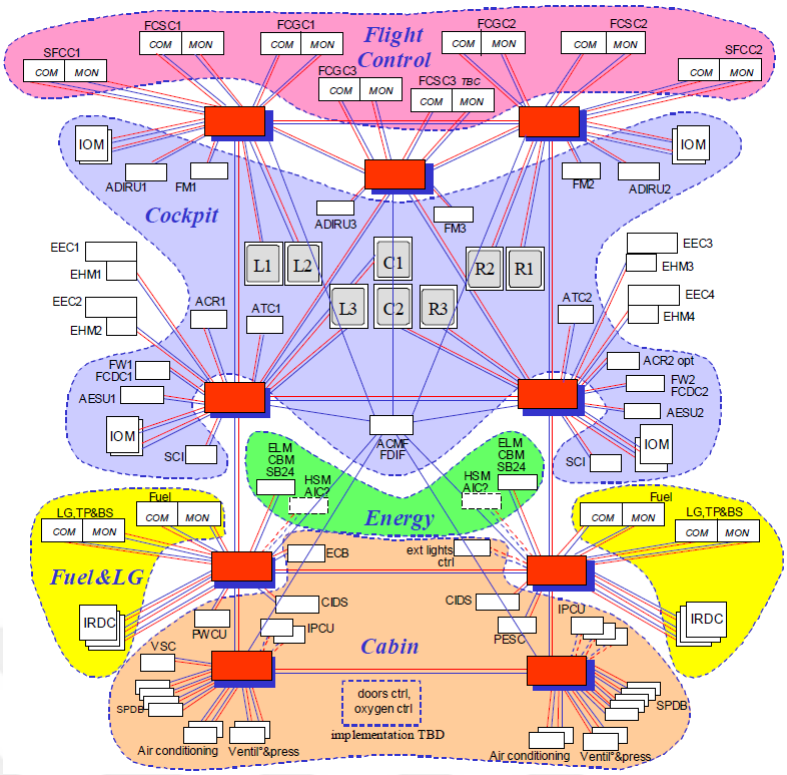


Figure 1.1 : The Network Topology on the A380 [10].

simulation is essential to ensure the IMA system operates reliably, safely and performs as intended in real-world scenarios. It allows designers to verify the design of the IMA system before deployment, allowing them to identify potential problems and make necessary modifications to meet the requirements of the avionic projects. Furthermore, simulation is a cost-effective way to test and validate the system without the need for physical prototypes, reducing development costs and speeding up the time-to-market for the system.

1.1 Purpose of Thesis

This thesis implements ARINC 653 concepts, such as partition, process, and inter-partition communication modes, in the OMNET++ simulator. To the best of our knowledge, this is the first paper to design these necessary components to build an IMA Module in OMNET++. The IMA simulation model allows for temporal control of applications and enables the designer to calculate network performance metrics such as application delay in the simulator. Since these ARINC 653 concepts are implemented as separate modules, developers can design their IMA system flexibly

and prototype it quickly by using the benefits of object-oriented programming (OOP). The functional and timing correctness of the designed IMA system modules are tested on an example scenario [2] that performs packet transfer using ARINC 653 concepts and AFDX-compatible devices.

Also, many different delay sub-steps in the network data path are taken into account in various studies [11]–[13]. However, no study considers the overheads of send and receive calls performed on the application side. This study will consider all software-based overheads for the application delay calculation for both the source application and the destination application, which will increase the accuracy of the application delay calculation.

1.2 Literature Review

Many simulation environments enable engineers to simulate IMA systems. However, some works develop their simulation tool for system evaluation, which requires unnecessary high-level and low-level coding efforts such as visualization and computing resource management of the newly developed simulator [14,15]. One of the tools used to model and validate the real-time system is UPPAAL. The tool is well-suited for systems that can be represented as processes with a finite control structure and utilize real-valued clocks, such as timed automata [16]. ARINC 653 basics, such as partition and process, are developed by [2,17] as timed and StopWatch automata, respectively. Moreover, developed modules are tested with sample scenarios for validation. Although the tool is appropriate for obtaining worst-case scenarios, it does not scale for large systems due to the combinatorial explosion problem. Besides, the development process is impractical compared to other tools. MATLAB/SIMULINK environment can also be used to design and evaluate the IMA system using a toolbox developed by Italian Aerospace Research Centre (CIRA) and MathWorks. When the SIMULINK block of the ARINC 653 package is created, the ARINC 653 standard and the specific code execution characteristics of the Wind River VxWorks 653 RTOS are considered [18]. Although the toolbox provides an excellent opportunity to perform hardware-in-the-loop (HIL) simulations of IMA scenarios, it restricts users to develop their system according

to Wind River VxWorks 653 RTOS. Besides, each environment package is not available and modifiable since MATLAB/SIMULINK is a commercial environment. IMA system design can also use network simulators like NS-2, NS-3, OPNet, and OMNET++. These tools use programming languages like C and C++ for development, allowing for code re-usability between simulation and real-world applications. Since OPNet is commercial, it has the same drawbacks as MATLAB/SIMULINK. Among open-source simulators, OMNET++ is chosen due to its scalability, extensibility, and integrability [12]. The simulator has a comprehensive framework named INET that implements each TCP/IP stack layer and some application layer protocols. Besides, it is regularly updated, well-documented, and has a large and active community [19].

Numerous studies have measured the end-to-end network delay between AFDX devices. In [1], an AFDX ES model is developed in OMNET++ simulator, and the model's performance is investigated with different traffic scenarios. Moreover, some papers propose a concept that extends the priority level of VL with two priority levels in AFDX standard to optimize network delay [20,21]. Also, network delay performance is enhanced with a genetic algorithm to determine VL priority level in [20]. Unlike network delay, relatively few papers examine delay at the application level. Both [12,13] design models that consider buffer latency which is the time the message waits to be consumed at the destination application's buffer. Then, the studies measure the end-to-end delay of VLs for sample network topology. Moreover, the results of [12] are compared with [22], which calculate the delay using the trajectory approach. However, the scheduling of applications is performed according to a simple messaging plan which does not conform to the [7] standard that manages tasks with fixed-priority preemptive scheduling. Thus, it is impossible to evaluate a complete IMA system using these two papers.

Badache et.al. investigates the temporal allocation of the IMA system and suggests a formula for calculating application delay to avoid the overwriting of messages by newer ones [11]. However, it does not consider consumed time in the transmission buffer. The studies [12,13] take times in the transmission buffer into account. However, they need to include transmission and reception operations overheads which should be considered for precise application delay calculation. Lu et.al. defines the overheads

between buffers and processors for transmission and reception operations, but it does not include them in its final evaluation [23]. This thesis considers the overheads of both transmission and reception when the proposed IMA models are evaluated.

1.3 Hypothesis

The essential components required for building an IMA system can be developed on the OMNET++ simulator. In this way, the development process of complex IMA systems is enhanced in terms of cost and reliability. Also, developing the IMA modules enables researchers to increase the precision in delay calculation by including the overheads of the send and receive user calls.

2. SYSTEM MODEL

2.1 ARINC 653

This section focuses on the fundamentals and system functionalities of the ARINC 653 standard and explains the system model that is used to emulate the standard concepts. The standard provides a set of methods (APEX services) for the interface between the application and the O/S of an avionics computer resource. The use of APEX services allows applications to run on different O/Ss without additional porting costs. Besides, the standard defines a software architecture that enables multiple applications with different levels of criticality to run securely on a single hardware platform. This is ensured with a technique called partitioning which ensures the spatial and temporal isolation of avionics applications. Partitioning involves splitting avionics applications into distinct software units, referred to as partitions. Each partition of an ARINC 653 system can be thought of as an individual avionic application that performs specific tasks for the entire system. However, they are logically separated within the same avionic hardware platform rather than being physically separate. Moreover, any malfunction or error in a partition will not affect others thanks to the partitioning method. As partitions are just only logical entities, they do not have an executable unit to carry out avionic tasks. Instead, they consist of a programming unit called a process, which comprises an executable program, data and stack areas, program counter, stack pointer, and other attributes such as priority and deadline [7]. The ARINC 653 standard provides detailed descriptions of all the system functionalities, including partition management, process management, memory management, and time management. While all of these requirements must be met to develop a fully compatible ARINC 653 O/S, this thesis focuses on only a few of them, specifically partition management, process management, and inter-partition communication.

Partition scheduling is an essential aspect of partition management in which the partition itself acts as the scheduling unit, and no priority is assigned to it, as stated in [7]. Partitions are restricted to accessing the computing resource of the module only during a predetermined time known as the partition time window, which is positioned on a fixed-size timeline called the MTF. The partition is activated only when its partition time window is reached. Then, the process scheduler selects the appropriate process of the activated partition for execution. The core module executes the MTF repeatedly and continuously until it is powered off. Figure 2.1 shows an example of a MTF with three partitions.

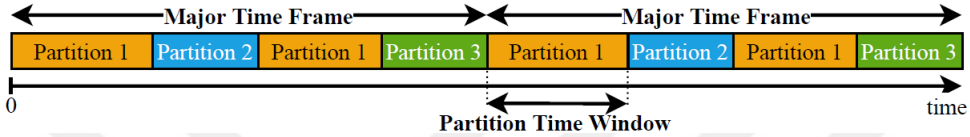


Figure 2.1 : Sample Major Frame.

As stated at the beginning of this section, avionic computer tasks such as I/O operation and data filtering are performed by processes. Each process is created and initialized at the beginning of the owner partition. Moreover, the required utilization of resources should be defined at the system build.

Each process has fixed attributes; *name*, *entry point*, which defines starting address, *stack size*, *base priority*, which defines initial priority, *period*, *time capacity* and *deadline* [7]. The *time capacity* defines the elapsed time that process can access the computing resources of a module, and it is used to calculate *deadline* time of the process, which can be calculated by adding *time capacity* to the process's release point time. Moreover, the *time capacity* does not only define the execution time of the process; the time that process is waiting for a resource also consumes the capacity of the process. For example, a process may receive a message from a socket and perform a data-filtering algorithm using the received message. In the receive operation of the socket, if no message is found in the socket buffer, the process can be blocked at a semaphore. The elapsed time while waiting for the network packet at semaphore also consumes the *time capacity* of the process. The *time capacity* of each process is set to the infinite value in our system model, so the process' *deadline* is not checked by the developed partition manager module.

There are two types of processes periodic (synchronous) and aperiodic (asynchronous) in terms of periodicity. Aperiodic processes have only an initial *release point* that is determined by the start service. After the start service is invoked for an aperiodic process, it is eligible to be selected by the process manager with partition start and *deadline* of the process is calculated by summation of the partition's start time and *time capacity*. After an aperiodic task finishes its task and calls the stop service, it will not be activated again during MTF. On the other hand, periodic processes are executed in regular time intervals which are defined with the *period* attribute. The *period* defines the regular time interval of the process's execution. An explicit and unique value is assigned to *period* for the aperiodic process. Thus, the process manager does not consider the periodicity of the aperiodic processes. The periodic process can also start to execute with the start of partition as aperiodic ones. However, newer *release points* have been created for them during core module execution. Moreover, the *PERIODIC_WAIT* service should be called by a periodic process when it finishes its work for the current activation. After this call, a new release point is assigned to the process. Hence a new *deadline* is also calculated. In this thesis, the processes are designed as aperiodic.

In contrast to partition scheduling, process scheduling involves giving a priority level to each scheduling unit (process). ARINC 653 follows a fixed-priority preemptive scheduling approach, where the scheduler chooses the process with the highest priority for execution, and any lower-priority process currently running is preempted. The priority level assigned to each process is determined by considering its criticality and the specific deadlines it needs to meet.

In ARINC 653, inter-partition communication is carried out using a messaging protocol, where communication ports and channels are utilized to send messages from a source partition to one or more destination partitions. The functioning of ports is described in [7], but the implementation may differ based on the design of the O/S. In this thesis, a shared storage area is employed for both source and destination partitions, and the memory content of the port is updated according to the algorithm of [7] for the system model. The ports have two modes of operation, namely sampling and queuing. For sampling mode, [7] allows a port buffer to hold only one message at

a time, which means that the existing message in the buffer remains there until a new one overwrites it. Additionally, the freshness of the message can be determined using a term called the refresh rate, which controls the duration between the arrival and reading time of the message. A refresh rate is assigned to each sampling port in order to verify the age of the messages. In ARINC 653, the partition port has also another mode called queuing that follows the first-in/first-out (FIFO) principle for the storage area. This mode uses a buffer where the source partition writes messages to the FIFO and the destination partition retrieves the oldest message from the buffer. To prevent any unintentional message loss, the application software manages queuing port overflow, as specified in [7]. Our work utilizes these three sub-functionalities of ARINC 653, and the developed system model is shown in Figure 2.2.

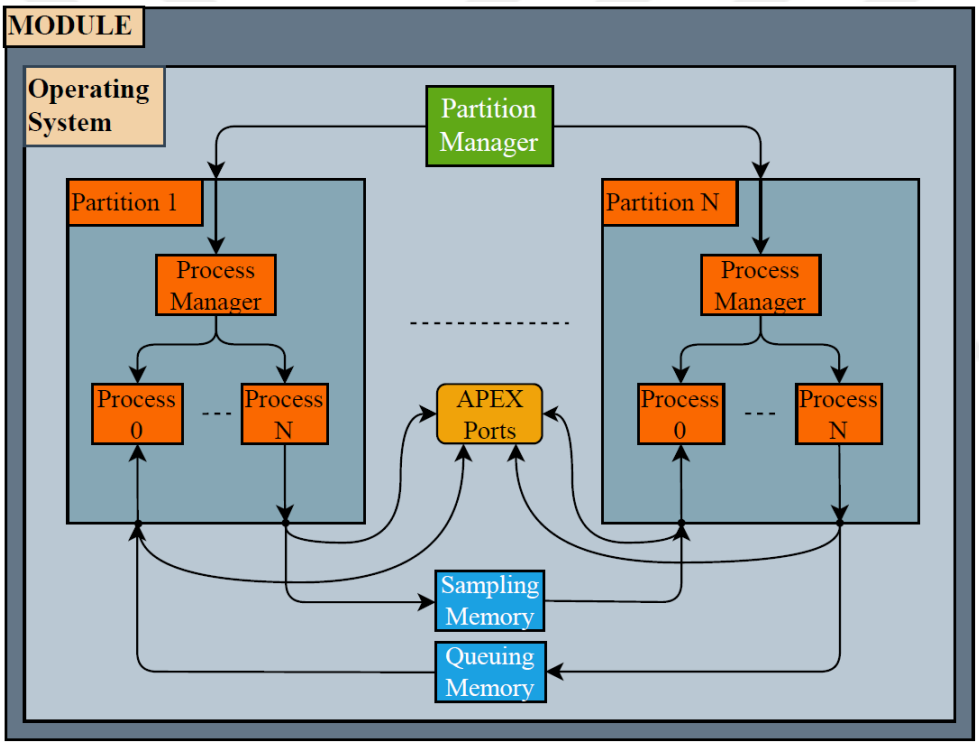


Figure 2.2 : System Model of ARINC 653 Module.

The main responsibility of the *partition manager* component is the initiation and termination operations of the *partition* components based on the MTF. The *partition* component should have two sub-components namely *process manager* and *process*. Also, the *partition* component transmits the received requests from the *partition manager* to the *process manager*. When the *process manager* receives a request, it identifies the active *process* component with the highest priority at first. Then, *process*

manager triggers that *process* to execute its task. The management of *processes* can be performed by these four components according to the ARINC 653 standard. Besides, inter-partition communication is implemented by APEX ports and buffer of *sampling* and *queuing* memory components. When a *process* wants to write a message to a memory component, it should first find the source APEX port of the memory component. Then, the *write* service of the memory components is invoked by the writer *process*. One of the differences between the reader and the writer *processes* is that the reader *process* identifies the destination APEX port of the memory components rather than the source port. Also, the *read* service of the memory component is invoked by the *process* instead of the *write* service.

2.2 AFDX

AFDX allocates network device bandwidth to the applications by using VLs, which is similar to how ARINC 653 allocates computing resources to partitions. The bandwidth allocated to each VL is determined by two parameters: BAG and S_{max}. BAG specifies the minimum time interval between consecutive VL frames, while S_{max} indicates the maximum frame size that can be transferred over the VL. AFDX communication requires two primary devices, ES and Switch. The ES is used to send and receive packets within an IMA module by partition's processes, while the switch is responsible for determining packet routes in the network and applying filtering and policy functions based on the [24]. The components of the ES device system model for transmission and reception are depicted using blue and orange blocks, respectively, in Figure 2.3.

In AFDX, source applications of partitions assign a sequence number between 0 and 255 to messages that will be sent, which is used to verify message integrity. The sequence number 0 is reserved for resetting communication in the VL, while other numbers are used for regular communication. The first message in a VL is assigned sequence number 1, and subsequent messages' sequence numbers are incremented by one. When the sequence number reaches the value of 255, it backs to 1 in the next message. After the message is set up by the source application, it is written to AFDX communication ports ($AFDX_{port}$) by using the write service. These ports have two modes, sampling and queuing, and the algorithms used for these modes

are similar to the ones used in ARINC 653. Once the message is received by the *regulator* component from $AFDX_{port}$, it is checked that the BAG time condition is met. If the condition is satisfied, the message can be received by the *VL Scheduler* component. The*VL Scheduler* chooses messages from various VL queues, using a scheduling algorithm such as round-robin. After a message is selected by the *VL Scheduler*, it is duplicated and transmitted over two different networks (Network A and Network B) by the *redundancy management* module to enhance communication reliability. After the destination ES receives the messages, the *integrity checker* component ensures that the frames are in the correct order by using sequence numbers. The first message that arrives at the *redundancy management* component is sent to the *DEMUX* component, while later incoming messages with the same sequence number are discarded. The message is then routed by *DEMUX* to the appropriate $AFDX_{port}$ based on its quintuplet, which includes the UDP source port, source IP address, destination MAC address, destination IP address, and UDP destination port. Finally, the destination application retrieves the messages from the $AFDX_{port}$ with the read services.

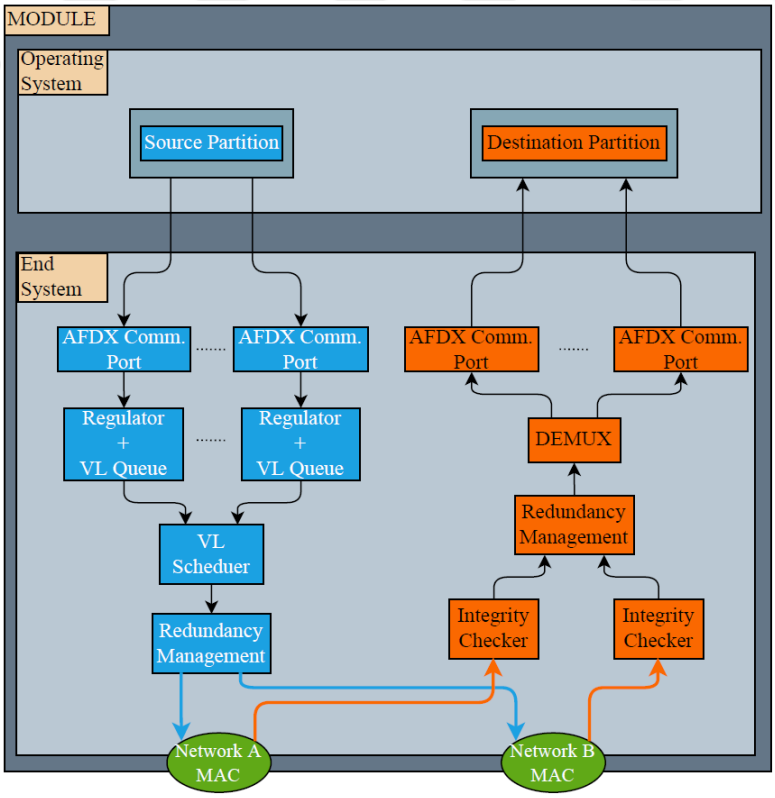


Figure 2.3 : System Model of ES Device.

The AFDX Switch performs three main tasks: *filtering*, *policing*, and *switching*. *Filtering* is used to drop packets that are not within the specified size range, as defined by S_{min} and S_{max} and *policing* is used to discard packets that do not satisfy the BAG rule. Finally, *switching* involves routing incoming packets to predetermined network ports, as specified by the network planner [1]. Most of the components except *DEMUX* and $AFDX_{port}$ are implemented in [1]. We utilize these developed and tested AFDX components as-is in our work, while we implemented the remaining two modules according to the specification provided in [24].

2.3 Application Delay

The time elapsed from when a packet leaves the APEX port of the source partition until it is received by the destination partition is known as the application delay (D_{app}) and includes not only the network delay ($D_{network}$), but also software delay such as buffer delay and read service call overheads [11]. The green line in Figure 2.4 represents the packet flow path of D_{app}, while the red line represents the path of $D_{network}$.

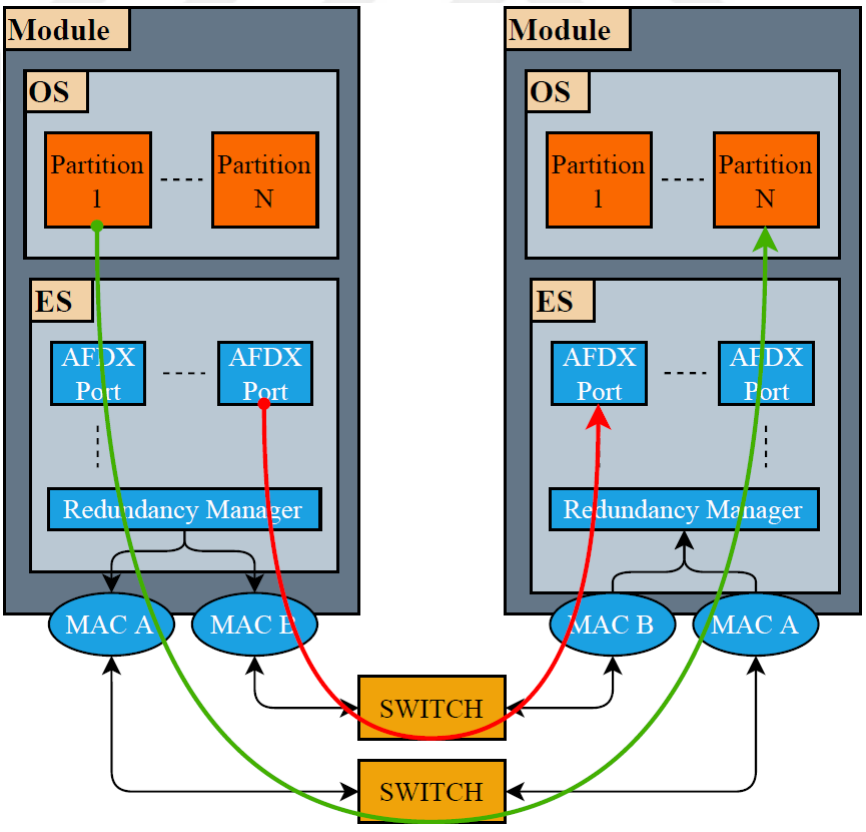


Figure 2.4 : Application vs Network Delay.

Although both application and network delay paths are represented with a complete line, the paths can be divided into sub-steps as in [23]. The first step is named D_{txOp}. It is comprised of obtaining and processing raw data operation and it does not contain any networking task. Reading IMU and gyroscope sensors data and applying complementary filter method to the data can be given as an example of this step. In the second step (D_{txApp}), overheads of writing the frames to the device buffer are included in D_{app} calculation. This step covers the essential operations such as system call, context switch, and data copy of an O/S. For example, when the filter application calls a write request to a device sampling port, O/S switches from user mode to kernel mode with a system call. Then, the application attempts to take the semaphore that protects $\text{AFDX}_{\text{port}}$'s buffer integrity. However, if it is already taken by another thread, the sender application will be suspended and a context switch will occur. When the application is awakened, it takes the semaphore and access to $\text{AFDX}_{\text{port}}$'s buffer. Then, it copies the filtering data to the port buffer of $\text{AFDX}_{\text{port}}$. Afterward, transmitter ES retrieves the frame from the $\text{AFDX}_{\text{port}}$ and sends it to the physical link. This step (D_{txDev}) covers overheads at the device level such as VL scheduling and frame duplication. Then, frames are transferred through the physical link which also introduces a delay ($D_{phyLink}$). After that, receiver ES takes the frames from the physical link and applies the integrity checking, redundancy management, and demux operations as a fifth step (D_{rxDev}). Then, the frames wait in $\text{AFDX}_{\text{port}}$ buffer to be consumed by the destination application. The elapsed time in the buffer is named buffer delay and represents the sixth step (D_{rxBuff}). Then, the destination application is activated by the process manager and the frame is read from $\text{AFDX}_{\text{port}}$ in the seventh step (D_{rxApp}). Same overheads of D_{txApp} such as system call and data copy are also occurred in D_{rxApp}. Finally, the received frame is processed and transmitted to the destination component in the last step (D_{rxOp}). For example, received data from the complementary filter may be processed by a control algorithm to convert it to a motor control command in the destination application. These sub-steps are visualized in Figure 2.5.

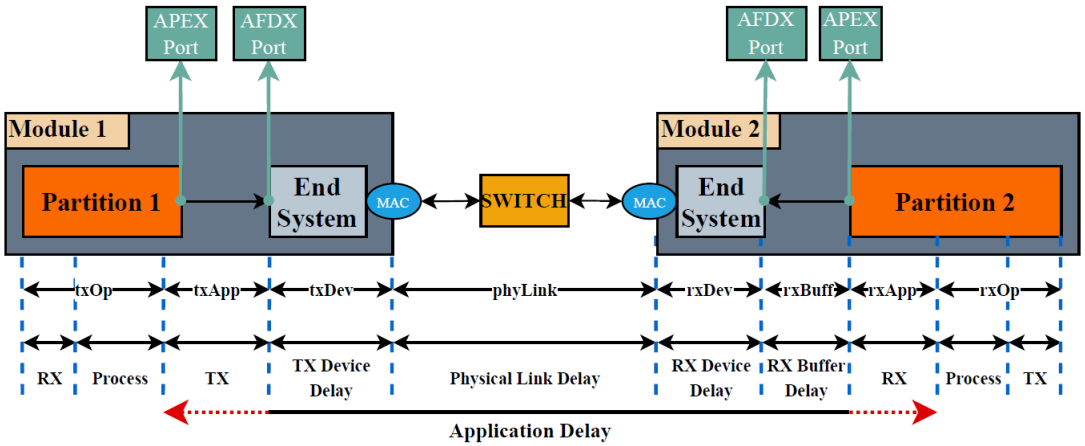


Figure 2.5 : Latency Sub Steps.

Badache et.al. formulates D_{app} as summation of physical link delay ($D_{phyLink}$) and buffer delay (D_{rxBuff}) as follows [11]:

$$D_{app} = L_{i,j} + J_{i,j}, \tag{2.1}$$

where $L_{i,j}$ represents $D_{phyLink}$ between the i^{th} and j^{th} partition, and $J_{i,j}$ shows the D_{rxBuff}.

The maximum value (${D_{app}}^{max}$) of D_{app} occurs when a packet experiences a maximum delay (${D_{phyLink}}^{max}$) in the physical link and arrive at the destination partition exactly when it is terminating, as stated in equation 2.2. This coincidence causes the buffer latency ($J_{i,j}$) to be equal to the destination partition period (T_j). Thus, ${D_{app}}^{max}$ can be expressed as the sum of T_j and ${D_{phyLink}}^{max}$ as shown in equation 2.2.

$${D_{app}}^{max} = {D_{\text{phyLink}}}^{\text{max}} + T_j \tag{2.2}$$

Additional steps need to be included to equation 2.2 since the filtered sensor data is generated prior to $D_{phyLink}$ in the D_{txOp} process. As a result, the calculation for D_{app} should begin after D_{txOp} step. Furthermore, ${D_{app}}^{max}$ cannot end with D_{rxBuff} as the data can be utilized by the destination application only at the end of D_{rxApp}. While D_{txDev} and D_{rxDev} are deterministic because they are performed by hardware, their value can be added to 2.2 as they are. However, the execution times of D_{txApp} and D_{rxApp} which are comprised of mostly software operations are not deterministic because of memory and device operations involved. When calculating ${D_{app}}^{max}$, the maximum value of

these two operations (${D_{txApp}}^{max}$ and ${D_{rxApp}}^{max}$) should be added to equation 2.2. After taking into account these additional delays, the new expression for ${D_{app}}^{max}$ can be written as:

$$ {D_{app}}^{max} = {D_{txApp}}^{max} + D_{txDev} + {D_{phyLink}}^{max} + D_{rxDev} + T_j + {D_{rxApp}}^{max}, \tag{2.3} $$

Although the complete application delay calculation can be computed with equation 2.3, the thesis focuses on the software-based application delay so the deterministic delay steps (D_{txDev} and D_{rxDev}) are included to the formula that is used in the thesis. Finally, the formula is updated as follows:

$$ {D_{swApp}}^{max} = {D_{txApp}}^{max} + {D_{phyLink}}^{max} + T_j + {D_{rxApp}}^{max}, \tag{2.4} $$

3. SIMULATION MODEL

3.1 OMNET++ Simulator

OMNET++ is a discrete event simulator (DES), so knowing what DES is and how it works is vital for understanding the proposed simulation model. DES models a system's behavior as a series of discrete events that occur over time [25]. The situation of the system is represented by states and the change in the system's situation is indicated by the transition between states. The state transition is named as event whose execution time is zero for DES. In DES, the time at which events happen is commonly referred to as *event timestamp*, but the term *arrival time* is used instead of it in OMNeT++ [26]. This is because the word *timestamp* is already reserved for a user-defined attribute in the event class library. On the other hand, the time of the model can be named as *simulation time* or *virtual time*. The *simulation time* represents the timeline that *event timestamp* is observed, and it is totally different than *real time*. The *simulation time* elapses only between events, and there will be no change in the system status between two consecutive events.

Computer networks and devices can be modeled with DES. For example, an Ethernet device can be represented with two states: *idle* and *transfer*. At the beginning of the simulation, the device is in the *idle* state, which remains in this state when there is no packet transfer. However, when a packet is sent or received by the device, the device switches to the *transfer*. An Ethernet device may receive a packet from the network stack 2 nanoseconds after the simulation starts, and the transmission lasts for 5 nanoseconds. Then, the transmission will end, and the device will switch back to the *idle*. The graphical representation of this scenario is shown in Fig. 3.1.

An OMNET++ network module is comprised of two basic terms *modules* and *connections*. Modules have two main types, which are named as simple and compound. The simple module is the core unit of OMNET++'s node, and its behavior is programmed with C++ programming language. On the other hand, compound modules

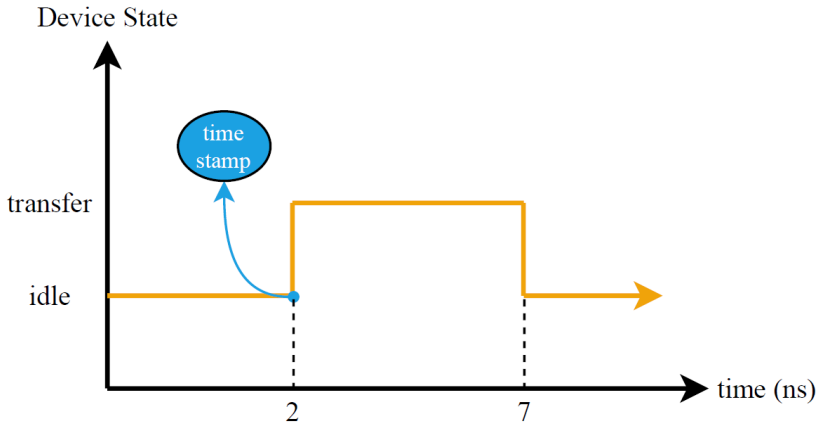


Figure 3.1 : DES Scenario of Ethernet Device.

do not have a C++ function that defines the module's behavior. They consist of a combination of simple and other compound modules, which are connected to each other via *connections* and *gates*. An example OMNET++ network module, which contains a compound module, is illustrated in Figure 3.2.

When OMNET++ users program their simple modules, they use the *cSimpleModule* class as a base class. In *cSimpleModule* class, there four main virtual methods namely, *initialize*, *finish*, *handleMessage*, and *activity*. While the default version of *initialize* and *finish* does not do anything, *handleMessage* and *activity* throw an error. At the beginning of the simulation, the constructor of the classes is executed first. After that, the network simulator builds the network model and calls each module's *initialize* method.

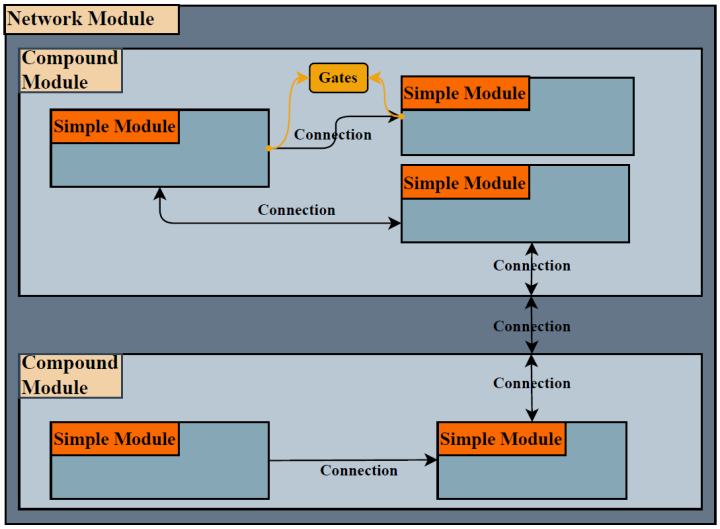


Figure 3.2 : Sample OMNET++ Network Model.

The purpose of the *initialize* method is that simulation-related code cannot typically be placed in the constructor of a simple module because the simulation model is still in the process of being set up, and many necessary objects may not yet be available [26] during constructor execution. For example, users may want to write a code that allocates memory for each node that is connected to a specific gate. In this situation, the connection number can be computed using the *gateSize* function in *initialize* since the network simulator already set up the network module. The *finish* method is called after a successful simulation completion, and it is used to record network statistics such as delay, latency, and packet drop. The method is not called after the simulation ends with an error, and the destructor of the objects is called after *finish* method.

A simple module's behavior is determined by the *handleMessage* or *activity* method. Both methods cannot be used simultaneously for a module, so only one can be implemented for each simple module. The *handleMessage(cMessage* msg)* method is invoked when a message arrives to the module, and the input parameter (*msg*) corresponds to the address of the received message. The act of invoking a message can be considered as an event of DES in OMNET++ because it results in a change in the network model's situation. Moreover, *simulation time* does not elapse in the *handleMessage* method. While a simple module is programmed with *handleMessage* method, there are three main methods, namely, *send*, *scheduleAt*, and *cancelEvent*. While the *send* method is used to send messages through a specified gate, the *scheduleAt* method sends messages to the module itself. This method is beneficial in creating timers within the simulation environment. The *cancelEvent* method is used to delete the message that is sent with *scheduleAt*.

The *activity* method is another way to specify the behavior of a simple module that works like a co-routine. Since the workflow of this method resembles threads and processes, it is more appropriate to implement software-based ARINC 653 concepts with the *activity* method rather than the *handleMessage* method. This notion is supported by [26], which states that "By using the activity() function, it's possible to write a simple module in a way that works as operating system's process or thread". Unlike *handleMessage*, a separate stack is needed for each simple module when programmed with *activity*. Thus, the use of this method increases the memory

consumption of the network model as the number of modules increases. However, only the operations that are implemented as software in the real world should be programmed with the *activity*, while implementing hardware-based operations such as integrity checking and redundancy management with *handleMessage* is more appropriate and sensible. Thus, the simulated network model does not harm the OMNET++ scalability, and some methods can be programmed with process-style descriptions more easily and properly. The *activity* method has six main methods namely, *send*, *scheduleAt*, *cancelEvent*, *receive*, *wait*, and *end*. While the functioning of *send*, *scheduleAt*, and *cancelEvent* methods are the same as in *handleMessage*, the other three methods cannot be used by a simple module that is programmed with *handleMessage*.

The *receive* method returns the address of the messages that come to the simple module. Since the *simulation time* does not elapse in the method as *handleMessage*, the inside of the *activity* can resemble the DES event. However, the *simulation time* elapses while waiting for a message with the *receive* method. There are two signatures for *receive*: "cMessage *receive()" and "cMessage *receive(simtime_t timeout)". While the method whose signature does not have an input parameter waits until a message comes to the module and never returns until a message comes to the simple module. On the other hand, the method with *simtime_t* parameter waits for an absolute *simulation time* that is defined by the input parameter for packet reception. If a message does not come to the module during this time, the *receive* method will return a NULL address. The *wait* method is used to block execution of the *activity* method and allows *simulation time* elapses. Moreover, the *end* method can be used to terminate the execution of the simple module's caller.

When one module sends a message to another, OMNET++ creates an event and adds it to the Future Event Set (FES) list with its corresponding timestamp. The simulator executes events in the FES according to their timestamps. If the timestamps of two events are the same, the one with higher priority assigned by the user will be executed first. If their priorities are equal to each other, the event added to the FES earlier will be processed first.

3.2 OMNET++ IMA Module

The proposed OMNET++ IMA simulation model includes two main modules: the *O/S* and the *ES* module. While the ARINC 653 concepts are implemented by *O/S*, the *ES* offers AFDX end-system capabilities. The *O/S* module was not designed as a compound module to show its relationship with *ES*.

3.2.1 Operating system module

The primary goals of the *O/S* module are to schedule partitions and processes based on ARINC 653 standard. The thesis provides main OMNET++ modules such as *partitionManager*, *partition*, *processManager*, and *process* to accomplish these tasks. Firstly, a simple send-and-receive operation between IMA modules is performed in a sample scenario created by ourselves to check whether packet transfer can be performed. In the scenario, there are two main IMA modules which are named *isrRadar* and *displayUnit*. The *isrRadar* module collects raw data from different sensors and processes them to produce the enemy aircraft positions. Then, the produced data is sent to the *displayUnit* module, which demonstrates the positions of the enemy aircraft on a screen for pilots. The *isrRadar* module's operations which are collecting sensor data, processing them, and sending processed data, are performed by three distinct partitions *sensor*, *algorithm*, and *transmitter* respectively. The *isrRadar* is developed as a compound OMNET++ module, and its internal modules and their connections are shown in Figure 3.3. There are five types of OMNET++ module; *partitionManager*, application partitions (*sensor*, *algorithm*, and *transmitter*), *networkTxDriver*, *portChannel*, and *ES* in the *isrRadar*.

3.2.1.1 Partition manager module of OMNET++

The *partitionManager* module is programmed as a simple OMNET++ module and is responsible for activating and deactivating application partitions. Activation and deactivation operations are performed by sending *Start Partition* and *Stop Partition* messages to application partitions, respectively. The necessary module data structure is constructed in the *initialize* method. Two main tasks, identifying the gate and obtaining the timing feature of application partitions, are performed in the *initialize* method.

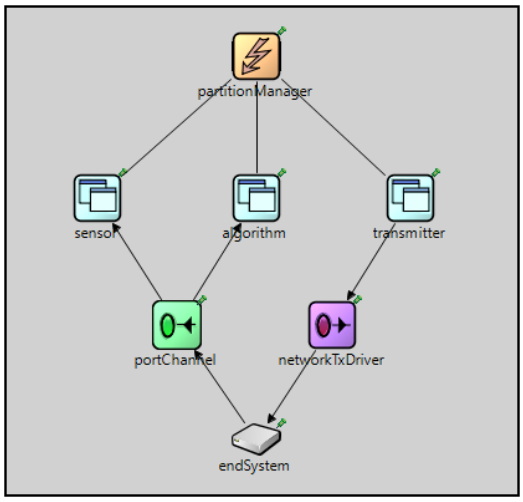


Figure 3.3 : Sample OMNET++ IMA Module.

Each application partition is connected to the *partitonManager* via a gate, namely *Scheduler_Out*, and no other modules are connected to the *partitionManager* module through the *Scheduler_Out*. When the *initialize* method of the *partitionManager* is invoked with simulation start, the number of the application partition is computed by getting the size of *Scheduler_Out* with the *gateSize* method at first. Then, memory addresses of the connected application partitions are found by using *gate*, *getNextGate*, and *getOwnerModule* methods. Then, *ID* of the application partitions are found by using the module address and *par* method. Afterward, gate index and application partition ID matchings are preserved in an array. Thus, the gate indexes to which the partitions are connected are identified.

In addition to gate identification, the offset and duration of each application partition should be found to perform activation operations with the correct timing. The offset and duration of each partition are preserved in the XML file, which is named *radarMajorFrame.xml*. The XML file separates partition information using tags, namely, the *partition*, which contains ID, offset, duration, and name of application partitions. The content of the *radarMajorFrame.xml* can be seen in Figure 3.4. The XML file is parsed by using the methods of the *cXMLElement* class to construct a structure that contains the offset and duration of the application partitions. Once all the necessary data structures are set up, a self-message is sent to the module itself for the first application partition's offset. This is achieved using the *schedAt* method at the end of the *initialize* method.

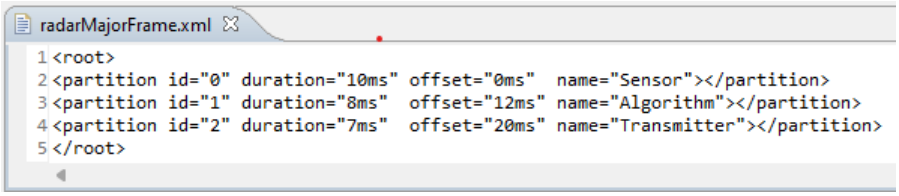


Figure 3.4 : Major Frame of the Sample OMNET++ IMA Module.

To invoke *partitionManager* at partition switch times regularly, a self-message is sent to the module itself by using the *schedAt* method in both *initialize* and *activity* method. However, gate indexes of the current active partition and the partition that will be activated should be already known to send the *Start Partition* and *Stop Partition* messages. These indexes are calculated before sending a message with *schedAt*, and the computed gate indexes of the current active partition and newly activated partition are preserved in *prevPartitionIndex* and *nextPartitionIndex* attributes, respectively. When the module receives the self-message, it will send *Partition Stop* message to the application partition, which is connected to *partitionManager* with $prevPartitionIndex^{th}$ index of the *Scheduler_Out* gate if the index value is a non-negative number. After that, *Partition Start* message is sent to the application partition, which is connected to *partitionManager* with $nextPartitionIndex^{th}$ index of the *Scheduler_Out* gate if the index value is a non-negative number. For example, the *nextPartitionIndex* is set to 0, while the *prevPartitionIndex* is set to -1 since there are no active partitions at the end of *initialize* method. After the index setting, a self-message is sent to the time after the offset of the first application partition. When the self-message is received in *activity* method, the method first checks the value of *prevPartitionIndex*. Since it is a negative value, the *Stop Partition* message is not sent to any partition. However, *Start Partition* is sent to the first partition of the MTF since the *nextPartitionIndex* is non-negative. The pseudo-code of the *initialize* and *activity* method of the *partitionManager* can be examined in the Algorithm A.2. Besides, the activation messages are sent with *sendDelayed* methods whose second parameter defines the delay time of the invocation message. If a delay time is defined, the cost of the partition switching overhead can be simulated on the model to obtain a more realistic result.

3.2.1.2 Partition and process manager module of OMNET++

The *partition* module is made up of two simple modules: *processManager* and *process*. Each *partition* must contain a *processManager* which schedules the *processes* within the *partition* based on the fixed-priority method. When a *Partition Stop* message is sent by the *partitionManger*, the message reaches the *processManager* module at first. Then, *processManager* deactivates all *processes* and sends a *Process Stop* message to each of the *process*. On the other hand, if it receives a *Partition Start* message, it will activate all of them and send a *Process Start* message to the *process* with the highest priority. Additionally, the active *process* triggers the *processManager* to select a new *process* after finishing its execution by using *gate*, *getPreviousGate*, and *getOwnerModule* methods. Similar to the overhead involved in partition switching, switching between processes also comes with a cost in an embedded system. This cost can also be applied to the simulation model with the second parameter of the *sendDelayed*. The content of the *sensor* application partition is illustrated in Figure 3.5.

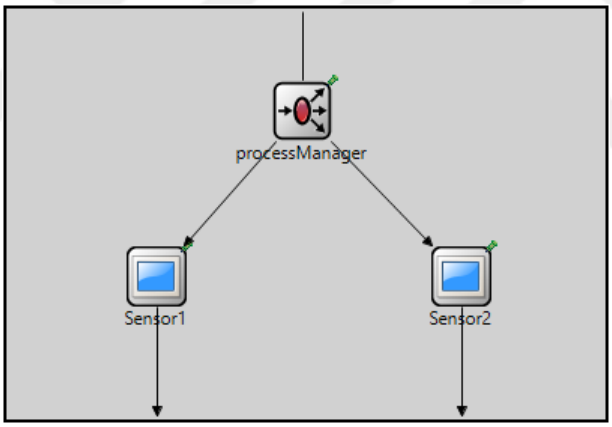


Figure 3.5 : Sample OMNET++ Partition Module.

3.2.1.3 Process module of OMNET++

The main tasks of the *process* module are transmitting and receiving AFDX messages to and from the ES ports. The *process* can perform its tasks only after it receives a *Process Start* message. In the transmission, the module creates an AFDX message and sends it to the *NetworkTxDriver* module. When *NetworkTxDriver* receives a message, it scans the transmission ports of the *ES* (*afdxPortTX*) to find a port with a matching quintuplet at first. Then, if a port is found, the message will be written

to that port buffer. Once the packet is placed in an *afdxPortTX*, the remaining packet transmission operations are performed by *ES* module. However, before sending the message to *NetworkTxDriver*, the *waitAndEnqueue* method is called to simulate the overhead of the user send service. The *waitAndEnqueue* waits for the time determined by the first parameter. If a message is received during the wait operation, it will be enqueued to a queue whose address is given as a second parameter. If *process* module receives a *Process Stop* message during the waiting, the action of the *process* will not be performed.

In the case of receiving, the *process* module first finds the address of the *portChannel* module, which connects an ES receiver port (*afdxPortRX*) to the one or more *process* module(s). Each *portChannel* module has a VL ID parameter, so it controls whether it is connected to the *afdxPortRX* module whose VL ID is the same or not. If it is not, a run-time error will be thrown at the *initialize* method. Although *portChannel* is not compulsory for the IMA model, it provides a more understandable view by reducing connection complexity. After finding the address of *portChannel*, the address of the connected *afdxPortRX* is also found. Then, the *getPacket* method of the port is called using the module address. This method returns the message stored in the *afdxPortRX* buffer to the caller *process*. As in transmission, the *waitAndEnqueue* method is called before all of these operations to simulate the overhead of the user receive service. The activity methods of the simple transmitter and receiver process are represented in Algorithm A.3.

3.2.2 End system module

The developed ES module expands the modules given in [1] with newly developed *demux* and *afdxPort* OMNET++ modules. The *afdxPort* modules should function in both queuing and sampling modes according to ARINC 653 standard, but only the sampling mode is implemented in the thesis. Figure 3.6 shows the simulation model that is used in the thesis.

The *afdxPort* consists of two types: transmitter (*afdxPortTX*) and receiver (*afdxPortRX*). In transmission, the role of *afdxPortTX* is just establishing a connection between the *process* and the *VL queue*. Once it receives a message from a *process*,

it forwards it directly to the connected VL queue. In the reception, AFDX packets coming from the *redundancyChecker* module are directed to the *afdxPortRX* through the *demux* module. The operation of the *demux* module resembles the DEMUX concept, which passes the input signal to the output according to selection bits in digital design. When an AFDX message arrives at the *demux*, it checks the ES table to determine the destination *afdxPort* based on the message's destination MAC and IP addresses and UDP port. Once the appropriate *afdxPortRX* is identified, the *demux* writes the message to that *afdxPortRX* buffer. The *process* module can access and read these messages when it becomes activated, as described in Chapter 3.2.1.3.

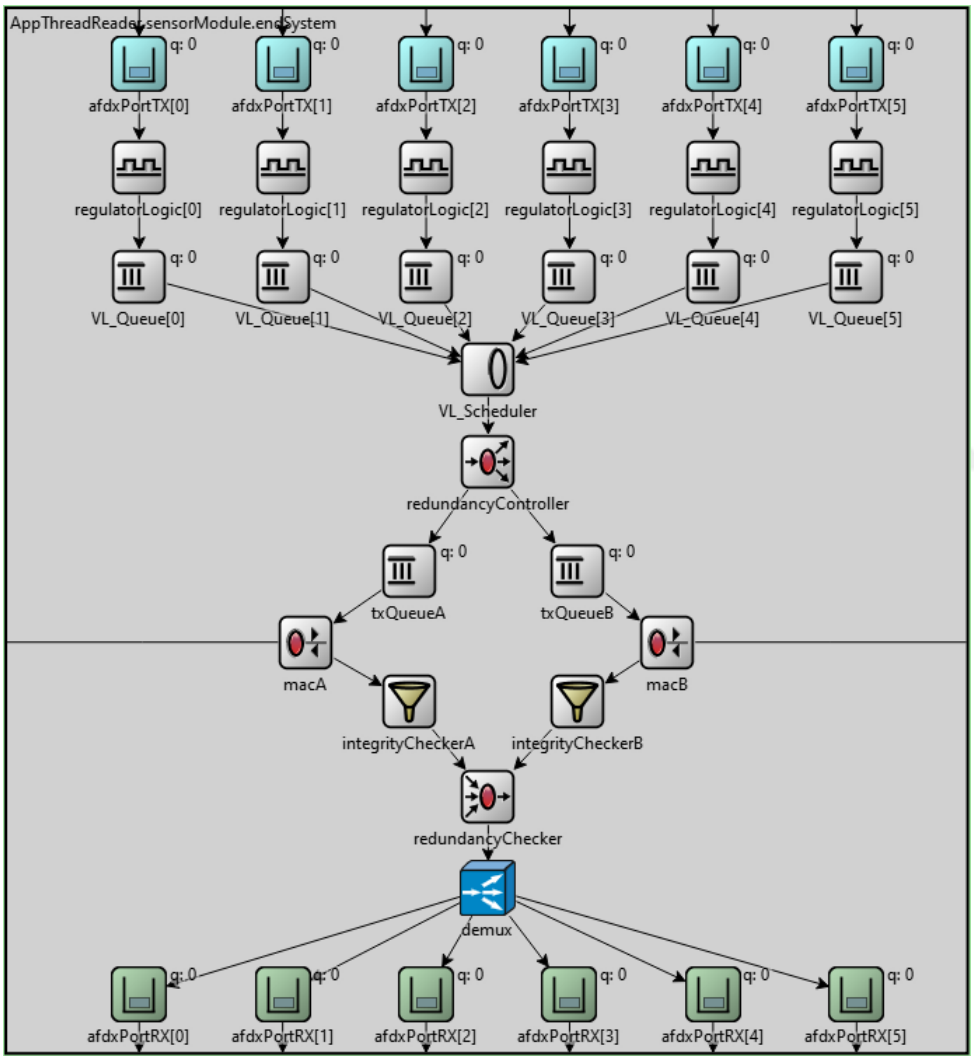


Figure 3.6 : Sample OMNET++ Process Module.

Unlike the *afdxPortTX* module, the *afdxPortRX* module is accessed by two modules, namely *demux* and *process*. Therefore, it is crucial to manage concurrent access to

ensure data integrity. Since the execution of the *activity* method can be seen as a DES event, accessing the *afdxPortRX* buffer within this method automatically maintains the integrity of the data. However, ensuring that the port access must occur within a single event is essential. For instance, if a write operation is designed to be executed in multiple events, reading from the buffer between these write events may lead to the reception of an incorrect message.

4. RESULTS

The message transfer capability of the proposed IMA module is verified through a basic send-and-receive scenario, as outlined in Chapter 3. However, it is crucial to assess not only the functional behavior but also the temporal behavior of the proposed modules. Therefore, the proposed modules are evaluated through a scenario involving parameters such as application delay and the relationship between BAG and Packet Loss.

4.1 Scenario

To validate the proposed OMNET++ IMA module, a Navigation and Guidance System scenario described in [2] is examined. This scenario involves five applications: Anemometer (Anemo), Flight Warning (FW), Flight Manager (FM), Multifunction Control Display Unit (MCDU), and Autopilot (AP). The Anemo and FW applications reside within the same IMA module (Module 1), while others are distributed across different IMA modules (Module 2,3,4).

In this scenario, sensor data which are altitude (Z) and broadcast speed (M), are obtained by the Anemo (module1), and they are sent to the AP (module4) using VL 1 as separate frames. While FW (module1) sends equipment status to MCDU (module3) through VL 2 in order to keep, the crew informed about the aircraft's situation, FM (module2) transmits the upcoming position that the aircraft should reach to the AP via VL 3. Additionally, the crew can send a new flight plan using the MCDU, which sends frames with VL 4 to the AP. Each VL connection has a BAG value of 2 milliseconds (ms), and the size of each frame is set to 4000 bits. Moreover, the scenario includes two switches that connect the four IMA modules, and the network topology and VL transfer path are depicted in Figure 4.1.

The execution of an application in the given scenario [2] is divided into three sub-steps: reception, processing, and transmission. Each sub-step is allocated a specific time

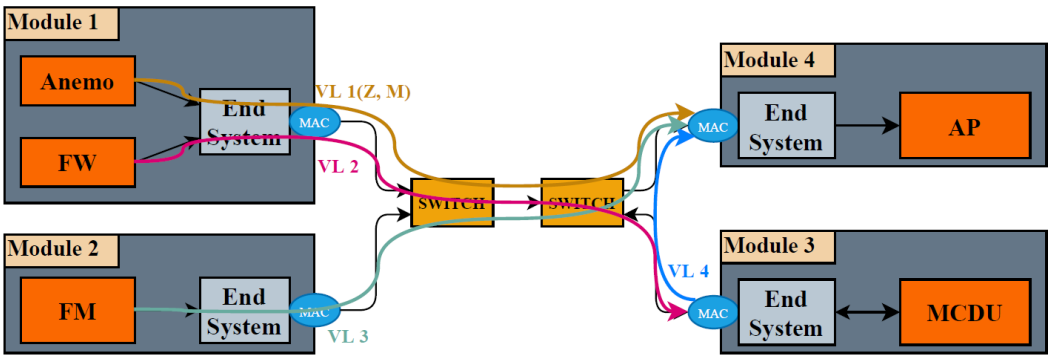


Figure 4.1 : Network Topology.

interval, indicating the best and worst-case completion times. Table 4.1 demonstrates the periods of the applications and the completion time intervals for these sub-steps. For instance, the processing step of Anemo is expected to be completed within a minimum of 3 ms and a maximum of 8 ms after the start of the major time frame. Although the processing step concludes before 8 ms, the transmission step will not begin before 8 ms. Once the module time reaches 8 ms, the transmission step is executed within 2 ms.

Table 4.1 : Applications Information.

Application Name	**Sub steps Interval (ms)**			**Application Period (ms)**
	RX	***Process***	***TX***	
Anemo	[0-3]	[3-8]	[8-10]	20
FW	[12-14]	[14-17]	[17-19]	20
FM	[0-5]	[5-15]	[15-19]	20
MCDU	[0-5]	[5-17]	[17-20]	20
AP	[0-4]	[4-12]	[12-15]	15

In the OMNET++ implementation, each sub-step is treated as an individual event, and the duration of each event is simulated using the *waitAndEnqueue* method, which was explained in Chapter 3. Before executing each sub-step, the *waitAndEnqueue* method is invoked. Thus, the overhead of each sub-step, such as frame transmission and reception, is also simulated. The pseudo-code of the FM's process will be as in Algorithm B.1. Also, the graphical representation of the FM's sub-steps is demonstrated in Figure 4.2.

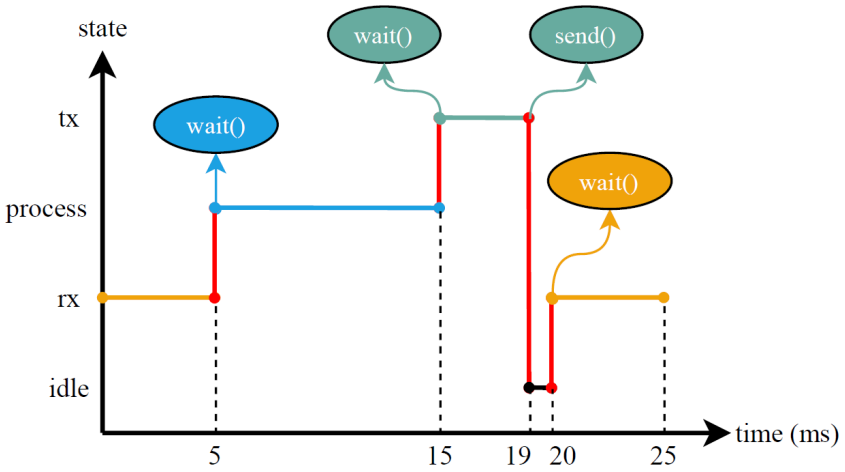


Figure 4.2 : FM Process Timeline.

4.2 Simulation Outcomes

The scenarios where the $D_{swApp}{}^{max}$ occurs for all VLs, as determined by equation 2.4, are depicted in Figure 4.3. Orange blocks represent sender applications, while blue blocks denote receiver applications. The $D_{swApp}{}^{max}$ for VL 1 may be observed in the 8^{th} activation of Anemo. Since the frames, Z and M that will be sent are created at the processing step; the $D_{swApp}{}^{max}$ calculation should start 148 ms after the start of Module 1 for 8^{th} activation. Since Anemo sends the two frames simultaneously, the second frame must wait 2 ms at the ES due to the BAG time. When the second frame leaves ES, the reception step of AP has already begun. Consequently, the second frame waits for the destination application's new period on the sampling port of the destination application. Furthermore, suppose the transmission step of the 9^{th} activation of Anemo is completed after at least 1 ms (169 ms). In that case, the second frame of the 8^{th} activation will not be overwritten by the frames of the 9^{th} activation. Based on these assumptions, the $D_{swApp}{}^{max}$ for VL 1 can be computed as 21 ms. The $D_{swApp}{}^{max}$ for other VLs can be calculated using a similar approach.

The comparison between the simulation and theoretical results for $D_{swApp}{}^{max}$ is presented in Figure 4.4. The results indicate that the OMNET++ simulation outcomes align with the theoretical results, confirming the validation of the developed modules. However, it is essential to note that the simulation results are not identical to the theoretical results. This discrepancy arises from utilizing the *uniform* function in the OMNET++ simulator. The *uniform* function is responsible for generating the

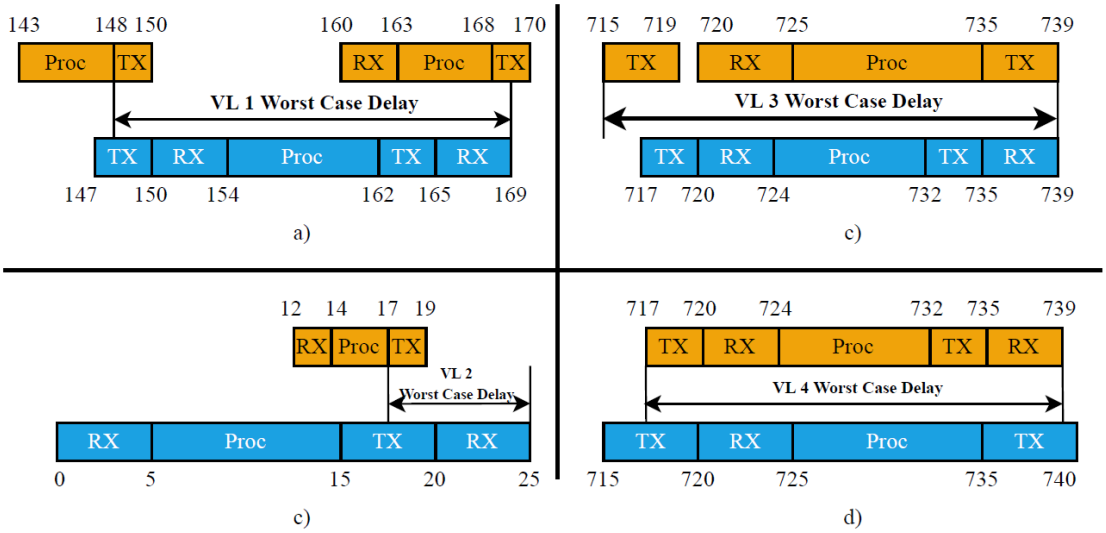


Figure 4.3 : Worst Case Delay Scenario.

execution times for the sub-steps of the applications. The generation of the execution time is performed randomly with a uniform distribution. The function takes two parameters: the lower and upper bounds of the execution time range. While the first parameter sets the lower bound, the second parameter determines the upper bound. The function generates a random time within the range of *[lower bound, upper bound)*, meaning it cannot generate the worst-case execution time. As a result, achieving perfectly matching results between simulation and theory is unattainable because of the *uniform* function.

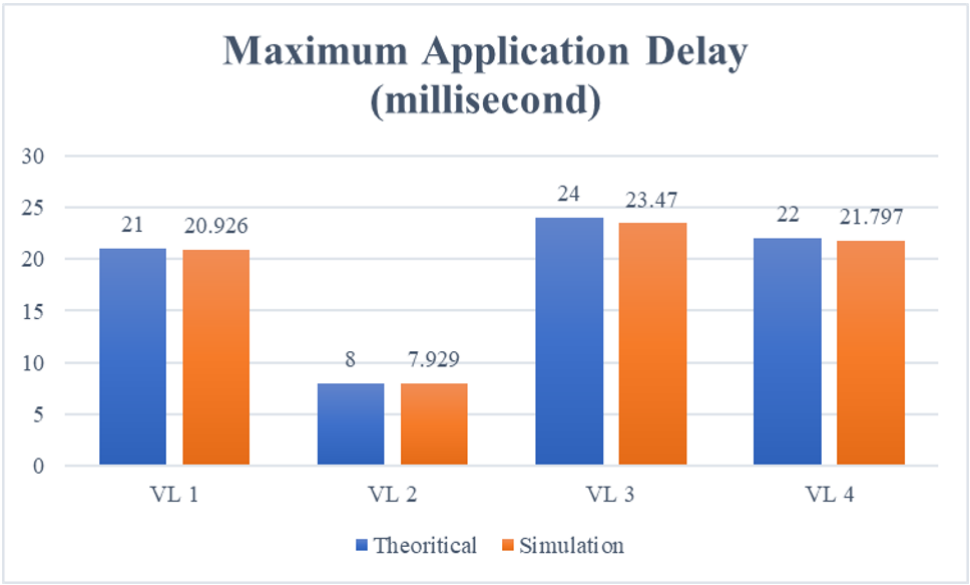


Figure 4.4 : Theoretical vs Simulation Application Delay.

In addition to the application delay, the OMNET++ proposed modules are verified by examining the relationship between BAG and packet loss. Generally, IMA systems should be designed to ensure packet loss does not happen. However, in the given scenario [2], transmitting Z and M in separate frames leads to packet loss in VL 1. The packet loss occurs due to the relationship between the BAG value and the period of the destination partition (AP). Since the BAG value for VL 1 is smaller than the period of the destination partition (AP), some packets overwrite older ones until the subsequent activation of the destination partition (AP). However, the packet loss can be reduced by aligning the BAG value with the period of the destination partition. When the BAG value equals or exceeds the period, the destination application does not experience any packet loss. The packet loss ratio for VL 1 is depicted for various BAG values in Figure 4.5.

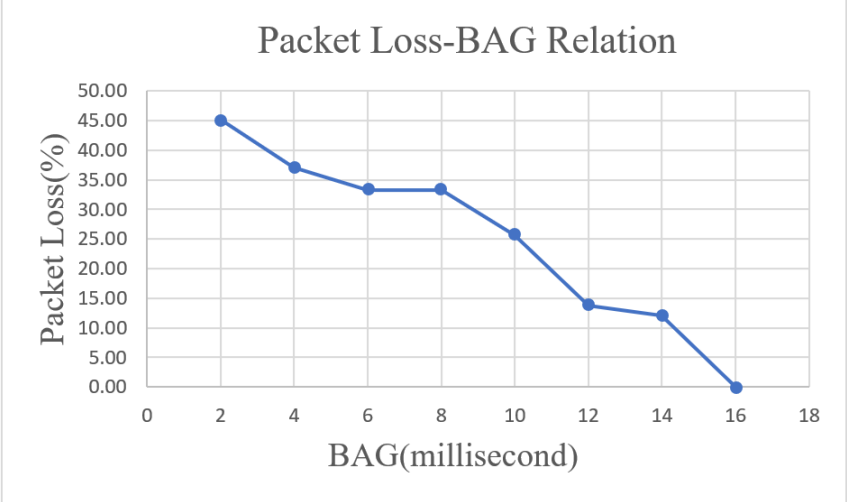


Figure 4.5 : Packet Loss and BAG Relationship.

4.3 Advantages of the IMA Simulation Model

In this section, the contribution of the simulation model to the IMA development process will be examined using the scenario in which simulation validation is performed. Hardware images are utilized instead of the placeholder blocks in the network topology demonstration to show the contribution clearly. The network topology with hardware images is shown in Figure 4.6. While TTTech Switch A664 Lab v2.0 image is used for AFDX switch [27], the ES hardware image is taken by [28].

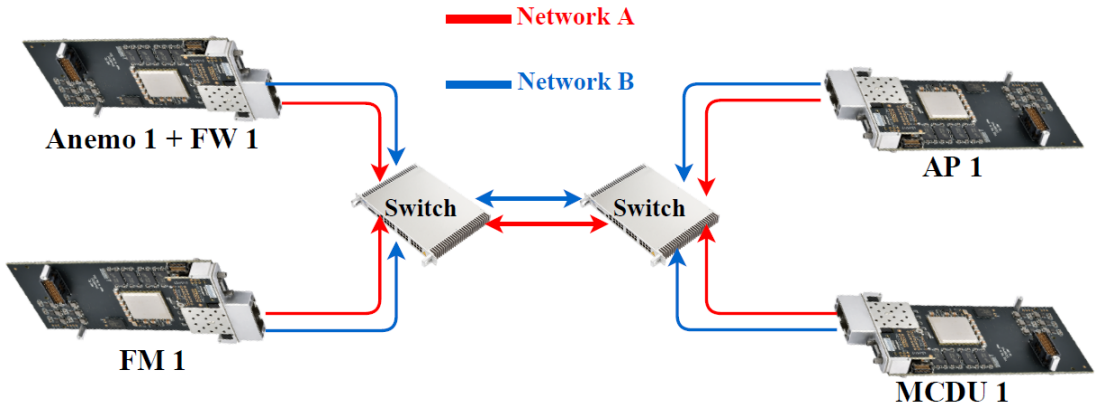


Figure 4.6 : Network Topology with Hardware Images.

In the aviation sector, redundant critical components are located in aircraft to increase fault tolerance. These components can perform the main components' tasks when the main components fail. Also, the redundant components can be used to obtain more accurate data. For example, two sensor systems that produce the same raw data can be used in avionic systems to increase the reliability of the data obtained from sensors. The application that interprets these sensor data applies a sensor fusion algorithm to the raw data for getting more accurate results. When the redundancy method is applied to the Navigation and Guidance System scenario, the network topology with redundant components can be seen in Figure 4.7. It can be easily seen that cable complexity which makes the testing process cumbersome, is increasing with redundant modules. While there are only ten cables in the non-redundant topology, the cable number becomes twenty-four in the redundant network topology. In physical testing, increased cable usage makes the testing setup more error-prone and decelerates the testing process. In addition, as the complexity of IMA systems increases, the physical space required for testing also increases. Conducting physical tests for systems of a certain size becomes nearly impossible. However, since simulation models do not occupy any physical space, engineers can test highly complex systems, such as those implemented in the A380 [10], without hassle.

Furthermore, the configuration of the AFDX switches must be updated with additional modules by adding new port-MAC mappings. Although this update can be done by adding just a few lines of code to the simulation configuration file in the

simulation-based development process, the hardware content of the AFDX switches must be updated in physical testing. For example, the developer must load a newer .bit file to each AFDX switch if FPGA is preferred as hardware in the AFDX switch. Also, simulation can increase the efficiency of the development process in terms of money. For instance, engineers may want to add switches and End Systems to their IMA system. Thanks to the simulation, they can observe the effect of the new network devices addition without buying them physically.

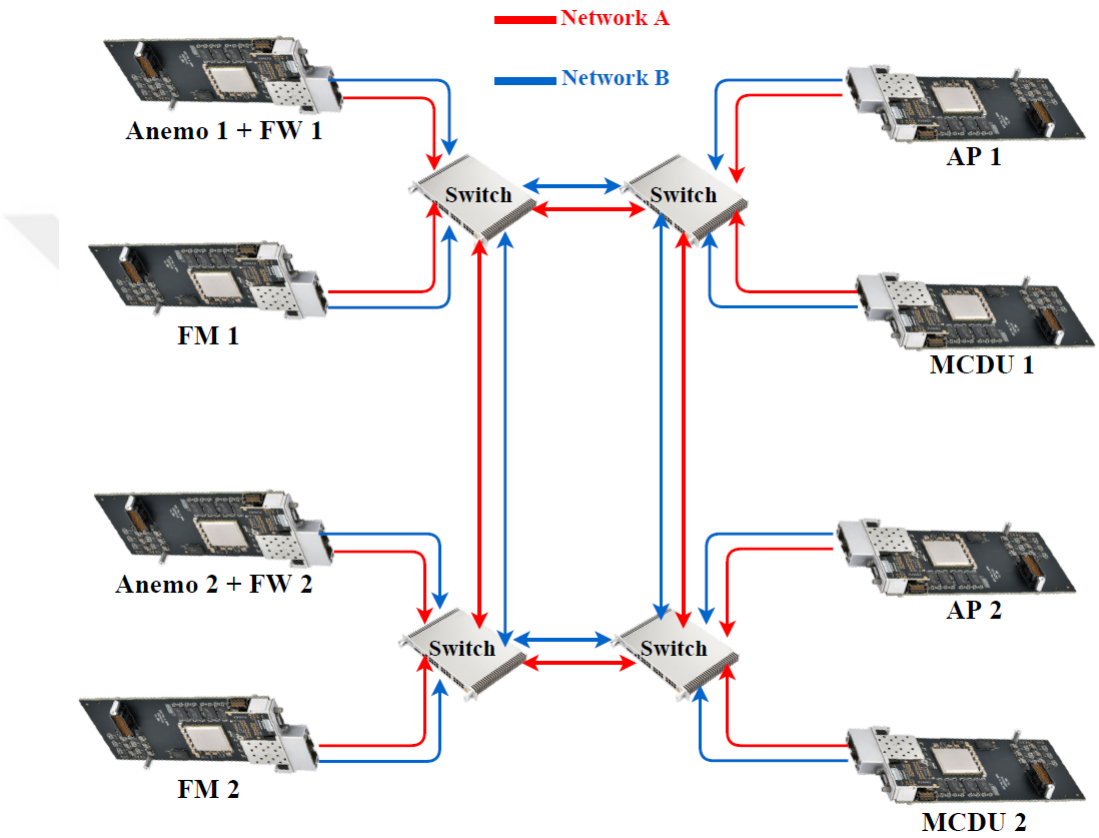
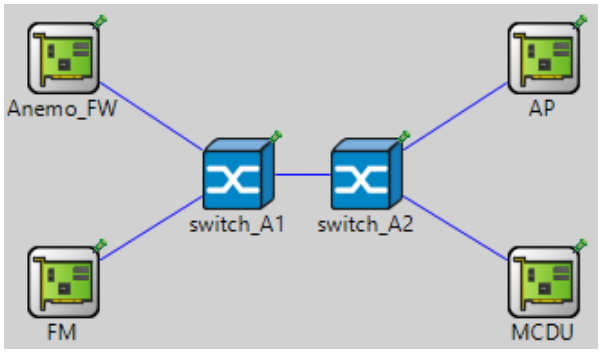
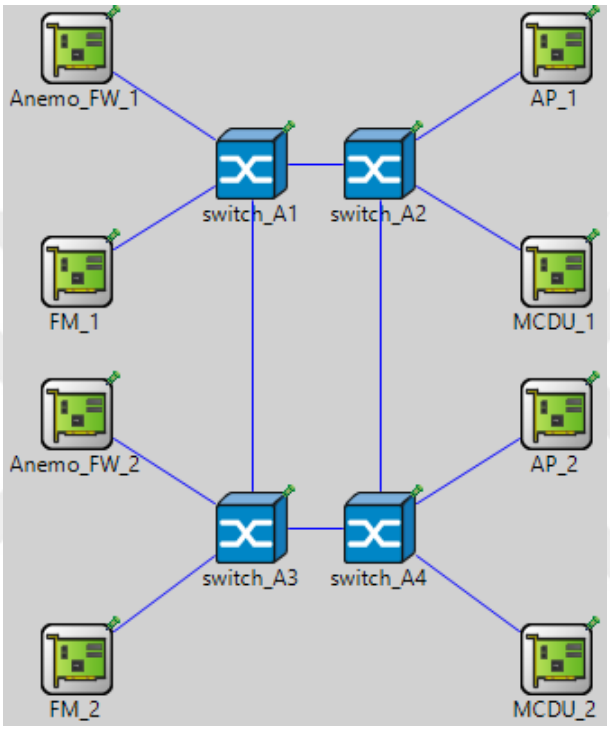


Figure 4.7 : Redundant Network Topology with Hardware Images.

In OMNET++, implementing scenarios with redundancy components is straightforward. The developer only needs to make a copy of the existing components and paste them into the model. After establishing the necessary connections, the components become operational within the model. The scenario's redundant and non-redundant network topology can be seen in the Figure 4.8 for OMNET++.



(a) Non-Redundant OMNET++ Scenario.



(b) Redundant OMNET++ Scenario.

Figure 4.8 : Network Topology of the Scenario for OMNET++.

5. CONCLUSIONS AND FUTURE WORK

When developing an IMA system, it is essential to consider relevant standards. ARINC 653 and AFDX have received significant attention and consideration among these standards. These standards have numerous configuration parameters and offer various design options for engineers and designers. However, determining the optimal system configuration can be a complex task. In this regard, conducting a substantial portion of the IMA system design process within a simulation environment can save limited resources such as time and money.

Several options for selecting a simulation environment are available, including coding a custom simulator using tools like UPPAAL, MATLAB/SIMULINK, OPNet, NS-2, NS-3, and OMNET++. Among these options, open-source network simulators such as NS-3 and OMNET++ appear more favorable due to their code re-usability and source code access. Among the open-source simulators, OMNET++ is chosen for its superior scalability and extensibility compared to the alternatives. Additionally, OMNET++ benefits from an active community and comprehensive frameworks like INET, which provide a detailed implementation of the OSI layer. To the best of our knowledge, this thesis represents the first effort to design the necessary components for building an IMA System within a network simulator. Thanks to this work, the communication performance of an IMA system can now be evaluated without physical implementation. Furthermore, while previous studies have considered various communication overheads such as link-level and buffer delay, they have yet to include the overheads of receiving and sending user calls to their application delay calculations. Including these overheads will enhance the accuracy of the delay calculation.

The ARINC 653 standard explains all the necessary aspects for building a complete operating system, including the management of partitions and processes and communication between them. However, for this thesis, designing a fully compatible O/S adhering to the ARINC 653 standard is not practical or necessary. Instead,

the focus is on evaluating the communication performance by activating partitions and processes following the ARINC 653 standard. While the *partitionManager* and *partition* modules are developed to perform temporal control of the applications, *process* and *processManager* modules are designed for the implementation and management of the tasks that perform communication operations respectively. The functionality of the developed modules is verified through a scenario that includes basic send and receive operations. The objective is to assess whether the transfer of packets can be executed successfully.

Complete validation does not only include functional behavior checking; the temporal behavior of the proposed modules should also be examined. Thus, the modules are tested with a scenario using different metrics: application delay and packet loss ratio. It is observed that the application delay obtained in the simulation environment converged to the theoretical calculations. Also, packet loss is observed in VL 1 of the scenario. The reason for packet loss is the difference between the BAG of VL 1 and the period of the destination application. As expected, packet losses decrease as the BAG value converges to the destination application's period, and when they are equalized, the packet loss is removed. These two outcomes prove that processes are operated at the correct timing.

As future work, the scope of the designed ARINC 653 modules can be expanded. For example, processes are designed to operate aperiodically, and their time capacity is infinite. Thus, implementing a periodic process and adding control of the time duration increases the model's realism. Also, implementing synchronization objects makes the model more flexible for the scenario considering concurrent access. Besides, developing a VL scheduler algorithm that considers the MTF when choosing AFDX packets from the VL queue is possible. This algorithm may reduce latency and improve the fairness of packet egress.

Furthermore, the proposed IMA modules can be used with standards such as Ethernet, Time-Triggered Ethernet (TTEthernet), and Time-Sensitive Networking (TSN). For example, many networking devices in the market only support the Ethernet standard, although most of the IMA system uses TTEthernet and TSN. However, the networking

devices that only support Ethernet should also be integrated IMA system without violating ARINC 653 partitioning method. As the architecture of each network stack may differ, the performance of network communication can vary across different IMA systems. Consequently, selecting an appropriate network topology and network stack architecture becomes challenging as the IMA system grows in complexity. The proposed IMA modules allow for examining various network stack architectures, topologies, and devices without needing physical implementation. This ability enables the evaluation and exploration of different configurations and setups in a virtual environment, eliminating the need for physical effort.

Also, it is possible to develop a tool that automates the testing process for OMNET++. For instance, instead of the developer manually changing each possible VL BAG value when examining the second validation metric (BAG-Packet Loss Ratio), a software tool can execute each simulation scenario with different BAG values. Additionally, the number and diversity of input parameters of a scenario may be high. However, in such cases, a suitable cost function should be defined to determine the outcomes of simulation scenario tests. Thanks to the automatization tool, optimum input values that the developer may overlook within an extensive input parameter set can be obtained.

REFERENCES

[1] **Akpolat, E.C.**, **Şeker, M.**, **Cevher, S.**, **Şapla, E. and Ata, S.** (2020). An Omnet++ Simulation for Performance Analysis of ARINC 664 P7 Avionics Data Network, *2020 28th Signal Processing and Communications Applications Conference (SIU)*, pp.1–4.

[2] **M. Lauer, J. Ermont, C.P. and Boniol, F.** (2010). Analyzing End-to-End Functional Delays on an IMA Platform, *4th International Symposium on Leveraging Applications*, volume6415, pp.243–257.

[3] **Heimbigner, D. and McLeod, D.** (1985). A Federated Architecture for Information Management, *3*(3), 253–278, `https://doi.org/10.1145/4229.4233`.

[4] **Gaska, T.**, **Watkin, C. and Chen, Y.** (2015). Integrated Modular Avionics - Past, present, and future, *IEEE Aerospace and Electronic Systems Magazine*, *30*(9), 12–23.

[5] **Wind River Systems** (2022). *VxWorks 653 Multi-core Edition*, Wind River Systems, Inc., white paper.

[6] **TUBITAK-BILGEM**. `https://bilgem.tubitak.gov.tr/en/content/rtos/gis-real-time-operating-system`, date retrieved: 09.05.2023.

[7] **Airlines Electronic Engineering Committee (AEEC)** (2010). *ARINC 653 Avionics Application Software Standard*, ARINC Specification 653, part 1.

[8] **Fuchs, C.M.**, **Schneele, S. and Klein, A.** (2012). The Evolution of Avionics Networks From ARINC 429 to AFDX.

[9] **Gaska, T.**, **Watkin, C. and Chen, Y.** (2018). Model-driven Development and Simulation of Integrated Modular Avionics (IMA) Architectures, *EUROSIM Scientific Membership Journal*, *28*(2), 61–66.

[10] **Butz, H. and Sas, A.** (2007). THE AIRBUS APPROACH TO OPEN INTEGRATED MODULAR AVIONICS (IMA): TECHNOLOGY, METHODS, PROCESSES AND FUTURE ROAD MAP.

[11] **Badache, N.**, **Jaffres-Runser, K.**, **Scharbarg, J.L. and Fraboul, C.** (2014). Managing temporal allocation in Integrated Modular Avionics, *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pp.1–8.

[12] **Rejeb, N.**, **Ben Salem, A.K. and Ben Saoud, S.** (2017). AFDX simulation based on TTEthernet model under OMNeT++, *2017 International Conference on Advanced Systems and Electric Technologies (IC$_A$SET), pp.*423 − −429.

[13] **Yang, Q.**, **Lu, H. and Tu, X.** (2020). Simulation and Experiment of AFDX Network Based on OMNeT++, *2020 Chinese Automation Congress (CAC)*, pp.5849–5854.

[14] **Li, X. and Xiong, H.** (2009). Modelling and simulation of integrated modular avionics systems, *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*, pp.7.B.3–1–7.B.3–8.

[15] **Xiang, W. and He, F.** (2018). End-to-End Delay Analysis Considering Partition Scheduling On a DIMA Platform, *TENCON 2018 - 2018 IEEE Region 10 Conference*, pp.1548–1553.

[16] **Distributed, E. and (DEIS), I.S.** `https://docs.uppaal.org/`, date retrieved: 25.04.2023.

[17] **P. Han, Z. Zhai, B.N.U.N. and Kristjansen, M.** (2019). Schedulability Analysis of Distributed Multicore Avionics Systems with UPPAAL, *Journal of Aerospace Information Systems*, *16*(11), 1–27.

[18] **Corraro, G.**, **Bove, E.**, **Garbarino, L. and Memoli, E.** (2018). A novel approach for the development and coding of avionics functionalities for IMA architectures, *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pp.1–8.

[19] **Gökçe, P, I.** (2023). *AFDX (AVIONICS FULL-DUPLEX SWITCHED ETHERNET) NETWORK SIMULATION AND PERFORMANCE ANALYSIS*.

[20] **Akpolat, E.C.**, **Faruk Gemici, , Demir, M.S.**, **Hökelek, , Coleri, S. and Çırpan, H.A.** (2021). Genetic Algorithm Based ARINC 664 Mixed Criticality Optimization Using Network Calculus, *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp.1–6.

[21] **Yeniaydın, M.**, **Gemici, F.**, **Demir, M.S.**, **Hökelek, , Coleri, S. and Tureli, U.** (2021). Priority Re-assignment for Improving Schedulability and Mixed-Criticality of ARINC 664, *2021 IFIP Networking Conference (IFIP Networking)*, pp.1–6.

[22] **Bauer, H.**, **Scharbarg, J.L. and Fraboul, C.** (2009). Applying and optimizing trajectory approach for performance evaluation of AFDX avionics network, *2009 IEEE Conference on Emerging Technologies Factory Automation*, pp.1–8.

[23] **Lu, J.**, **Xiong, H.**, **He, F. and Wang, R.** (2020). Enhancing Real-Time and Determinacy for Network-Level Schedule in Distributed Mixed-Critical System, *IEEE Access*, *8*, 23720–23731.

[24] **Airlines Electronic Engineering Committee (AEEC)** (2009). *ARINC 664 P7-1 Avionics Full-Duplex Switched*, ARINC Specification ARINC 664 P7-1, part 7.

[25] **Discrete-event simulation**. https://en.wikipedia.org/wiki/Discrete-event_simulation, Retrieved May 1, 2023, from http://en.wikipedia.org.

[26] **David Wu, A.V.** https://doc.omnetpp.org/omnetpp/manual, date retrieved: 01.05.2023.

[27] **TTTech**. *TTE-Switch A664 Lab v2.0*, https://www.tttech.com/aerospace/products/switches/tte-switch-a664-lab-v2-0.

[28] **Speedgoat**. *IO781: ARINC-664 Part 7 / AFDX protocol support with Simulink*, https://www.speedgoat.com/products/communication-protocols-arinc-afdx-io781.

APPENDICES

APPENDIX A : Pseudo Code of the OMNET++ Modules
APPENDIX B : A process' co-routine of the Example Scenario

APPENDIX A : Pseudo Code of the OMNET++ Modules

Algorithm A.1 Activity Methods of Simple Send&Receive Process

```
/* The co-routine of the transmitter process.*/
function ACTIVITY
    while 1 do
        msg ← RECEIVE();
        if msg == StopProcess then
            processState ←PASSIVE;
        else if msg == StartProcess then
            processState ←ACTIVE;
            WAITANDENQUEUE(WAITACTIONTIME, &WAITQUEUE);
            if Stop Process In waitQueue then
                CONTINUE();
            else
                afdxMsg ← CREATEJOBFORMODULE();
                SEND(afdxMsg, APEXPORTNAME, APEXPORTINDEX);
                RESCHEDULEPROCESS();
            end if
        end if
    end while
end function

/* The co-routine of the receiver process.*/
function ACTIVITY
    while 1 do
        msg ← RECEIVE();
        if msg == StopProcess then
            processState ←PASSIVE;
        else if msg == StartProcess then
            processState ←ACTIVE;
            WAITANDENQUEUE(WAITACTIONTIME, &WAITQUEUE);
            if Stop Process In waitQueue then
                CONTINUE();
            else
                afdxMsg ← RECVAFDXMSG();
                RESCHEDULEPROCESS();
            end if
        end if
    end while
end function
```

Algorithm A.2 Initialize and Activity Methods of Partition Manager

```
/*Construct Data Structures and Send a Self-Message for the First Partition*/
function INITIALIZE
    partitionNumber ← GATESIZE(SCHEDULER_OUT);
    modulePartitionInfo ← NEW PARTITIONINFO[partitionNumber];
    for i = 0 to partitionNumber do
        partitionGate ← GATE("SCHEDULER_OUT$O", i)->GETNEXTGATE();
        partitionModule ← partitionGate->GETOWNERMODULE();
        partitionId ← partitionModule->PAR("ID");
        partitionsGate[partitionId] ←i;
    end for
    majorFrameXML ← PAR("MAJORFRAMEXML").XMLVALUE();
    xmlPartition ← majorFrameXML->GETELEMENTSBYTAGNAME("PARTITION");
    for partitionIterator = xmlPartition.begin to xmlPartition.end do
        modulePartitionInfo[i].ID ← partitionIterator->GETATTRIBUTE("ID");
        modulePartitionInfo[i].duration ← partitionIterator->GETATTRIBUTE("DURATION");
        modulePartitionInfo[i].offset ← partitionIterator->GETATTRIBUTE("OFFSET");
        i++;
    end for
    prevPartitionIndex ← −1;
    nextPartitionIndex ← 0;
    SCHEDULEAT(SIMTIME() + modulePartitionInfo[nextPartitionIndex].OFFSET, TIMERMSG);
end function

/* The co-routine of the processManager.*/
function ACTIVITY
    msg ←NULL
    while 1 do
        msg ← RECEIVE();
        if prevPartitionIndex > 0 then
            prevPartitionId ←modulePartitions[prevPartitionIndex].partitionId;
            SENDDELAYED(STOPMSG,BIGCONTEXTSWITCH,SCHEDULER_OUT$O,
            partitionsGate[prevPartitionId]);
        end if
        if nextPartitionIndex > 0 then
            nextPartitionId ←modulePartitions[nextPartitionIndex].partitionId;
            SENDDELAYED(STARTMSG,BIGCONTEXTSWITCH,SCHEDULER_OUT$O,
            partitionsGate[nextPartitionId]);
        end if
        prevPartitionIndex ← CALCULATENEXTINDEX();
        nextPartitionIndex ← CALCULATEPREVINDEX();
        timeOffset ← CALCULATETIMEOFFSET();
        SCHEDULEAT(SIMTIME() + timeOffset, TIMERMESSAGE);
    end while
end function
```

Algorithm A.3 Process Manager Methods

```
/* The method deactivates all connected process.*/
function DEACTIVATEALLPROCESS
    for i = 0 to processNumber do
        SETPROCESSSTATE(i, PASSIVE);
        SENDDELAYED("PROCESS STOP", processSwitchTime, i);
    end for
end function

/* The method activates all connected process.*/
function ACTIVATEALLPROCESS
    for i = 0 to processNumber do
        SETPROCESSSTATE(i, ACTIVE);
    end for
end function

/* The method selects an active process with the highest priority for all connected
processes.*/
function SCHEDULEPROCESS
    currPrio ← −1;
    procIndex ← −1;
    for i = 0 to processNumber doPRIO(i)
        if ISACTIVE(i) and currPrio < PRIO(i) then
            currPrio ← PRIO(i)
            procIndex ← i;
        end if
    end for

    if procIndex ≠ −1 then
        sendDelayed("PROCESS START", processSwitchTime, i);
    end if
end function

/* The co-routine of the process manager process.*/
function ACTIVITY
    while 1 do
        msg ← receive();
        if msg == StopProcess then
            deactivateAllProcess();
        else if msg == StartProcess then
            deactivateAllProcess();
            scheduleProcess();
        else
            throwcRuntimeError("PARTITION RECEIVE UNDEFINED MESSAGE!");
        end if
    end while
end function
```

APPENDIX B : A process' co-routine of the Example Scenario

Algorithm B.1 The co-routine of the FM process

```
function ACTIVITY
    while 1 do
        msg ← receive();
        if msg == StopProcess then
            processState ←PASSIVE;
        else if msg == StartProcess then
            processState ←ACTIVE;
            /* Wait for the rx substep of the FM */
            WAITRANDOMTIME(0, GETRXTIME(), WAITTIME);
            if Stop Process NOT in waitQueue then
                afdxMsg ← CREATEJOBFORMODULE();
                /* Wait for the remaining time of the FM's rx substep */
                WAITREMAININGTIME(GETRXTIME(), WAITTIME);
                if Stop Process NOT in waitQueue then
                    /* Wait for the process sub-step of the FM */
                    WAITRANDOMTIME(0, GETPROCESSTIME(), WAITTIME);
                    if Stop Process NOT in waitQueue then
                        /* Wait for the remaining time of the FM's process substep */
                        WAITREMAININGTIME(GETPROCESSTIME(), WAITTIME);
                        if Stop Process NOT in waitQueue then
                            /* Wait for the tx sub-step of the FM */
                            WAITRANDOMTIME(0, GETTXTIME(), WAITTIME);
                            SEND(afdxMsg, APEXPORTNAME, APEXPORTINDEX);
                            RESCHEDULEPROCESS();
                        end if
                    end if
                end if
            end if
        end if
    end while
end function
```

CURRICULUM VITAE

Name SURNAME: Mümin Göker GAYRETLİ

EDUCATION:

- **B.Sc.:** 2019, Istanbul Technical University, Electric Electronics Faculty, Electronics and Communication Engineering
- **B.Sc.:** 2020, Istanbul Technical University, Computer and Informatics Faculty, Computer Engineering (Double Major)

PROFESSIONAL EXPERIENCE:

- 2019, Graduated with second-highest honors from ITU Electronics and Communication Engineering
- 2019-2021, Part-Time Embedded Developer at ITU Aerospace Research Center
- 2020-2023, Software Engineer at the TUBITAK BILGEM

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **Gayretli, M. G.**, Yeniçeri, R., and Demir, M. S., (2023). Partitioning Network Stack Implementation in OMNET++, *2nd INTERNATIONAL GRADUATE RESEARCH SYMPOSIUM IGRS'23*, March 16-18, 2023 Istanbul, Turkey.
- **Gayretli, M. G.**, Yeniçeri, R., Demir, M. S., and Hökelek, I, (2023). An OMNET++ Simulation Model for IMA Systems, *11nd IEEE International Black Sea Conference on Communications and Networking*, July 5-7, 2023 Istanbul, Turkey.

OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:

- Hüner, Y., **Gayretli, M. G.**, and Yeniçeri, R, (2021). HW/SW Design Space Exploration of A Complementary Filter on Zynq SoC, *8th International Conference on Electrical and Electronics Engineering (ICEEE)*, Antalya, Turkey, pp. 1-5.