Pós-Graduação em Ciência da Computação

Paulo Freitas de Araujo Filho

Contributions to In-vehicle Networks: Error Injection and Intrusion Detection System for CAN, and Audio Video Bridging Synchronization

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2018

Paulo Freitas de Araujo Filho

Contributions to In-vehicle Networks: Error Injection and Intrusion Detection System for CAN, and Audio Video Bridging Synchronization

M.Sc. Dissertation presented to the Centro de Informática of the Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Master of Computer Science.

Advisor: Divanilson Rodrigo de Sousa Campelo

Recife

2018

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

A663c Araujo Filho, Paulo Freitas de
Contributions to in-vehicle networks: error injection and intrusion detection system for CAN, and audio video bridging synchronization / Paulo Freitas de Araujo Filho. – 2018.
94 f.: il., fig., tab.

Orientador: Divanilson Rodrigo de Sousa Campelo.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2018.
Inclui referências.

1. Redes de computadores. 2. Técnicas de injeção de erros. I. Campelo, Divanilson Rodrigo de Sousa (orientador). II. Título.

004.6 CDD (23. ed.) UFPE- MEI 2018-111

Paulo Freitas de Araujo Filho

Contributions to In-Vehicle Networks: Error Injection and Intrusion Detection System for CAN, and Audio Video Bridging Synchronization

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 04/07/2018.

BANCA EXAMINADORA

_____

Prof. Abel Guilhermino da Silva Filho
Centro de Informática / UFPE

_____

Prof. Eduardo Coelho Cerqueira
Faculdade da Engenharia de Computação e Telecomunicação / UFPA

_____

Prof. Divanilson Rodrigo de Sousa Campelo
Centro de Informática / UFPE
(**Orientador)**

I dedicate this dissertation to all my family, friends and professors who gave me the necessary support to get here.

ACKNOWLEDGEMENTS

I would like to thank God for the guidance and strength.

To my mom and dad, Ana Rosa and Paulo Araujo, for giving me all the support and assistance I could possibly need, while also pushing me for reaching my best.

To my sister and brother in-law, Juliana and Paulo Mendes, for being there for me no matter what.

To my niece, Letícia, who brings the happiness and joy of a child.

To my fiancée, Aline, who is also always there for me as a source of support, companion and love.

To all my cousins, family and friends that are there for me in the best, but also on the worst and more difficult moments.

To my advisor, Divanilson Campelo, who provided me as many career opportunities as he could and gives me the support and guidance, not only from a professor, but also from a friend.

To Edilson, Fábio, David, Rodrigo, Caio, Luís and Eron that helped me during the development of this project.

To Colt Correa, Dave Robins, and Laks, from Intrepid Control Systems, who gave me the opportunity to apply my academic knowledge to solve problems from the industry, and from whom I learned so much.

To the examining board professors Eduardo Cerqueira and Abel Guilhermino.

To CNPq for the financial support and to CIn for the structure and research environment.

To all of you, thank you so much.

Paulo Freitas

ABSTRACT

Vehicles are equipped with Electronic Control Units (ECUs) responsible for performing tasks as simple as lighting up brake lights or as complex as controlling the wheels of an electric car. The exchange of information between ECUs uses the Controller Area Network (CAN) and the Controller Area Network with Flexible Data Rate (CAN FD), which provides higher data rates and payloads, as the main and most used intra-vehicular networks, at least until today. Interconnected ECUs must work perfectly and interact with each other as well as with other car components in a reliable way, thus it is indispensable to test and predict the behavior of these units in error situations. For this, an error injection mechanism can be very advantageous for checking various error conditions in real-world scenarios that affect the safety of vehicles. Furthermore, nowadays, cars are also equipped with network technologies that provide connectivity to the external world. This offers numerous possibilities in terms of new applications and services to be provided, however makes the car a network node subject to cyber-attacks. It is then necessary to provide security mechanisms to prevent, or at least detect, attacks. Besides CAN and CAN FD networks, the advent of 100BASE-T1 Ethernet has feasible the outcome of many new automotive applications with higher bandwidth demands. In order to be use Ethernet in applications that require determinism, a series of IEEE standards, which together constitute the Audio Video Bridging (AVB), were developed and proposed. The IEEE 802.1AS, for instance, is the AVB standard that defines the generalized Precision Time Protocol (gPTP), responsible for node synchronization within AVB networks. In this context, this dissertation intends to make contributions for CAN/CAN FD networks as well as for the automotive Ethernet. For CAN networks, it proposes a novel Error Injection Technique to assist with system level validation tests and also an Intrusion Detection System based on machine learning algorithms. For automotive Ethernet, it designs and proposes a hardware implementation of the gPTP protocol that achieves the required nanoseconds precision, while also providing implementation details for future researches on that protocol.

Key-words: Controller Area Network (CAN). Error Injection Techniques. Intrusion Detection Systems (IDS). Audio Video Bridging (AVB). AVB Networks Synchronization. IEEE 802.1AS.

RESUMO

Veículos são equipados com unidades de controle eletrônico (ECUs) responsáveis por executar tarefas tão simples quanto acender as luzes de freio, ou tão complexas quanto controlar as rodas de um carro elétrico. A troca de informações entre ECUs utiliza a Controller Area Network (CAN) e a Controller Area Network with Flexible Data Rate (CAN FD), com maior taxa de transmissão e carga útil, como principais e mais utilizadas redes intra-veiculares, pelo menos por enquanto. ECUs interconectadas devem trabalhar perfeitamente e interagir umas com as outras, bem como com outros componentes veiculares, de forma confiável, sendo então imprescindível testar e prever o comportamento dessas unidades em situações de erro. Para isso, um mecanismo de injeção de erro pode ser muito vantajoso para verificar várias situações de erro em cenários reais, que possam afetar a segurança do veículo. Além disso, atualmente, os carros também são equipados com tecnologias de redes que proveem conectitividade com o meio exterior. Essa conectividade oferece inúmeras possibilidades em termos de novas aplicações e serviços a serem oferecidos, contudo, torna os carros sujeitos a ataques cibernéticos. É necessário então prover mecanismos de segurança para prevenir, ou ao menos detectar, ataques. Além das redes CAN e CAN FD, o advento da Ethernet 100BASE-T1 tem viabilizado uma grande gama de aplicações automotivas com maiores demandas de banda. A fim de usar a Ethernet para aplicações com requisitos temporais e determinísticos, uma série de padrões do IEEE, os quais juntos compõem o Audio Video Bridging (AVB), foi desenvolvida e proposta. O IEEE 802.1AS, por exemplo, é o padrão do AVB que define o generalized Precision Time Protocol (gPTP), responsável pela sincronização de nós em redes AVB. Esta dissertação propõe contribuições tanto para as redes CAN/CAN FD, como também para a Ethernet automotiva. Para redes CAN, são propostos uma nova técnica de injeção de erros, para auxiliar em testes de validação em nível de sistema, e um sistema de detecção de intrusão baseado em algoritmos de aprendizagem de máquina. Para a Ethernet automotiva, é proposta uma implementação em hardware do protocolo gPTP que atinge os requisitos de precisão de nano-segundos, enquanto que também oferecendo detalhes de implementação necessários para futuras pesquisas sobre o protocolo em questão.

Palavras-chaves: Controller Area Network (CAN). Técnicas de Injeção de Erros. Sistemas de Detecção de Intrusão (IDS). Audio Video Bridging (AVB). Sincronização em redes AVB. IEEE 802.1AS.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADAS	Advanced Driver Assistance Systems
AVB	Audio Video Bridging
AVTP	Audio Video Transport Protocol
BE	Best Effort
BMCA	Best Master Clock Algorithm
BRAM	Block RAM
CAN	Controller Area Network
CAN FD	Controller Area Network with Flexible Data Rate
CBS	Credit Based Shaper
CLBs	Configurable Logical Blocks
CRC	Cyclic Redundancy Check
CSN	Coordinated Shared Networks
DLC	Data Length Code
DSP	Digital Signal Processing
ECU	Electronic Control Unit
EPON	Ethernet Passive Optical Network
FF	Flip-Flop
FIFO	First-in First-out
FPGA	Field Programmable Gate Array
gPTP	generalized Precision Time Protocol
IDS	Intrusion Detection System
iForest	Isolation Forest
ILA	Integrated Logic Analyzer
IP	Intellectual Property
IPS	Intrusion Prevention System
LIN	Local Interconnect Network
LUTRAM	Look-up Table RAM
LUTs	Look-up Tables

MD	Media Dependent
MDT	Maximum Detection Time
MI	Media Independent
MOST	Media Oriented Systems Transport
OCSVM	One-class Support Vector Machine
PL	Programmable Logic
PS	Processing System
PTP	Precision Time Protocol
QoS	Quality of Service
RC	Rate Constrained
RSE	Rear Seat Entertainment
RTOS	Real Time Operating System
SJW	Synchronization Jump Width
SoC	System-on-a-chip
SRP	Stream Reservation Protocol
SVM	Support Vector Machine
TSN	Time Sensitive Networking
TT	Time-Triggered
UTP	Unshielded Twisted Pair
UTSP	Single Unshielded Twisted Pair

CONTENTS

1 INTRODUCTION

1.1 PRELIMINARY BACKGROUND AND RELATED WORKS

Today's vehicles are equipped with Electronic Control Units (ECUs), which are responsible for performing tasks as simple as lighting up brake lights or as complex as controlling the wheels of an electric car. In the early days, ECUs operated stand-alone by performing single and unique tasks. With the introduction of new functionalities in the car that demanded exchange of information between ECUs, stand-alone ECUs were no longer a viable solution. Initially, the exchange of information among ECUs was through point-to-point links, which quickly were proven to be inappropriate as the amount of cables increased exponentially with the number of ECUs and, consequently, impacted the cost and the weight of vehicles. An approach that provided a shared medium among ECUs was needed.

Upon this scenario, the Controller Area Network (CAN) protocol was developed by BOSCH, which adopted an open licensing policy and a good cooperation with semiconductor companies that contributed for a large availability of CAN controllers and transceivers to the industry (MATHEUS; KÖNIGSEDER, 2015). Because of that, the CAN protocol was quickly and largely adopted not only by the automotive industry, but also by the industrial automation, aerospace and medical engineering industries. Moreover, despite new automotive applications that require higher data rates, the CAN technology remains, at least until today, as the main and most used technology for intra-vehicular networks, as it is a cost-efficient and industry-proven solution (NAVET; SIMONOT-LION, 2013). In fact, based on the CAN protocol, the Controller Area Network with Flexible Data Rate (CAN FD) protocol was developed to be compatible and to coexist with CAN networks. By offering higher data rates and payloads than traditional CAN systems, CAN FD has attracted the attention from the automotive industry as one of the prominent candidates for the next-generation in-vehicle networks responsible for control signals (WOO et al., 2016).

A fundamental aspect of interconnected ECUs is that they must work perfectly and interact with each other as well as with other car components like sensors and actuators in a reliable way. For this reason, one of the main tasks for car manufacturers and component suppliers regarding ECUs is to predict the behavior of these units in error situations. For this, an error injection mechanism can be very advantageous for checking various error conditions in real-world scenarios that may affect the safety of passengers.

In this context, the literature has provided a number of error injection approaches for CAN. An on-the-fly error injection mechanism is proposed in (CHANDRAN; GUDDETI; SADASHIVAIAH, 2016) to help validation at the system level in CAN/CAN FD networks. It emulates an error injection module, which replaces the input signal of a node by a

signal containing the injected error. This approach, however, can only test a device per time. Furthermore, the work in (GESSNER et al., 2014) presents the implementation of a physical fault injector for CAN. It relies in a star topology and it is focused exclusively on CAN, not CAN FD networks. Other works, such as the ones in (NOVÁK, 2009), (LUO et al., 2009) and (NOVAK; FRIED; VACEK, 2002), present error generation and injection techniques for CAN networks. None of them, however, allows the creation of a list of frames to be preserved in tests or details how to support different bit rates.

Along with intra-vehicular network technologies for ECUs communication, some of today's cars are also equipped with network technologies, such as Bluetooth, 4G/5G and IEEE 802.11p, among others, that provide connectivity to the external world. This connectivity offers numerous possibilities in terms of new applications and services that can be provided, such as tracking and traffic lights detection systems. On the other hand, it also makes the car a network node subject to cyber-attacks, what could potentially put in risk people's safety. Therefore, it is necessary to provide security mechanisms to prevent, or at least detect, attacks.

In this dissertation, we present a novel approach to generate and inject errors in CAN/CAN FD networks without the need of connecting multiple additional devices. The technique acts directly on the CAN/CAN FD bus to enable the smash or corruption of specific bits of specific frames. As a result, by means of a single tool, testers can generate and inject on the bus many error conditions in real time for validating safety requirements at the system level. In addition, we also present an Intrusion Detection System (IDS), based on machine learning algorithms, that successfully detects frames that are injected in CAN networks and may jeopardize the driver's and passengers' safety. The proposed systems are implemented using low-cost development boards and real CAN networks. When used together, both systems may constitute an Intrusion Prevention System (IPS) in a way that intrusion frames are detected and corrupted in real time by the injection of error frames.

Both CAN and CAN FD protocols are message oriented broadcast networks, mainly used for control data. While CAN networks have a single data rate and can transfer up to 8 bytes of data at a maximum rate of 1Mbit/s, CAN FD networks can switch between two data rates in order to transfer up to 64 bytes of data at a maximum rate of 10Mbit/s. Besides these two intra-vehicular protocols, other ones, such as the Local Interconnect Network (LIN), Media Oriented Systems Transport (MOST) and FlexRay, were also developed and applied to the automotive domain. Each has its own advantage and purpose, such as cost, data rates beyond 10Mbps or deterministic behavior.

The complexity brought by this multi-protocol architecture as well as by the rising of new and more complex automotive applications have significantly increased the amount of software to be flashed into the ECUs. The increase was such that the time needed for programming or updating an Electronic Control Unit (ECU) firmware became a problem

and a new strategy for this was needed. Initially as a solution for programming ECUs and also for diagnostics, engineers developed the automotive Ethernet, which was then standardized as the 100BASE-T1 (STANDARD, 2016), a physical layer for 100Mbit/s Ethernet networks suitable for the hostile environment of a car.

The emergence of Automotive Ethernet makes viable the introduction of novel car functionalities that demand high bandwidth. However, since Ethernet was not originally conceived to support deterministic data, a few strategies to overcome this limitation needed to be proposed. A series of IEEE standards, which together constitute the Audio Video Bridging (AVB) protocols, were developed to provide time-synchronized streaming of audio and video using IEEE 802.3 Ethernet. Soon, it was realized that the AVB technology had the potential for being applied not only in multimedia applications, but also on time sensitive applications with hard timing deadlines.

The IEEE 802.1AS is the AVB standard that defines the generalized Precision Time Protocol (gPTP), which is responsible for node synchronization within an AVB network. Even though this protocol has a great importance and directly affects a system's capacity of accomplishing synchronized activity such as turning the wheels of an electric car, there are not many implementations of this protocol available for the industry. Most are proprietary technologies that offer no implementation details and others do not offer the demanded nanoseconds precision for some systems with strict timing requirements.

In this dissertation, we also propose and present a hardware based implementation for the gPTP protocol targeting AVB end points. The intention is to provide a gPTP implementation that achieves the nanosecond precision specified in the IEEE 802.1AS standard, and that can be used to synchronize nodes with hard synchronization requirements. The synchronization of the wheels of an electric vehicle, for example, cannot afford any lack of synchronization, which may compromise the direction of the wheels and then cause some accident. Moreover, this dissertation provides implementation details that may help researchers to propose improvements and enhancements to the gPTP protocol.

1.2 OBJECTIVES AND GOALS

While making an overview of intra-vehicular networks, from its initial conception and traditional protocols towards the new protocols related to the Automotive Ethernet, this dissertation has three main objectives:

1. Propose a novel Error Injection Technique for CAN/CAN FD networks to assist with system level validation tests;

2. Propose a CAN Network Intrusion Detection System based on machine learning algorithms, that could be used along with the proposed error injection technique for corrupting frames intruders;

3. Propose a hardware implementation of the gPTP protocol that achieves the required nanosecond precision, while also providing implementation details for future researches on that protocol.

1.3 DISSERTATION OVERVIEW

The remainder of this work is described as follows: Chapter 2 explains the fundamental knowledge regarding traditional intra-vehicular networks. Its focus is on CAN and CAN FD networks, which are detailed described. Chapter 3 presents the proposed CAN/CAN FD Error Injection Technique design, implementation and validation tests as well as the proposed machine learning IDS technique for CAN networks and how these two techniques can be used together as an IPS system. Chapter 4 introduces the automotive Ethernet topic by covering its origins, generations and AVB/Time Sensitive Networking (TSN) concepts. Then, it focus on the IEEE 802.1AS AVB standard, which defines the gPTP responsible for synchronizing nodes within AVB networks. Chapter 5 presents the proposed design for the gPTP protocol hardware implementation bringing detailed diagrams and technical explanations. Chapter 6 explains the setup and methodology adopted for testing the gPTP implementation, while also presenting the results obtained from it. Finally, Chapter 7 concludes this dissertation and discusses possible future researches.

1.4 MAIN CONTRIBUTIONS

This section summarizes the author's main contributions to the intra-vehicular networking field during his Master Degree studies:

1. Publication of the paper "Experimental evaluation of cryptography overhead in automotive safety-critical communication", IEEE Vehicular Technology Conference (VTC) 2018, pp. 1-6, Jun 2018, E. A. Silva Jr, P. Freitas de Araujo-Filho, D. R. Campelo;

2. Submission of the paper "A Low-cost Responsive Intrusion Detection System for CAN Networks", to IEEE Communication Letters, June 2018.

3. Design and implementation of an error injection technique that is currently commercialized and used by the automotive industry;

4. Design and implementation of an Audio Codec Driver embedded to an AVB endpoint that is currently commercialized and used by the automotive industry;

5. Hardware implementation for the gPTP protocol that will be commercialized and used by the automotive industry as part of an AVB end-point.

2 TRADITIONAL INTRA-VEHICULAR NETWORKS

By its origins, vehicles were essentially mechanical machines with wheels. In order to add simple features, electronics started to be introduced. Sensors, actuators and ECUs, for controlling them, were introduced and vehicles were equipped with power windows and other simples functionalities.

Back on those days, each new function was implemented and added as a standalone ECU. If data needed to be exchanged between then, point to point links were used. Thus, the number of required connections increased exponentially with the number of ECUs. This approach was inefficient and made the number of wiring to dramatically increase (TUOHY et al., 2015). As a matter of fact, cabling is the third heavier and more expensive component in a car according with (MATHEUS; KÖNIGSEDER, 2015).

To overcome this problem, intra-vehicular networks were developed and introduced. ECUs were then connected to each other using a shared media, such as a network bus for CAN and Flexray. Each intra-vehicular network has its own characteristics, having advantages, but also limitations. In this chapter CAN and CAN FD networks will be extensively covered while other technologies such as LIN, MOST and FlexRay will be discussed more briefly.

2.1 CAN AND CAN FD NETWORKS

The Controller Area Network (CAN) was one of the first intra-vehicular networking technology developed and is largely used until today. It was developed by BOSCH in the 80s being later standardized in ISOs.

Despite its well proven technology and robust operation, it is also important to highlight some other factors that made it to be well accepted and used. Not only by the automotive industry, but also by industrial automation, aerospace and medical engineering industries (MATHEUS; KÖNIGSEDER, 2015). An open licensing policy was adopted, resulting in the technology standardization and accessibility to other companies. A good cooperation between BOSCH and semiconductor companies was established, so a good portfolio of CAN controllers and transceivers were available in the market. BOSCH, being involved in many industry sectors, was a customer for its own technology.

The CAN protocol defines only the first two layers in the OSI model, the physical layer and the data link layer. In the physical layer, it can use only one or two wires. The option with one wire is the single wire low speed CAN network, which can reach up to 125Kbits/s. The two wire configuration is the dual wire high speed CAN network, which can reach up to 1Mbit/s, and, because of that, is the most used one (GMBH.; REIF; DIETSCHE, 2014). Besides the wiring, a CAN transceiver is responsible for reading voltages and translating

it to a state that can be either dominant, logical 0, or recessive, logical 1. In case of the high speed CAN, the voltage observed is the differential voltage between the wires, which are called CAN High and CAN Low.

In the data link layer, it is represented by a CAN controller, connected to the CAN transceiver. This controller is responsible for reading the logical state of the bus (dominant 0 or recessive 1) and decoding the data being transfered. Moreover, it manages whether the bus is busy and the node should wait for transmitting some data or if it can transmit its data right away.

Each node is, therefore, formed by an ECU, a CAN controller and a CAN transceiver. They are then connected to the bus by the CAN High and CAN Low wires, in case of a dual wire high speed CAN network. Observe the illustration in Figure 1.

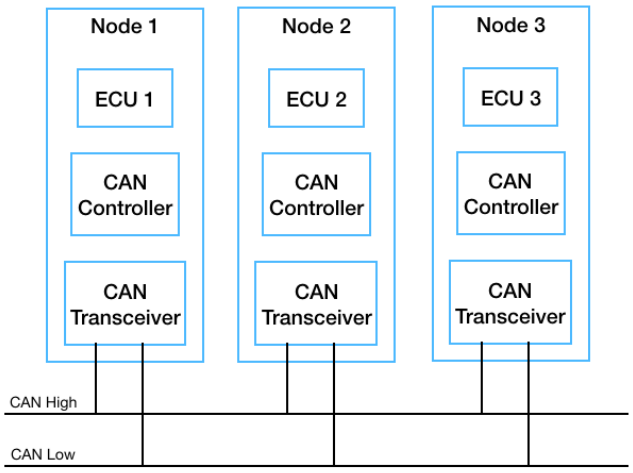

Figure 1 – Illustration of nodes connected to a CAN network

Since the nodes share a common bus, it is necessary to manage somehow which one gets access to the medium and can transmit, while the others simply listen to the bus. This mechanism is defined as arbitration. Each CAN frame can be divided in some fields, each of them composed by a different number of bits. They are, in sequence: Start of Frame field, Arbitration field, Control field, data field, Cyclic Redundancy Check (CRC) field, Acknowledgment field and End of Frame field. The arbitration mechanism occurs during the transmission of the arbitration field bits and is based on a logical AND operation.

Consider, three nodes A, B and C trying to transmit a message. Consider the first 11 bits of each message arbitration field to be as described in table 2.

They all transmit the first bit, and a logical AND operation occurs with these bits resulting in a logical 1, which is actually the bit transmitted by all of them. Then, they transmit the second bit, and a logical AND operation occurs with these bits. This time, the result is a logical 0, which corresponds to the bit transmitted by the nodes B and C. Since node A transmitted a logical 1, it stops transmitting and it is said to has lost the arbitration. After this, only nodes B and C keep transmitting and performing logical

Table 2 – Arbitration field first 11 bits for the messages to be transmitted by nodes A, B and C

	Arbitration Field (first 11 bits)										
	Bit 0	**Bit 1**	**Bit 2**	**Bit 3**	**Bit 4**	**Bit 5**	**Bit 6**	**Bit 7**	**Bit 8**	**Bit 9**	**Bit 10**
Node A	1	1	1	0	1	0	1	0	1	0	1
Node B	1	0	1	0	1	0	0	0	1	1	1
Node C	1	0	1	0	0	1	1	0	1	0	1

AND operations. When bit 4 is reached, the logical AND operation results in a logical 0, then B loses the arbitration and stops transmitting. C is the only node transmitting now. In a simpler way, during the arbitration field, if more than one node wants to transmit, priority is given to the one with a dominant bit in the dispute. The other ones stop the transmission and wait for the bus to be idle again to try to retransmit the frame.

When it comes to addressing, all nodes in a CAN network broadcast their messages, so every node connected to the bus can receive the messages transmitted. The addressing is message-oriented, so, each message has an identifier and each node has a list of message identifiers of interest. In this way, despite being able to receive any message being transmitted in the bus, the nodes only actually get the message whose identifier is in its list of interest.

The message identifier size can vary depending whether the frame is of a base/standard format or extended format. In the former case, the identifier has 11 bits and, consequently, can address 2^{11} messages. In the latter case, the identifier has 29 bits and can address 2^{29} messages.

Besides the base or extended format, there are four different CAN frame types: Data, Remote, Error and Overload frame. Data frames are used to transfer data from a transmitter to a receiver node. Remote frames, on the other hand, are used to request the transmission of data from some node. Error frames are transmitted to indicate that some error condition was detected on the bus. Overload frames are used to synchronize idle detection and provide an extra delay between data and remote frames (ROBERT BOSCH GMBH, 1991).

In addition to the four frame types, data frames and remote frames are preceded by a Interframe Space, formed by three consecutive recessive bits. If, a dominant bit is detected on the bus before the occurrence of three consecutive recessive bits, an overload frame starts. If not, after these three bits, the bus is considered to be in an idle state and a data or remote frame can start upon the appearance of a dominant bit. The dominant bit that marks the start of a data or remote frame is called Start of Frame bit and is the first bit of the frame being transmitted.

Figure 2 describes the first bitst of a data or remote frame in the standard or extended format. It is possible to notice in this figure the location of some bits of interest. For instance, the location of the base identifier and identifier extension, which is the last 18 bits of the identifier in case of a extended format frame. As a matter of fact, the standard

and extended format are distinguished by the IDE bit. If this bit is 0, the frame is of base format, while it is 1 in case of a extended format.

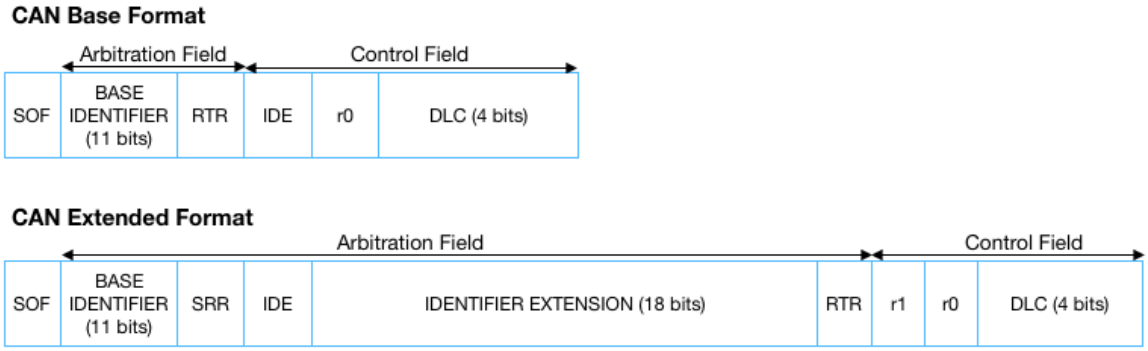

Figure 2 – Fields of a data or remote CAN frame of standard or extended format

Another bit of interest is the RTR bit, which distinguishes between a remote and data frame. Being 1 in the former case and 0 in the latter. Note that if there is a dispute between two nodes, one trying to transmit a data frame and the other a remote frame of same identifier, the one transmitting a data frame wins the arbitration.

The Data Length Code (DLC) field represents the four bits of the data length code. It determines the number of bytes transmitted by the frame in the data field. If the DLC is, for example, "0000", the data field has 0 bytes. If it is "0101", the data field has 5 bytes. If it is "1XXX", the data field has 8 bytes. "X" represents a don't care bit, i.e., this bit can be either 0 or 1. The data field is not represented in the Figure 2, but follows the DLC field. It contains the actual data being transmitted, and can have up to 8 bytes.

Also not represented in Figure 2, the remaining fields in the frame are the CRC, the Ack and the End of Frame fields. The CRC field serves as a redundancy check while the acknowledgment field serves for the node to acknowledge the receipt of a frame. For instance, if a frame is transmitted and there is no other node in the bus to receive it, an error occurs and the transmitting node keeps trying the retransmit the frame. Finally, the End of Frame field contains seven recessive bits that delimit the end of the frame. After it, if no error or overload condition is detected, the bus goes to idle and is ready for the transmission of another data or remote frame.

2.1.1 CAN FD

So far, it has been said CAN networks can transfer up to 8 data bytes of payload with a bit rate of 1 Mbps. This is more than enough for control signals between ECUs and from/to sensors and actuators. However, it may not be sufficient when it comes to other applications such as lane departure systems, parking aids or blind-spot detection. Thus, motivated by applications requiring faster communication or increased data payload (PRADEEP, 2013), the CAN FD protocol was proposed and developed.

This technology is based on CAN in a way nodes of both protocols interoperate and can share the same network, as long as there is some compatibility between the controllers and transceivers. The protocol efficiency is increased by a increased data payload. The data field is able to transfer up to 64 bytes of data, instead of only 8 as in CAN networks. The DLC field works in the same way, encoding the length of the data field. But, now, only the "1000" code indicates a 8 bytes data field. Data fields with more than 8 bytes are encoded as exhibited in table 3.

Table 3 – DLC encoding for CAN FD networks

DLC 1st bit	DLC 2nd bit	DLC 3th bit	DLC 4th bit	Number of Data Bytes
1	0	0	1	12
1	0	1	0	16
1	0	1	1	20
1	1	0	0	24
1	1	0	1	32
1	1	1	0	48
1	1	1	1	64

In addition to a higher data per frame rate, a faster communication is achieved by switching between two bit rates. While a CAN node has only one bit rate, a CAN FD node can switch from a nominal bit rate to a data bit rate during portion of the frame. By doing so, a part of the frame, that is called data phase and includes the data field, can have a higher bit rate and then transmit bits faster.

Observe in Figure 3 a comparison between the structures of a CAN and CAN FD frames of base and extended format. Also, notice the bit EDL in CAN FD frames corresponds to a reserved bit (r0 or r1) in a CAN frame. This bit serves for distinguishing between CAN and CAN FD frames. If it is 1, it is the EDL bit and the frame is a CAN FD frame. If it is 0, it is a reserved bit and the frame is a CAN frame.

In the CAN FD frames diagrams, is possible to observe the start of the data phase in the BRS bit. This bit indicates whether the bit rate should be switched or not. If it is one, the bit rate is switched from the nominal bit rate, in the arbitration phase, to the data bit rate, in the data phase. If it is zero, there is no switch and the whole frame operates with the nominal bit rate (ROBERT BOSCH GMBH, 2012). In case of a switch, the bit rates are switched back at the CRC field.

2.1.2 Bus synchronization

While CAN networks only have one bit rate, called nominal bit rate, CAN FD networks may switch between a nominal bit rate and a data bit rate during a portion of the frame. These bit rates are achieved through bit time configurations, one for each bit rate. Thus, the nominal bit rate corresponds to a nominal bit time configuration and the data bit rate corresponds to a data bit time configuration.

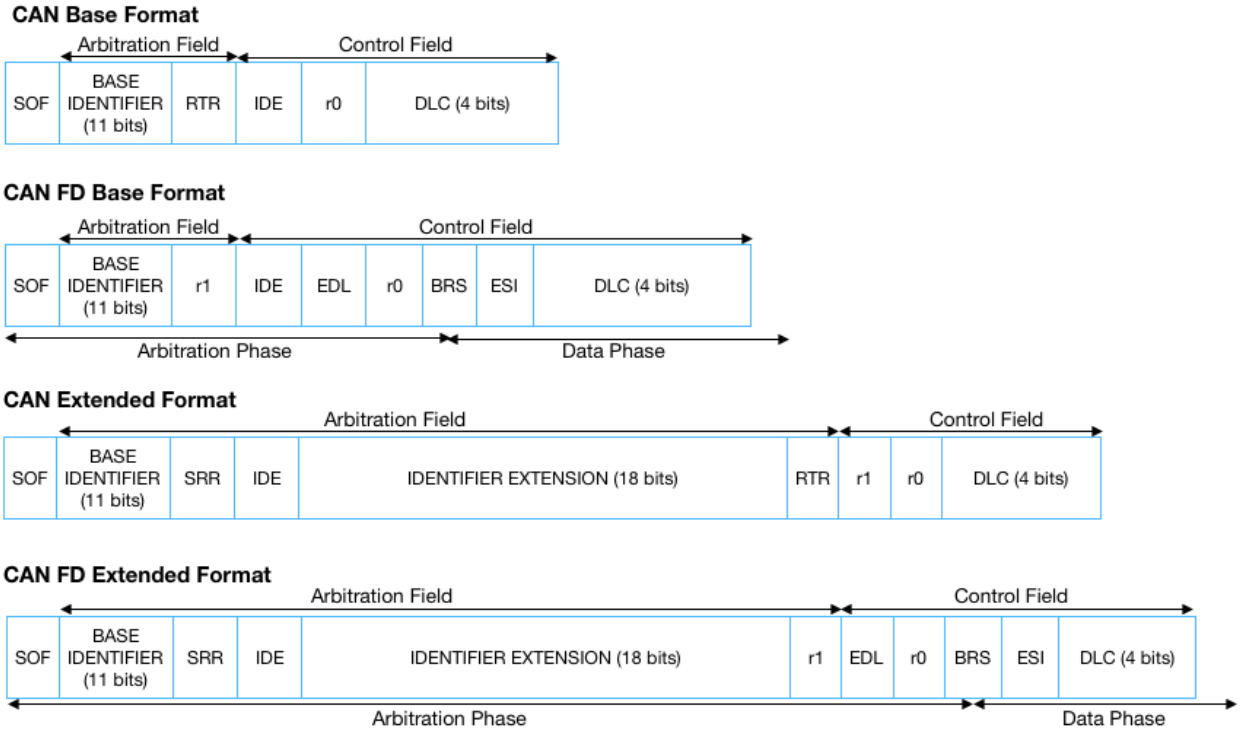

Figure 3 – CAN and CAN FD data or remote frames structure

Both bit times are divided into four time segments that are multiple of the time quantum, a fixed unit of time derived from the local oscillator (ROBERT BOSCH GMBH, 1991) and (ROBERT BOSCH GMBH, 2012). As described in Figure 4, the four segments of a bit time are:

- Synchronization segment (`Sync Seg`): Formed by a single time quantum, it is the portion of the bit time in which the various nodes in the bus must synchronize. The leading edge of a bit is expected to be positioned within this segment.
- Propagation segment (`Prop Seg`): Used to compensate for physical delay times within the network, it is twice the sum of the signal propagation time on the bus line, the input comparator delay and the output driver delay.
- Phase buffer segment 1 (`Phase Seg 1`) and Phase buffer segment 2 (`Phase Seg 2`): Used to compensate for edge phase errors by being lengthened or shortened during resynchronizations.

In addition, between the phase buffer segments is the sample point, the location in time at which the bus level should be read and interpreted as the value of that respective bit (ROBERT BOSCH GMBH, 2012).

In order to correctly read the bus level, the distance between sample points and edges from recessive (logic 1) to dominant (logic 0) bits must be controlled. The edges are

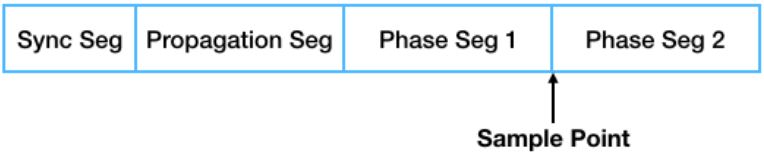

Figure 4 – Nominal CAN bit time

supposed to take place within Sync Seg, and if they lie in any of the other segments, there is a phase error that must be compensated by performing a synchronization (HARTWICH; BASSEMIR, 1999).

There are two types of synchronization: hard synchronization and resynchronization. The former is done at the start of frame, and the bit time is restarted with the end of Sync Seg, regardless of the phase error. The latter happens at every other recessive to dominant edge inside a frame and leads to a compression or an expansion of the bit time such that the position of the sample point is altered with regard to the edge (HARTWICH; BASSEMIR, 1999). If the edge lies before the sample point, the phase error is said to be positive and Phase Seg 1 is lengthened to compensate for that phase error by an amount up to a parameter called Synchronization Jump Width (SJW). On the other hand, if the edge lies after the sample point, the phase error is said to be negative and Phase Seg 2 is shortened also to compensate for the phase error by an amount up to SJW.

Once the synchronization to the bus is accomplished through hard synchronization and resynchronization, the bus is decoded, i.e., the bit value at the sample point is read and interpreted according to the CAN frame specification, as illustrated in 3. Correctly interpreting the bits — especially the RTR (Remote Transmission Request), IDE (Identifier Extension), EDL (Extended Data Length) and BRS (Bit Rate Switch) bits — allows to distinguish the frame from being a CAN or a CAN FD frame as well as from being of normal or extended format, in case of a data or a remote frame. Note that it is also necessary to identify the occurrence of error and overload frames.

2.1.3 Bit Stuffing

A last topic important to be discussed regarding CAN and CAN FD networks is the bit stream coding with stuff bits. Since synchronization only occurs in recessive to dominant edges, if there are too many consecutive dominant or recessive bits in the bus, there can be a long time with no synchronization being performed in the bus. To avoid this and guarantee a maximum distance between edges available for synchronization, the frame is coded with the method of bit-stuffing (ROBERT BOSCH GMBH, 2012).

This method basically inserts a dominant bit after every five consecutive recessive bits and a recessive bit after every five consecutive dominant bit. The bit inserted is called a stuff bit and should be disregarded while decoding the bus. Its only purpose is to create an edge so a synchronization can occur. This process occurs for all frame segments until

the CRC field for CAN networks. For CAN FD networks, it occurs until before the CRC field and stuff bits are inserted in a particular fixed manner in the CRC field.

2.1.4 Protocol Testing

As seen in the previous subsections, CAN networks are largely used in the automotive domain. They are mostly responsible for control data and can be considered the main intra-vehicular network technology, at least for now and when it comes to an average car. On the other hand, CAN FD networks are becoming the solution to go when there is a need for higher payloads and bit rates (but limited to 10Mbit/s), specially because its interoperability with CAN.

That said, the reliability and stability of CAN-based ECUs is of major concern (LUO et al., 2009). It is then necessary to deeply test ECUs and its interaction with each other as well as with sensors and actuators connected to the network bus. For this purpose, there are a few testing tools available and used by the industry. Most of them are based on the same procedure: setting up a trigger condition and then injecting errors upon this condition (MOSTAFA; SHALAN; HAMMAD, 2006). This approach enables testers to expose all ECUs connected to the bus to an intentionally created error condition or even to adulterate the contents of some frame payload. Moreover, it can be also used for vulnerability testing, since allows testers to inject errors to the bus and analyze how the ECUs would behave upon these error conditions.

2.2 OTHER USED NETWORKS

Before moving on to the next chapter, it is necessary to discuss, even if more briefly, other traditional intra-vehicular networks. Each of them has its own particularities and characteristics, with great advantages for some use cases, but also limitations. The next subsection addresses LIN, MOST and FlexRay technologies.

2.2.1 LIN

There are many vehicle applications that are simple and have very low requirements, such as power windows, central locks, or light sensors. These kind of application basically requires a simple sensor-actuator communication. Therefore, a solution cheaper than CAN would be preferable.

A joint force of companies then created the LIN consortium in order to standardize a solution for a cost efficient network capable of handling simple applications that would be over-performed by CAN (MATHEUS; KÖNIGSEDER, 2015). The solution, LIN, was a single wire system with data rate limited to 19.2Kbps.

LIN was designed such that up to 16 ECUs could share the media provided by the bus. In spite of that, there is no conflict on the bus because it is based on a master-slave access

method. A slave can only transmit after being told to by the master. Additionally, the communication scheduling is predetermined. LIN Description Files contains the scheduling tables and configures the entire LIN network (GMBH.; REIF; DIETSCHE, 2014).

LIN networks fulfill well its tasks and are largely used in today's cars. Usually, several LIN buses are connected to more complex ECUs. This configuration along with the technology well adoption, makes one to believe that a vehicle architecture based on only one network technology is not likely to be seem, at least in the short term. Therefore, as long as LIN keeps being the most cost efficient solution for the low requirement communication it was designed for, it shall be present in vehicle architectures.

2.2.2 FlexRay

Intending to eliminate all mechanical fallback from the car and to have pure electric functions, the industry became interested in "X-by-wire" applications. Some examples of this kind of application are steering-by-wire and braking-by-wire, which aims a entirely electrical steering and braking systems, respectively. "X-by-wire" applications are usually related to safety-critical control systems, and then need a reliable and deterministic network technology. A braking command, for example, can not afford to be delayed by other packets in the network. This could jeopardize the safety of the driver and its passengers.

BWM and other companies formed then the FlexRay Consortium and developed the FlexRay technology, later turned into ISO standards (MATHEUS; KÖNIGSEDER, 2015). FlexRay only defines the first two OSI layers (PHY and DLL) and is able to reach 20Mbit-s/s data rates over a twisted pair cable (GMBH.; REIF; DIETSCHE, 2014). Its communication is based on time slots and cycles. Each cycle consists of a static segment and a network idle time, but can additionally comprise a dynamic time segment. Access in the static segment is granted by TDM, in a way each time slot is assigned to a node. The dynamic segments use a "mini-slot" method with a preset order of frame identifiers combined with counter for multi-user access (MATHEUS; KÖNIGSEDER, 2015). These methods guarantee the needed known latency and determinism.

In spite of achieving its conception objectives, the use and acceptance of FlexRay did not developed as expected. Maybe because its actually complex technology combined with the slowly evolving "X-by-wire" safety-critical applications and the not suitability for being a in-car backbone network.

2.2.3 MOST

In the late 90s, the introduction of complex audio applications in cars become of great interest. Not only for playing music, but also for navigation systems. This kind of application, however, required a data transmission rate much higher than the one provided by CAN.

The solution was to introduce a new PHY technology, but the industry wanted more than that. The intention was to provide a system covering all network aspects from the PHY to the application layer. It was a challenging target to achieve. For this purpose, BMW, Daimler and OASIS (supplier, nowadays part of Microchip) founded the MOST Corporation to develop and establish MOST as a standard and a communication technology defining all seven layers of the ISO/OSI layering model (MATHEUS; KÖNIGSEDER, 2015). The aim was achieved and there are, today, three variants of MOST available to the industry. MOST25 and MOST150, first introduced with optical communication and reaching, respectively, 25Mbits/s and 150Mbits/s; and MOST50 with electrical UTP cabling reaching 50Mbits/s.

MOST is organized in a ring topology. One node acts as timing master to which the other nodes synchronize to. It provides asynchronous and synchronous channels. The former for transmitting application data and the latter for real time communication of audio and video data (GMBH.; REIF; DIETSCHE, 2014).

In spite of the effectiveness in handling audio and video data and the higher data rates achieved, MOST have significant issues for being largely adopted by the industry. First of all, Microchip charges for licensing the DLL/PHY technology, and does not provide a specification for interoperability and compliance. This creates a monopoly situation, which is not well seen or wanted by the automotive industry. In addition, by specifying all OSI seven layers and adopting a token ring topology, MOST became a complex technology with difficulties associated to adding or removing some nodes from the network. This scenario was actually one of the reasons the industry started researching about other network technologies with higher data rates to be applied in the automotive domain, process that has led to the Automotive Ethernet.

3 PROPOSED ERROR INJECTION AND IDS TECHNIQUES FOR CAN NETWORKS

In Chapter 2 CAN and CAN FD networks were broadly discussed as well as the importance of their reliable and error free operation. A major step in this process is to predict the behavior of CAN/CAN FD nodes in error situations. For this, an error injection mechanism can be very advantageous, especially for validating safety requirements at the system level in real-world scenarios.

In this line of thought, in this dissertation, we propose a novel approach, called Bitsmash, for injecting errors in CAN and CAN FD networks by means of a single device. The tool is connected to a CAN/CAN FD bus and configured to smash or corrupt specific bits of specific frames. The proper configuration of the bits to be corrupted results in the occurrence of error conditions predicted in the CAN/CAN FD protocols, so it is possible to inject customizable errors on the bus. A hardware implementation of the technique is presented such that customizable errors are successfully generated and injected in real CAN/CAN FD buses.

Besides that, in the context of the car as a node connected to the world, vehicles are subject to cyber-attacks, what could potentially put in risk people's safety. A malicious packet could, for example, be injected into the car's CAN network and disable its airbag system. Therefore, it is essential to provide security mechanisms in intra-vehicular networks.

For this purpose, in this dissertation, we also propose an intrusion detection system for CAN networks using machine learning algorithms. This system was deployed and successfully tested by detecting injected frames in CAN networks. Furthermore, both the Bitsmash and CAN IDS system can be combined into a single Intrusion Prevention System in a way that intrusion frames are detected by the CAN IDS system and corrupted by the injection of an error frame in real time.

In this chapter, both the Bitsmash and the CAN IDS systems are discussed by the means of their requirements, implementation and validation process. Moreover, an analysis is made on how to combine them into the single IPS system.

3.1 BITSMASH ERROR INJECTION TECHNIQUE

3.1.1 Requirements and Operation

The purpose of the Bitsmash technique is to assist in the validation of electronic components and safety requirements at the system level in real-world scenarios. In essence, it makes possible to inject configurable errors in CAN/CAN FD networks, so the system behavior under these conditions can be observed. Only one device is needed for injecting

errors in any node connected to the bus. The control of which node is affected by the error relies on the message identifiers expected by the node and configured for the errors.

The proposed technique offers full flexibility and control on corrupting any bit, from the first Data Length Code (DLC) bit (in CAN frames) or the BRS bit (in CAN FD frames), in any frame in the bus. Therefore, with the proper setup, it is possible to generate any of the following errors described in the CAN/CAN FD protocols: Bit Error, Form Error, CRC Error and Stuff Error (MATSUMOTO et al., 2012). Bit errors happen when a node sends a bit, but reads its opposite level when monitoring it. For example, it sends a recessive bit, which gets corrupted and a dominant level is read instead. Form errors happen when a bit responsible for the format of a frame is corrupted. For example, in case the end of frame (EOF) bit is dominant instead of recessive. CRC errors take place when the cyclic redundancy check (CRC) in the CRC field of a received frame does not match to the one calculated by each node for that frame. This happens, for example, when bits in this field are corrupted. Finally, stuff errors are related to the protocols coding. After five consecutive bits with the same logic level, a stuff bit with the opposite level is demanded, so a resynchronization is guaranteed to occur. If there is no stuff bit, i.e. a sequence of six consecutive bits with the same level is observed due to a corruption of bits, a stuff error occurs.

The errors are generated by corrupting specific bits of specific frames. This process involves two main aspects: 1) the identification of the bits that should be adulterated; and 2) the actual corruption of these bits. The first aspect corresponds to the Bitsmash decision-making process, which is based on the parameters described in Table 4. The decision-making process occurs in the following way. After a command to start the Bitsmash process is given, a timer starts counting and the Bitsmash module starts reading and decoding the CAN/CAN FD bus incoming bits. It keeps checking if the incoming identifier corresponds to one of the identifiers present in `ID_List` in order to decide to corrupt or preserve the incoming frame. The decision of corrupting the frame occurs when the identifier is in `ID_List` and the Boolean parameter `Smash_all` is 0 or when the identifier is not in `ID_List` and the Boolean parameter `Smash_all` is 1.

Once a frame is decided to be corrupted, the module waits for `Bits_to_wait` bits starting at the first data length code (DLC) bit in case of a CAN frame or at the BRS bit in case of a CAN FD frame, and then smashes `Bits_to_drive` bits driving to a dominant bit 0. Note that this process must be responsive enough such that the bit smashing occurs from the beginning of the first bit that should be smashed until the end of the last bit to be smashed. This process continues until the number `CAN_frames` of frames are corrupted or the timer reaches the `Timeout` value.

The full control and flexibility provided by Bitsmash through its configurable parameters makes possible to elaborate customized test plans. One could be to corrupt the data being read from some sensor by adding the identifiers of the messages sent by this sensor

Table 4 – Bitsmash Parameters List

Bitsmash Parameters List	
Parameter	Definition
`ID_List`	List of arbitration IDs whose frames should be corrupted or preserved
`Smash_all`	Boolean variable that is 0 for corrupting the frames whose ID is in `ID_List`, and 1 for preserving them and corrupting all the other frames
`CAN_frames`	Maximum number of CAN frames to be corrupted
`Bits_to_drive`	Number of bits to be smashed
`Bits_to_wait`	Number of bits to wait before starting smashing bits
`Timeout`	Duration of time in which frames can be corrupted

in `ID_List` and configuring `Smash_all` as 0. Another one could be to preserve the frames transmitted by some node and to corrupt all received frames by adding the correspoding identifiers in `ID_List` and configuring `Smash_all` as 1.

In order to sucessfully accomplish the Bitsmash process, it is necessary to correctly synchronize and decode the bus. Besides that, since the bit is read at the sample point, the decision to smash or not a bit happens at this point. If the decision is for smashing the next bit, the time between the decision making and the actual smashing, driving a dominant bit to the bus, corresponds to a `Phase Seg 2`. As a result, the smash of the bit, i.e., the dominant bit driving, is required to occur in less time than the duration of `Phase Seg 2`, which can be really short considering the smash in the data phase of a 10 Mbps CAN FD network. If this requirement is not satisfied, the smashing will be late and will not be able to start at the right time.

3.1.2 Hardware Implementation and Experiment

In order to guarantee the timing requirements discussed in the last section, the design of Bitsmash must be implemented in hardware instead of in a processor. However, the use of a processor makes the system easier to be configured for different bit rates as well as for configuring the Bitsmash parameters from Table 4. The Microzed development board was chosen as a low-cost development board based on the Xilinx Zynq-7000 All Programmable System-on-a-chip (SoC). It was programmed using the Vivado Design Suite for the programmable logic side, and the Xilinx SDK for the processor system side. Figure 5 show the block diagram of the error injection technique implementation in the Zynq SoC. The actual error injection module is implemented in the FPGA side of the SoC, while the Bitsmash parameters are configured by the Setup Module in the processor side of the SoC. The SoC is then connected to the CAN bus through a CAN transceiver.

3.1.2.1 Error Injection Module Design

The Error Injection Module in Fig. 5 is detailed in Fig. 6. Observe that this module is composed of four modules: two Bit Timing Logic (BTL) blocks, a multiplexer and an Error Injection Top Level block.

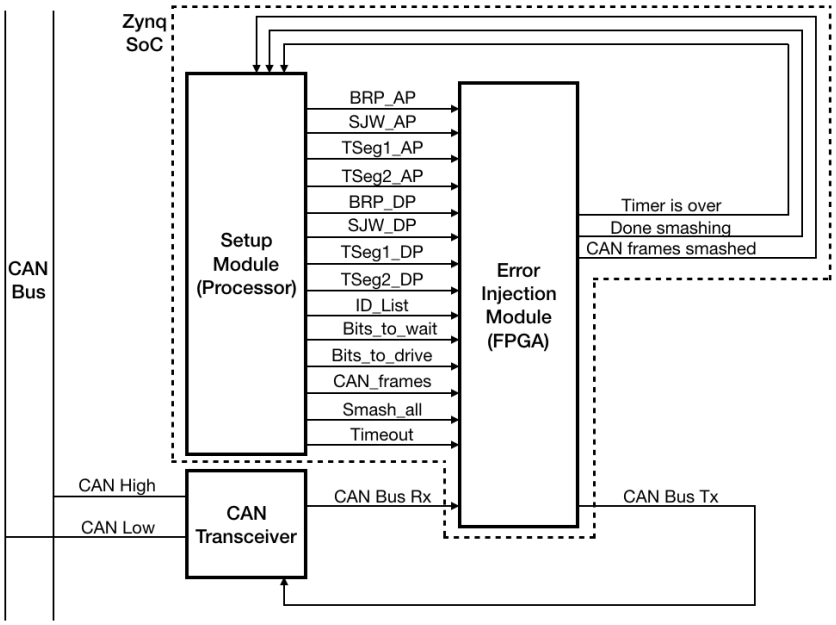

Figure 5 – Error Injection Technique SoC implementation and connection diagram

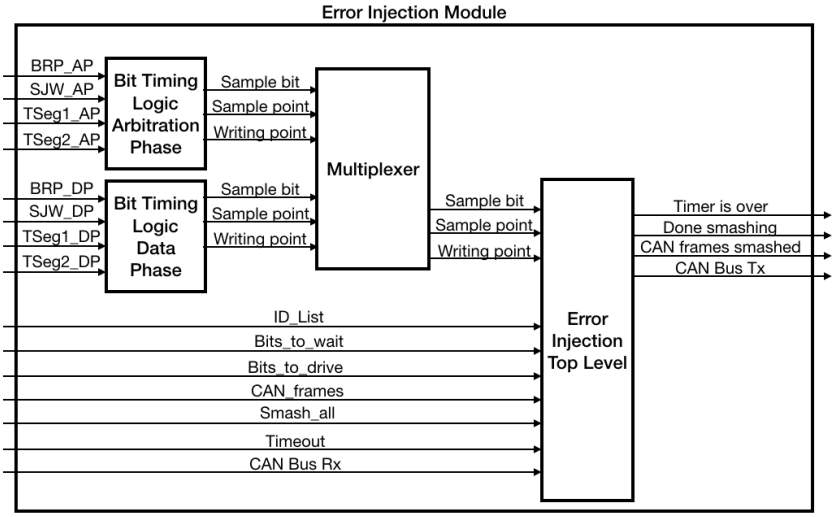

Figure 6 – Error Injection Module Block Diagram

The BTL modules are responsible for the whole synchronization process and are formed by other four blocks, as illustrated in Fig. 7: Edge Detector, Baudrate Logic, Synchronization State Machine and Sample Logic.

The Edge Detector, as its name indicates, detects edges, since the synchronization occurs at the edges. The Baudrate Logic receives a Bit Rate Prescaler (BRP), which is a value by which the clock should be divided, as an input parameter and divides the input clock according to it. The Synchronization State Machine is the core of the BTL module: It is a state machine that performs the synchronization and resynchronizations in response to the detected edges and outputs a sample point and a writing point — the latter identifies the beginning of the bit time. Finally, the Sample Logic receives the

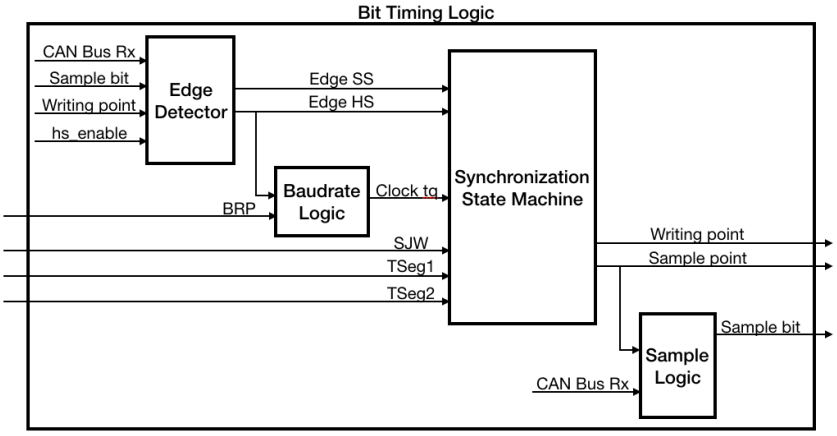

Figure 7 – Bit Timing Logic Module Block Diagram

sample point and outputs the correspondent bit value.

There are two BTL modules because, in case of a CAN FD frame and depending on its BRS bit, the frame can have different bit rates for the arbitration phase and the data phase. The multiplexer in the diagram is used to choose between the sample bit, the sample point and the writing point from one BTL module or the other, depending on if a bit rate switch is supposed to occur or not within a CAN FD frame.

The last module of the error injection module block diagram, the Error Injection Top Level block, is formed by other three blocks, as illustrated in Fig. 8: the Timeout Timer, the Top Frame, and the Corrupt Frame ID.

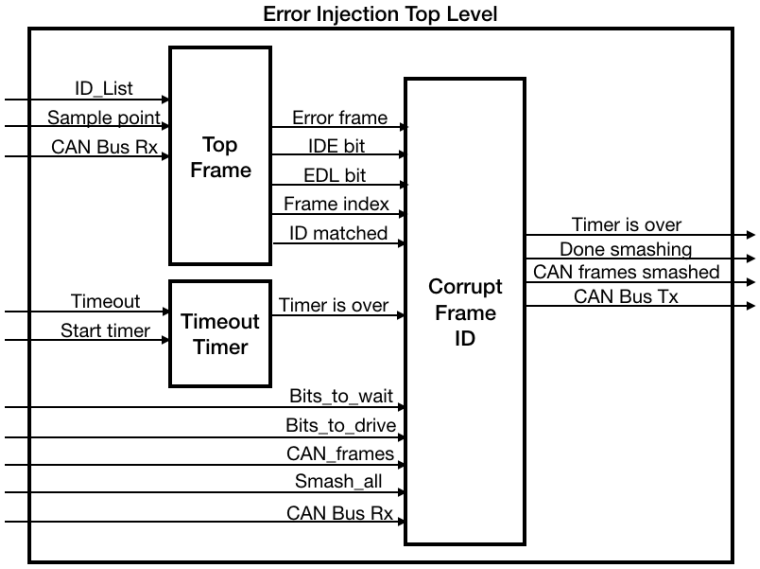

Figure 8 – Error Injection Top Level Block Diagram

While the Timeout Timer is simply a timer for measuring the time when the corruption of bits can occur, the Top Frame block is the one responsible for checking if the frame identifier received is in the `ID_list` or not. It decodes the bus, builds the frame and looks for an identifier match using the received sample points and bus values. This block then

outputs a signal informing if there was a match or not between the received identifier and the ones in `ID_list`.

The information of whether the identifier is in `ID_list` or not is then passed to the Corrupt Frame ID module along with the other Bitsmash parameters (i.e., `CAN_Frames`, `Smash_all`, `Bits_to_drive`, `Bits_to_wait` and `Timeout`). If the `Smash_all` parameter is 0 and the match has occurred or if it is 1 and there is no match, the module performs the smash of the desired bits, driving a dominant bit to the CAN bus through the CAN Bus TX output.

3.1.2.2 Experimental Setup

In order to validate the proposed design and implementation for Bitsmash, an experiment was conducted using real CAN/CAN FD nodes and bus. The experimental setup, illustrated in Fig. 9, constitutes of a CAN/CAN FD bus in which two neoVI FIRE 2 equipment, from Intrepid Control Systems, and a Microzed development board are connected. The FIRE 2 equipment work as network nodes capable of transmitting and receiving CAN and CAN FD frames. The development board comprises the Bitsmash design, responsible for corrupting frames according with real time configurations made to the parameters from Table 4. The Microzed, which includes the Zynq SoC, is connected to the CAN bus through a CAN transceiver.

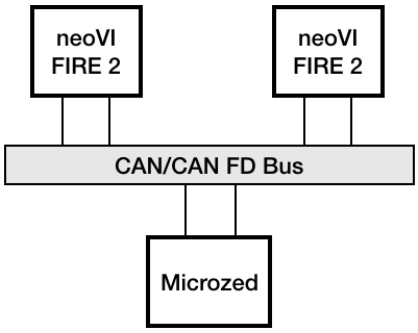

Figure 9 – Experimental Setup

The experimental setup was exposed to different types and formats of CAN and CAN FD frames, which were generated by the neoVI FIRE 2 equipments in association with Vehicle Spy software. Then, two categories of tests were performed: general smashing tests and error injection tests. The former corresponds to multiple tests in which Bitsmash was required to smash different bits in frames of different types and formats, and with different IDs. The objective was to verify whether the corruption of bits occurred as planned, regardless of any possible error condition that might have happened. The latter intends to validate the error injection mechanism. Bitsmash parameters were configured for customizing and injecting multiple error conditions from all four possible errors described before: Bit error, Form error, CRC error and Stuff error.

3.1.2.3 Results

The corruption of bits, in all tests performed for both the general smashing and error injection cases, was successfully achieved. The desired bits, and only them, were successfully smashed while also satisfying the required response time for different bit rates. In addition, when one of the CAN/CAN FD error conditions was injected, the neoVI products and Vehicle Spy software correctly accused that error. To observe if the smash was occurring successfully, i.e., if all the bits required to be smashed, and only them, were being smashed from the beginning to the end of the bit time, a PicoScope oscilloscope and the WaveBPS software, capable of decoding CAN frames, were used.

One of the performed tests aimed to corrupt three extended CAN frames with ID 124AB001. The Bitsmash module was configured according with the parameters values in Table 5. Then, one of the neoVI FIRE 2 connected to the bus was configured to send extended CAN frames with ID 124AB001. Note that, as long as the desired identifier is in *ID_List*, the other identifiers in the list do not matter for this particular test.

Table 5 – Bitsmash parameters values for a test setup

Parameter	**Values**
`ID_List`	{ 0x124AB001, 0x124ABEEF, 0x401, 0x555}
`Smash_all`	0
`CAN_frames`	3
`Bits_to_drive`	8
`Bits_to_wait`	5
`Timeout`	60000

The three successfully corrupted frames are presented in Fig. 10, followed by a non-corrupted frame, since the sending node keeps trying to transmit a valid frame. Figure 11 shows the smashed bits in detail. Note that, as expected, the Bitsmash module waits for five bits starting at the first DLC bit and then drives eight dominant bits. Because of the bit smash there were six following dominant bits causing a Stuff error that interrupted the data frame. Therefore, the parameters setup exhibited in Table 5 causes the injection of a Stuff error. One can realize that any parameters configuration driving six or more bits to the dominant level causes a Stuff error. In the same way, Bit errors, Form errors and CRC errors were also injected by setting proper configurations of parameters.

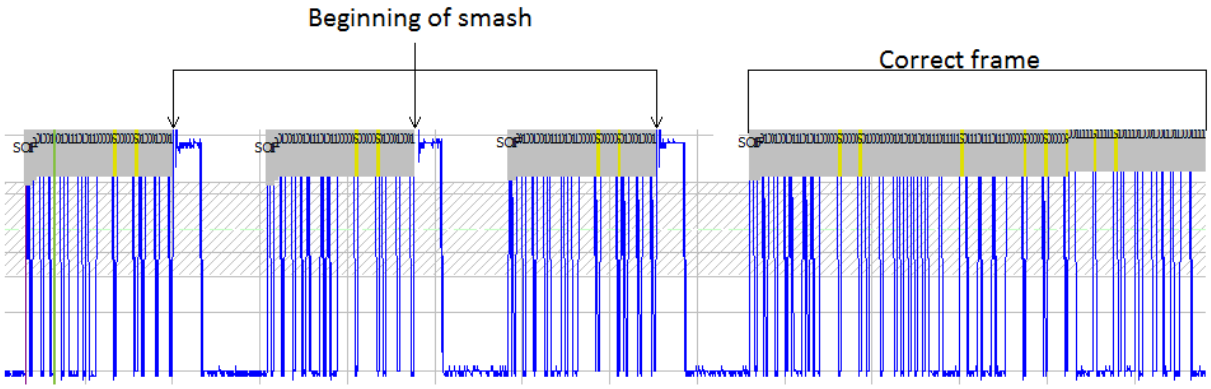

Figure 10 – Scope of three smashed frames followed by a valid frame.

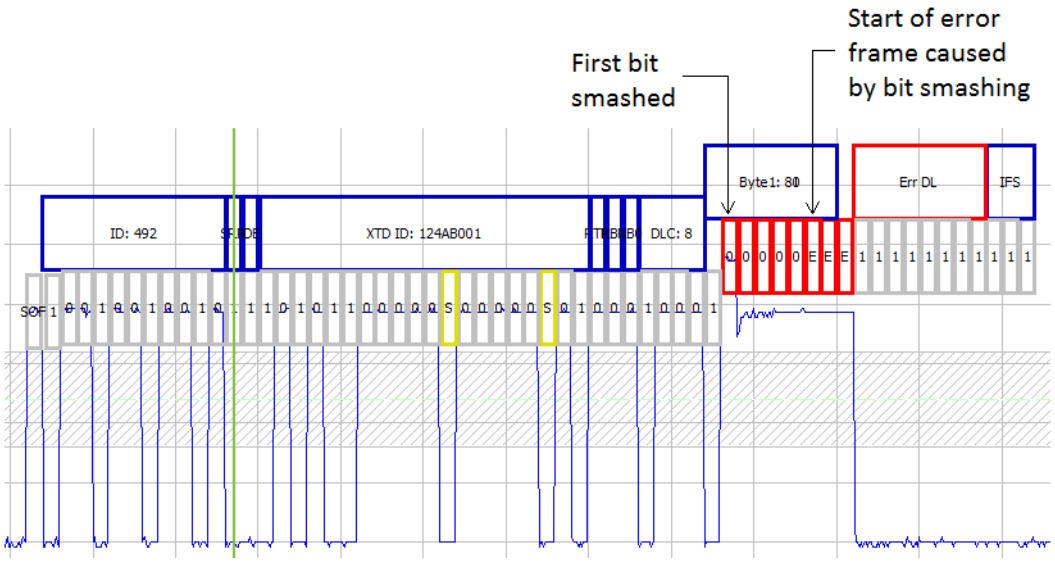

Figure 11 – Detailed scope of one of the smashed frames.

3.2 CAN IDS SYSTEM

3.2.1 Security Concerns in CAN Networks and Countermeasures

Even though the controller area network is the main intra-vehicular network (at least until today) for control data, until recently there was no concern for protecting this data as the car was considered an isolated device. However, when the car gained connectivity capabilities through technologies such as Bluetooth, 4G/5G, IEEE 802.11p and others, a paradigm change has occurred. The vehicle became a node in a network and could communicate with other vehicles, with the road infrastructure and with other devices such as mobile phones and computers.

In this scenario, the car became subject to cyber-attacks that could come from the network and put in danger the life of the driver and its passengers. A malicious CAN frame could, for example, be injected in the CAN bus of a vehicle and disable its airbag system, or even kill its engine. For the matter of fact, this has already been proved possible

by Charlie Miller and Chris Valasek, who remotely took control over a vehicle while it was being driven by a third person in a highway (MILLER; VALASEK, 2015).

A lot of research has been done and demonstrated the vulnerabilities and lack of security in cars as well as some possible countermeasures. The authors of (CHECKOWAY et al., 2011) discovered that a remote exploitation is feasible through a broad range of attack vectors, such as Bluetooth, CD players and cellular radio. In (WOO; JO; LEE, 2015) it is shown that long-range wireless attacks are possible using a malicious smart-phone application. In (LIU et al., 2017) attacking methodologies and possible countermeasures are summarized, while challenges and future directions in this field are discussed. Among the countermeasures presented, intrusion detections systems based on anomaly detection are proposed as a strategy for preventing frame injection attacks.

For this purpose, recently, a few IDS techniques have been proposed. An IDS technique based on an analysis of the time interval between frames with the same identifier is proposed in (SONG; KIM; KIM, 2016). Although great results are achieved, this approach is limited to periodic messages and can be compromised by possible delays in the transmission of frames. The work in (MARCHETTI; STABILI, 2017) also considers the frames IDs, but the sequence in which they appear in the bus, instead of the time interval between them. The authors of (KANG; KANG, 2016), on the other hand, use a deep neural network for classifying between normal and attack CAN packets in a CAN bus with the data bytes as features of the model. Despite the good results achieved with this strategy, deep neural networks demand devices with high computing power, what could be expensive and then not attractive for the automotive industry to embedded in a car.

3.2.2 The Proposed IDS Technique

It is well accepted that IDS techniques are a good strategy against some attacks, such as injection attacks, to the CAN bus. However, most of these techniques either have limitations or require considerable processing power. Therefore, in this dissertation, we propose an IDS technique that is able to detect injected frames with high accuracy while also being able to be deployed in a platform as cheap as a Raspberry Pi.

Frames that are regularly exchanged among nodes in a CAN bus are used to build a pattern of regular frames. Then, upon the arrival of a new frame, it is compared with the pattern established and classified as a regular frame, in case it fits in the pattern, or as an intrusion frame, if it differs from the pattern. In other words, the IDS technique intends to be a novelty detection system, whose task is to identify and classify data that differ from some pattern (PIMENTEL et al., 2014). For this purpose, two things are needed: strategies for building the regular frames pattern and a dataset with lots of regular and intrusion frames.

3.2.2.1 Machine Learning Algorithms for Novelty Detection

Upon the many possible strategies for building the pattern of regular frames, we decided to use two machine learning algorithms: the One Class Supported Vector Machine (ZHANG; XU; GONG, 2015) and the Isolation Forest (iForest) (LIU; TING; ZHOU, 2012). The choice of the OCSVM and iForest algorithms results from the fact that they have been indicated as potential candidates for the detection of intrusion frames in automotive CAN networks (WEBER et al., 2018).

Support Vector Machine (SVM) are machine learning algorithms that use separation hyperplanes to separate instances of data from different classes. A kernel function, which can be linear, polynomial or a radio-basis function (RBF), defines the mathematical form of the hyperplane, which is then optimized to minimize wrong classifications and maximize the distance between data from different classes. In the case of a novelty detection problem, the algorithm can be called a One-class Support Vector Machine (OCSVM) and trained only with regular frames, in a way that the optimum hyperplane aggregates all regular frames and keeps them separated from any other frame that is not regular (PIMENTEL et al., 2014).

On the other hand, the Isolation Forest algorithm builds an ensemble of isolation trees for a given dataset and takes as anomalies the instance that has short average path lengths on the trees. This approach is based on the fact that anomalies are, supposedly, the minority among the data and very different from regular frames. Thus, due to their susceptibility to isolation, anomalies are isolated closer to the root of the tree, whereas normal points are isolated at the deeper end of the tree (LIU; TING; ZHOU, 2008).

3.2.2.2 The Dataset and Cross-validation Approach

The dataset used was acquired from the Hacking and Countermeasure Research Lab. (KIM, 2018), which freely releases its dataset containing real-world data extracted from cars for academic purposes. This dataset contains a timestamp, the identifier, the DLC field and the eight bytes of payload for a few millions of frames, which are labeled as regular or intruders.

The frames in the dataset were divided into a training set, to train the model; a validation set, to validate the model and help with tunning of the model parameters; and a testing set, to represent the frames that would be incoming in the bus and to which the proposed IDS system would be applied to. To define these sets, first, 250,000 regular frames were randomly taken and split into 10 folds of 25,000 regular frames each, while the remaining regular frames were put on the testing set. Then, 25,000 intrusion frames were randomly taken into an intrusion fold, while the remaining intrusion frames were also put on the testing set.

The folds with regular frames were then used to form 10 pairs of training and validation

sets, in a way that for each pair a single and different fold was used as validation set along with the data in the intrusion fold, while the regular data from the other nine folds formed the training set. These ten pairs of training and validation sets were defined in order to apply cross validation to the models and achieve statistic significance for their accuracy results, such that it was possible to compare them to each other (DEMŠAR, 2006).

Besides that, despite the dataset has information regarding the frames timestamp, ID and DLC, it was decided to only use the frames payload as features due to the results of many classification tests performed. Moreover, instead of training only a model per algorithm, it was decided to also evaluate how much of data would be necessary to achieve good detection results in terms of accuracy. Thus, for each algorithm, three models were defined by varying the amount of data bytes used as features in the classification process. Three models were created for each algorithm by using the first six, the first seven or all the eight data bytes in the frames as features. Table 6 shows a description for the defined models.

Table 6 – IDS Trained Models

Models used for the proposed IDS	
Name	Description
`OCSVM 6 features`	Uses the OCSVM algorithm with the first six bytes of data as features
`OCSVM 7 features`	Uses the OCSVM algorithm with the first seven bytes of data as features
`OCSVM 8 features`	Uses the OCSVM algorithm with the eight bytes of data as features
`IF 6 features`	Uses the Isolation Forest algorithm with the first six bytes of data as features
`IF 7 features`	Uses the Isolation Forest algorithm with the first seven bytes of data as features
`IF 8 features`	Uses the Isolation Forest algorithm with the eight bytes of data as features

3.2.3 Experiment and Results

After preparing the training, validation and testing sets for the cross validation and also preparing the data contained in them to consider the aforementioned amount of features, the experiment itself constituted in the following steps:

1. Use the training and validation sets for tunning the hyper-parameters of the six models using the cross validation;
2. Use the cross validation methodology to obtain and train 10 variations for each of the six models;
3. Apply each trained model variation to classify the frames in the testing set as regular or intrusion frames.

As well as the data organization in folds and sets, the experiment steps were performed by programming with Python language and using appropriate libraries and packages that made the training and prediction tasks to be as simple as function calls.

Mean and standard deviation values were computed for the prediction results for each model variation and summarized in Table 7. Note the results obtained for the isolation forest models present higher accuracy rate mean and lower standard deviation than the results obtained for the OCSVM models. Also note that the values in Table 7 indicate that the more data bytes used as features the larger the accuracy rate mean and the smaller the accuracy rate standard deviation.

Table 7 – Models Accuracy Rate Mean and Standard Deviation

Model	Detection Rate	
	Mean	Standard Deviation
`OCSVM 8 features`	97.0633%	1.7735%
`OCSVM 7 features`	96.5949%	2.1515%
`OCSVM 6 features`	95.6728%	3.1849%
`iForest 8 features`	99.7215%	0.0383%
`iForest 7 features`	99.5916%	0.0530%
`iForest 6 features`	99.3140%	0.0885%

Furthermore, a confusion matrix was plotted for each of the six models, as described in Figure 12. The results exhibited in them show that for all the six models better accuracy rates are achieved for classifying intrusion frames. In other words, a wrong classification is more likely to occur for regular frames than for intrusion frames.

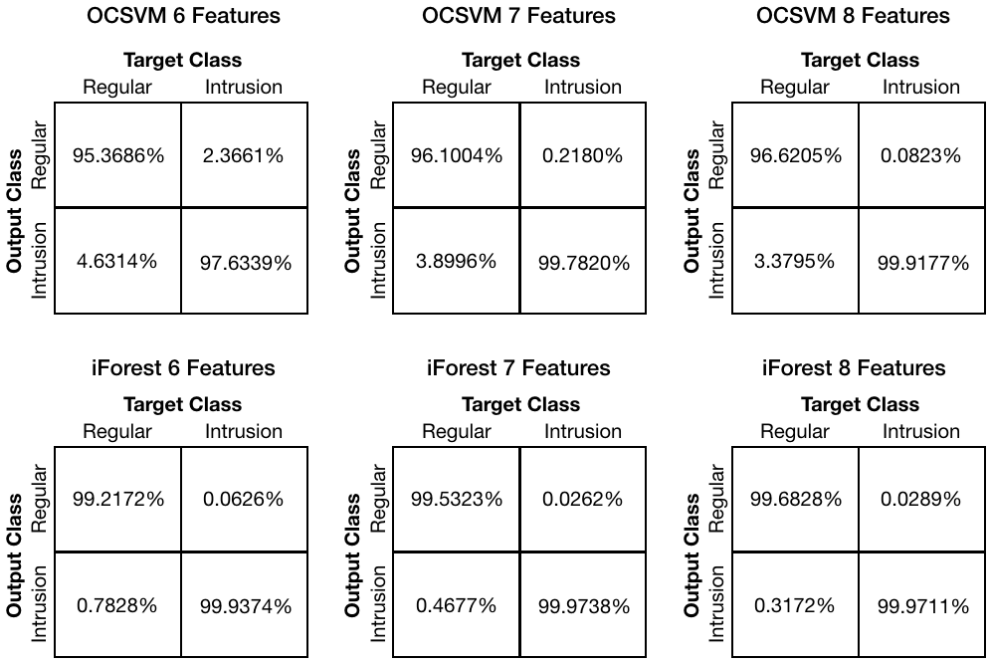

Figure 12 – Models Confusion Matrices

Despite the results from Table 7 may be meaningful, they only represent a punctual estimative. The calculation and plot of confidence intervals are a better approach to provide statistic significance to the obtained results. For this reason, first, the D'Agostino and Person's hypothesis test is used to verify whether the accuracy rates, obtained from the models variations, can be approximated by a normal distribution. The null hypothesis,

that the sample data come from a normal distribution, could not be rejected for any of the six models. Thus, the sample data may be approximated by a normal distribution for all the models (D'AGOSTINO; PEARSON, 1973). However, since only 10 folds are used, the number of samples is small and a approximation by the t-Student distribution is made instead.

Figures 13 and 14 show the confidence interval with 99% of confidence level for the six models. Note that the confidence intervals for the OCSVM models overlap, while the confidence intervals for the Isolation Forest models are disjoint among themselves and also from the OCSVM models. When confidence intervals overlap, it is not possible to consider a model to be superior to the others, as its results can be anywhere within the interval with the probability of the confidence level. Thus, it is not possible to affirm that any of the OCSVM models is better than the others. On the other hand, since the confidence interval for the IF 8 features model is disjoint from the others and is the one with higher accuracy rates, being the one more to right, it is possible to affirm it is the model with best results in terms of accuracy rate. In the same way, the IF 7 features model and the IF 6 features model are the second and third models with best accuracy rates, respectively, from the ones in the experiment.

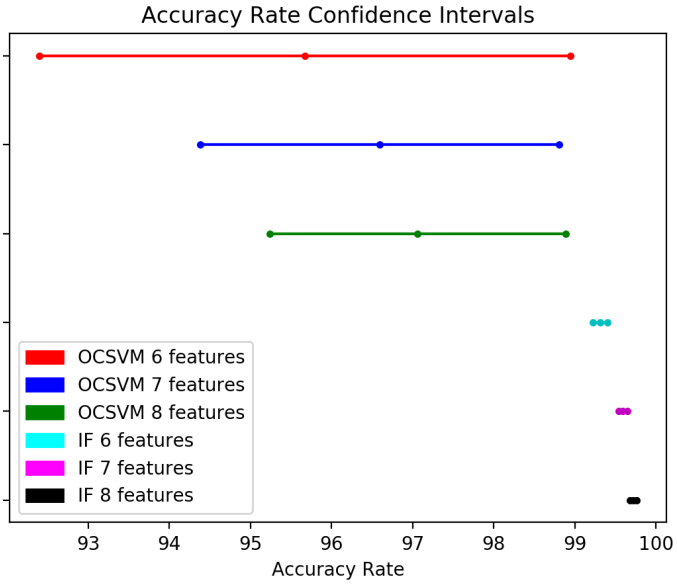

Figure 13 – Confidence interval with 99% of confidence level

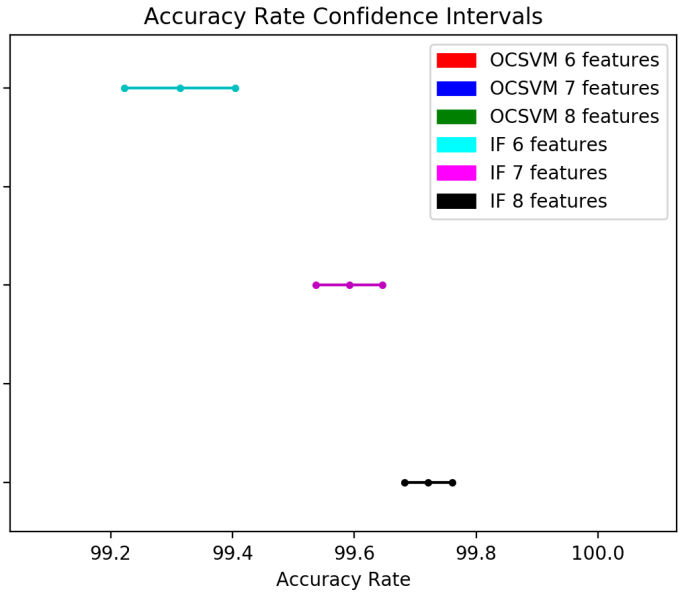

Figure 14 – Confidence interval with 99% of confidence level

3.3 CAN IPS SYSTEM: CORRUPTING INJECTED FRAMES

So far, we have proposed a novel Error Injection technique for CAN/CAN FD networks and a CAN IDS system. The former allows the injection of customizable errors in CAN/-CAN FD networks by precisely corrupting bits with great responsiveness. The latter is based on machine learning algorithms and detects whether a frame is a regular or an intrusion frame achieving high accuracy rates, even when only using the first six bytes of the payload as features.

In addition to the individual functionalities of these proposed systems, both of them could be used as the compounding modules for an Intrusion Prevention System (IPS). The IDS module would detect whether the incoming frames are intrusion frames and, if this is the case, notify the Error Injection module, which would then corrupt the intruder frame by smashing the next six bits within it, such that all ECUs would discard it. Observe in Figure 15 a block diagram for this possible IPS system. Note this is just a possible representation for such a system, for example, the same CAN bus decoder module from the Error Injection technique could also be used for decoding the IDS module incoming frame. In this way, this decoder module would be extracted from the Error Injection Module and its output would then be used for both the Error Injection Technique and IDS module.

This design, however apparently simple, has rigid timing requirements. In order to corrupt some intrusion frame, such that the ECUs would discard it, it is necessary to classify the frame as regular or intrusion fast enough for being able to corrupt it, if it is the case, before the frame is over. A late detection would not allow the corruption of the intrusion frame, such that the ECUs would accept it and suffer the consequences of the

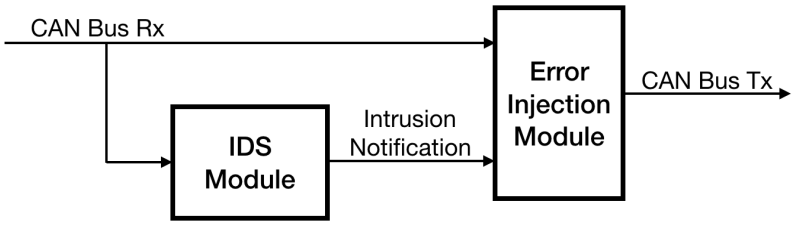

Figure 15 – Block Diagram for a possible IPS system

attack.

The frame detection can only start after the data that is used as features in the classification is acquired. The IDS models presented in the last section use the first six, the first seven or all the eight bytes in the frame payload as features. In case of only six data bytes are used, the total available time for detecting and corrupting a possible intrusion frame is the time that corresponds to the bits that comes after the six data bytes used until the *end of frame* bit, which is the last bit within a CAN frame. According to the CAN protocol frame format, this corresponds to 40 bits. In the same way, the total available time for detecting and corrupting frames when seven bytes are used is the time that corresponds to the bits that comes after these seven bytes until the *end of frame* bit. This corresponds to 32 bits, eight bits (or one byte) less than when using six data bytes as features. Finally, the total available time for detecting and corrupting frames when using eight bytes corresponds to the time of 24 bits, 16 bits (or two bytes) less than when using six data bytes as features.

The corruption of a intrusion frame, on the other hand, is accomplished by the smash of six consecutive bits, that in turn generates a bitstuff error. Then, after detecting the frame, there should still be enough time for smashing these six bits within the frame before the *end of frame* bit is reached. Thus, the Maximum Detection Time (MDT) is given by equation 3.1,

$$MDT = AT - CT \tag{3.1}$$

in which AT is the available time for detecting and corrupting the frame, CT is the time needed for corrupting the frame and MDT is the maximum time available for detecting the frame as regular or intrusion.

Observe in Table 8 the MDT values obtained from equation 3.1 in terms of amount of bits and μs for a 500Kbps CAN network, in case of six, seven or eight data bytes are used as features.

Table 8 – Maximum Detection Times Available

Number of Features Used	**AT** (in bits)	**DT** (in bits)	**MDT** (in bits)	**MDT for 500Kbps CAN networks** (in μs)
8 data bytes	24 bits	6 bits	18 bits	36μs
7 data bytes	32 bits	6 bits	26 bits	52μs
6 data bytes	40 bits	6 bits	34 bits	68μs

Considering this time analysis, the suggested IPS system requires the satisfaction of the MDT times from Table 8, which are, in fact, in the order of just tens of micro-seconds. Thus, it is necessary to measure the time taken by the IDS system to detect incoming frames. However, with the purpose of obtaining quick results and validate the suitability and accuracy rate of the proposed IDS system, this system was implemented by programming with Python language and using a traditional operating system based on Linux. This approach does not offer the required precision on taking measurements in the order of less than mili-seconds. Then, it was only possible to measure the total time needed to detect all the frames in the testing set defined for the experiment of the last section, and then calculate a mean detection time value for each frame. This mean value satisfies the established MDT limits even when using a cheap hardware platform with limited processing power, such as the Raspberry Pi Model 3, but is not enough for ensuring that the MDT would be individually satisfied for each frame. Furthermore, the use of a traditional operating system is not ideal for achieving determinism or satisfying real time deadlines.

Therefore, in order to appropriately validate the proposed IPS system, it is necessary to actually measure the individual detection time for each frame in the IDS testing set and to deterministically guarantee they are lower the established limits. So, the IDS system, that makes use of Python and a traditional operating system, needs to be re-designed using a lower level programming language, such as C, and also a Real Time Operating System (RTOS). Further research and implementation efforts are required and this is left as a future work, beyond the scope of this dissertation.

4 AUTOMOTIVE ETHERNET AND AVB TIME SYNCHRONIZATION

As seen in Chapter 2, intra-vehicle networks are not based (at least until now) on only one networking technology, but on many. Each technology has its particular advantage and purpose, and should coexist with other ones in an automobile.

The complexity brought by this multi-protocol architecture has significantly increased the amount of software to be flashed into ECUs. Besides that, new features for the automotive domain, such as infotainment and Advanced Driver Assistance Systems (ADAS) systems, will require bigger and more complex software. Nonetheless, many high end cars today have more than 100 million lines of code (CHARETTE, 2009).

This huge increase in the software size brings a problem: the time needed for programming or updating an ECU firmware. BWM used to use a 500 Kbps high speed CAN network for this (MATHEUS; KÖNIGSEDER, 2015). However, at some point, because of the size of the data being transfered, an activity that once took thirty minutes were taking hours to be accomplished. Changes were required.

This chapter first covers the path taken by engineers and researchers who came up with the automotive Ethernet, exploring its three generations. Then, the lack of determinism in Ethernet is discussed as well as strategies to overcome it, with especial highlights for Audio Video Bridging (AVB) and time-sensitive networking (TSN). Finally, after introducing AVB and its compounding protocols, a special attention is given to the IEEE 802.1AS standard: The gPTP protocol defined in it, for synchronizing the nodes within an AVB network, is discussed and explained.

4.1 AUTOMOTIVE ETHERNET

4.1.1 Origins and Generations

The doubtless need for changes in the way data was uploaded to ECUs made BMW to start studying and searching for a new interface technology for ECUs. This technology should fulfill a few requirements in order to be used in ECU programming and also in diagnostic purposes. First, it should have a high data rate to speed up ECU flashing, without requiring additional processing resources. Also, it should be cost-efficient and have good network integration, so that it could be used by any dealer around the world (MATHEUS; KÖNIGSEDER, 2015).

The first two technologies considered were MOST and USB, which were soon discarded. MOST was discovered to not offer sufficient data rate, to be resource and cost demanding, and a completely new interface for external testers. USB, on the other hand, had enough bandwidth and acceptance among external testers. But, it did not offer network support or sufficient cable length and robustness.

A third evaluation was made on 100BASE-TX Ethernet, which proved itself to provide sufficient data rate and to be readily available in computers. Besides that, it was a network technology, through which a car could be treated as a network node. Immunity requirements for in-car communication were met using two pairs of Unshielded Twisted Pair (UTP) cables. On the other hand, electromagnetic compatibility (EMC) emissions were much higher than the acceptable levels, incurring in distortions on FM radio, for example. It was not a problem for the programming and diagnostic use cases, since in both situations the car would be stationary at a dealer or factory. Diagnostics and ECU programming were the first Ethernet use cases for the automotive industry, representing the first generation of Automotive Ethernet.

Despite 100BASE-TX Ethernet could be used satisfactorily for diagnostic and programming, it was not suitable for any other application for a running car. By that time, an application of interest for the industry was a Rear Seat Entertainment (RSE) system. This kind of system required much more bandwidth than CAN (or even CAN FD) could provide, while MOST was also complex and expensive. To use Ethernet for this purpose, it would be necessary to shield the cables for preventing EMC emissions. However, this was not an option due to the consequently extra cost and weight for the car. A solution was still required to be found.

BMW then approached Ethernet PHY vendors in a try of obtaining solutions from them. Broadcom then came up and proposed the so called BroadR-Reach Ethernet technology, solving the EMC emissions problem and promising to transmit Ethernet packets at 100Mbps at vehicle runtime over a Single Unshielded Twisted Pair (UTSP) (MATHEUS; KÖNIGSEDER, 2015). This technology was then standardized as the 100BASE-T1 Ethernet and is already being used in some cars on the market (STANDARD, 2016).

The 100BASE-T1 technology was the mark for the second generation of Automotive Ethernet, focused on ADAS and infotainment systems (HANK et al., 2013). Many vehicles already use today a lot of different camera-based sensors, requiring simultaneously large bandwidth and low latency, what was not possible before.

Furthermore, vehicles are expected to have a growing number of camera-based sensors, sometimes requiring uncompressed data transfers, as well as other short and long range radars. Some of them may even require more than 100Mbps of bandwidth. In order to provide more than 100Mbps of bandwidth, a joint force was formed to develop and standardize the 1000BASE-T1 Ethernet, providing 1000Mbps data rates and suitable for the automotive environment (IEEE..., 2016).

The third generation of automotive Ethernet third is related to the introduction of a new architecture, rather than some new application or Ethernet use case. It proposes a new intra-vehicular network architecture based on an Ethernet backbone. Instead of having multi-protocol networks connected or disjoint, there would be a common Ethernet backbone to which all sub-networks would be connected through switches.

Currently there is a working group for the development of a 10Mbps automotive Ethernet network, expected to be called 10BASE-T1 (MATHEUS; KÖNIGSEDER, 2017). This network would aim applications currently addressed by CAN, CAN FD and LIN networks. One of the ideas is the belief in a car not only based on an Ethernet backbone, but with only Ethernet networks. Further comparison among 10BASE-T1 Ethernet and legacy in-vehicle networks will be necessary in order to have a better view about future trends in automotive networks.

4.1.2 Bringing Determinism to Automotive Ethernet

The idea of an Ethernet backbone approach represents a paradigm change in intra-vehicular networks, since heterogeneous architectures would be replaced by a top-down, unified approach. This process has, however, a major barrier. Ethernet networks do not support, by its original conception, deterministic traffic and cannot guarantee a deterministic behavior. Furthermore, regardless the existence of an Ethernet backbone, real-time applications, such as safety-critical data, require extensions and strategies for bringing determinism to Ethernet.

During the development of this dissertation, we conducted an experimental evaluation of the cryptography overhead in automotive safety-critical communication using an Ethernet network. We considered a dedicated network for safety-critical traffic and used a real-time operating system (RTOS) to verify whether the required deadline for safety-critical communication was satisfied after the application of layer 2 cryptography schemes. This research resulted in the publication of the paper "Experimental Evaluation of Cryptography Overhead in Automotive Safety-Critical Communication", published in the IEEE Vehicular Technology Conference (VTC), 2018.

To overcome the lack of determinism in Ethernet, several approaches were proposed and adopted, such as IEEE 802.1Q, TTEthernet and AVB (TUOHY et al., 2015). IEEE 802.1Q is a simple technique used for assigning priorities to packets by adding an extra field to the Ethernet packet header. It works as a lightweight Quality of Service (QoS) scheme when used together with a traffic queuing algorithm, such as the weighted fair queuing algorithm. This strategy was used in the automotive field in a lot of studies, such as (LIM; WECKEMANN; HERRSCHER, 2011). It does a performance study on an in-car switched Ethernet network without prioritization based on the Internet Protocol (IP) and with the necessary automotive requirements and service constraints. Moreover, an 802.1Q-based system proposed in (LEE; PARK, 2013) has been shown to meet hard real-time delay constraints by limiting the maximum transmission unit (MTU).

On the other hand, TTEthernet is a proprietary technology designed to allow the coexistence of time-triggered, real-time synchronized communication with lower priority event-triggered messages over Ethernet, by the application of time-division multiplexing (KOPETZ et al., 2005). Three types of traffic are supported: Time-Triggered (TT), which

takes priority over the other types; Rate Constrained (RC), which is guaranteed to have a predetermined bandwidth level; and Best Effort (BE), which stands for the regular Ethernet procedures (TUOHY et al., 2015). A comparison between TTEthernet and FlexRay is done by (STEINBACH; KORF; SCHMIDT, 2010), which considers TTEthernet to be a viable solution for time-triggered communication in vehicles.

Last but not least, AVB was designed to provide time-synchronized streaming of audio and video using 802.3 Ethernet. As automotive Ethernet is a switched-based network, data must be synchronized no matter if image and audio might travel through different paths with different delays (MATHEUS; KÖNIGSEDER, 2015). Two traffic classes with different latency guarantees are supported by AVB. Class A traffic guarantees a maximum latency of 2ms and corresponds to 802.1Q priority level 3, while Class B guarantees a maximum latency of 50ms and corresponds to 802.1Q priority level 2 (ZINNER et al., 2011). As AVB is of a great deal to the industry, and this dissertation concerns one the its protocols, AVB is discussed in detail in the next section.

4.1.3 AVB/TSN

AVB extends Ethernet by adding QoS features to support multimedia streaming (SRIDHARAN, 2015). This is done by using a series of IEEE standards associated with the first generation AVB (AVBgen1), as listed below:

- IEEE 802.1AS - 2011;
- IEEE 802.1Qat - 2010;
- IEEE 802.1Qav - 2009;
- IEEE 802.1BA - 2009.

The IEEE 802.1AS standard is the generalized Precision Time Protocol (gPTP), based on the Precision Time Protocol (PTP) defined by IEEE 1588. This protocol is responsible for timing and synchronization for time-sensitive applications in bridged local area networks. It first determines whether the nodes connected to the system are capable of running the gPTP protocol or not. A gPTP domain is established with those nodes that are capable. Then, the best node for acting as a master is defined as the grandmaster, to which all the other nodes will synchronize to. A synchronization process is started so each node in the domain is synchronized to the grandmaster clock. This protocol is central to this dissertation and it will be further explored in the following section.

The IEEE 802.1Qat Stream Reservation Protocol (SRP) allows the allocation of resources, such as buffers and queues, for reserving streams with guaranteed maximum latencies. Resources can be reserved within bridges along the path between the talker (end station that is the source of a stream) and the listener (end station that is the destination of a stream), so the class' maximum latency is achieved (BELLO, 2014).

The idea of IEEE 802.1Qav is to introduce forwarding and queuing enhancements for time-sensitive streams. It handles priority allocation of streams by adding data to the Ethernet header in a very similar way to IEEE 802.1Q (TUOHY et al., 2015). Time-critical and non-time-critical traffic are separated into different traffic classes so they can be handled accordingly. The IEEE 802.1Qav also applies Credit Based Shaper (CBS) algorithms that perform traffic shaping at the output ports of bridges and end nodes (BELLO, 2014).

Finally, the IEEE 802.1BA profiles the other three standards for plug-and-play and video streams automotive Ethernet use cases (TUOHY et al., 2015). Besides these four standards, another important standard associated with AVB is IEEE 1722, the Audio Video Transport Protocol (AVTP). It enables interoperable streaming, being the layer 2 transport protocol for time sensitive applications in bridged local area networks. Equivalently, the IEEE 1733 is the layer 3 transport protocol for time sensitive applications in bridged local area networks. Figure 16 illustrates the AVB stack, in which the AVB protocols along with IEEE 1722 have a background color.

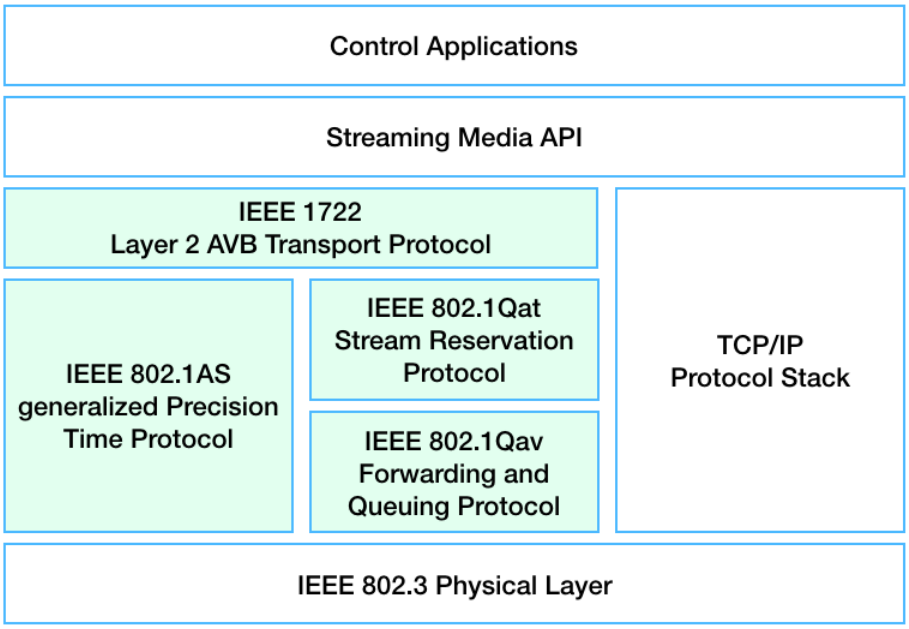

Figure 16 – AVB stack

Although the initial scope of AVB was to allow audio and video delivery in a time-synchronized manner, its potential use for other time-sensitive applications was soon realized (TUOHY et al., 2015). Thus, AVB was to be proved and improved for other time-sensitive applications in the automotive domain, such as safety-critical control systems, or even for other domains, such as aeronautical and industrial. With that in mind, the IEEE AVB task group changed its name to Time-Sensitive Networking (TSN) task group (AVNU..., 2018).

Already under the TSN name, a second group of IEEE standards was proposed and

can be referred to as AVB second generation (AVBgen2). By the time this dissertation was written, all of these standards were completed or nearly completed in IEEE and certification tests are under development. They were summarized by the AVnu Alliance for the 2017 IEEE Standards Association (IEEE-SA) Ethernet & IP @ Automotive Technology Day event, that happened in 2017 November in San Jose, California, USA. This summary is briefly reproduced below:

1. IEEE 802.1Qbv - 2015 (Done) - Time Aware Shaper
    a. Achieves the theoretical lowest possible latency in engineered networks.
2. IEEE 802.1Qbu - 2016 (Done) & IEEE 802.3br - 2016 (Done) - Preemption
    a. Reduces latency of time-sensitive streams in non-engineered networks.
3. IEEE 802.1Qch - 2017 (Done) - Cyclic Queuing & Forwarding
    a. Supports known latencies regardless of the network topology.
4. IEEE 802.1Qci - 2017 (Done) - Per Stream Filtering & Policing
    a. It is able to identify flows by other than layer 2 fields.
5. IEEE 802.1CB - 2017 (Done) - Frame Replication & Elimination
    a. Supports data redundancy "seamlessly" for the applications.
6. IEEE 802.1AS-Rev - 2017 (Draft 5.0) - Enhanced Generic Precise Timing Protocol
    a. Supports clock redundancy.
7. IEEE 802.1Qcc - 2017 (Draft 1.6) - Stream Reservation Protocol Enhancements
    a. Supports a "central controller" or pre-defined (flashed) "engineered configuration" or both.
    b. Supports a standardized interface to make reservations without needing to use stream SRP.
    c. Used to configure the features of the previous standards.

Beyond these standards, another one, which is in early stages of development in the IEEE TSN task group, constitutes the AVB third generation (AVBgen3):

1. 802.1Qcr - 2017 (Draft 0.1) - Asynchronous Traffic Shaping
    a. Supports deterministic latency without using network topology info.
    b. Supports zero congestion loss for asynchronous traffic.

Therefore, a long way has already been taken to enable time-synchronized audio and video delivery as well as to expand this initial purpose to other time-sensitive applications. On the other hand, there is still a lot to do, especially because the publication of a standard is only the first step to be taken. Actually implementing a protocol, once it has been standardized, is really a challenging job to be done. For example, to the best of our knowledge, even though the frame preemption protocol was published in 2016, until the beginning of 2018 there were only a few available implementations for the industry, and most of them are still in a prototype testing phase.

In this dissertation, an experimental implementation of the IEEE 802.1AS standard is presented. This standard is detailed in the next section, while the proposed implementation is addressed in the subsequent chapters.

4.2 AVB TIME SYNCHRONIZATION - IEEE 802.1AS

IEEE 802.1AS, which defines the generalized Precision Time Protocol, is the AVB standard responsible for the distribution of precise timing and synchronization within AVB networks. It is its duty to ensure the fulfillment of jitter, wander and time-synchronization requirements for time-sensitive applications (GARNER; OUELLETTE; TEENER, 2010).

The use of this protocol was initially focused on the automotive industry, from infotainment systems to more complex ADAS systems as well as on autonomous driving. However, it was soon expanded to other segments with time-sensitive requirements, such as avionics and industrial plants. Timing requirements fulfillment and synchronization are accomplished by the same means IEEE 1588 provides clock synchronization. For the matter of fact, the IEEE 802.1AS includes a very specific profile of the IEEE 1588, in which required and prohibited PTP features are specified.

4.2.1 Protocol Overview

gPTP requires all bridges and end stations to meet the requirements of IEEE 802.1AS, being able to transport synchronization (GARNER; RYU, 2011). They are then called time-aware bridges and time-aware end stations. The first, if not grandmaster, receives time information from the grandmaster, corrects it to compensate for delays and transmits the corrected time information to the next attached nodes. The latter, if not a grandmaster, receives time information from the grandmaster and synchronize itself to it.

The delays that should be compensated are composed of the residence and propagation times. The first corresponds to the time needed for a bridge to receive time information from the grandmaster and transmit it to the next attached time-aware systems. The residence time is then local to a bridge and not so difficult to be computed. On the other hand, the propagation delay corresponds to the time taken for the synchronized time in-

formation to transit between two time-aware systems. Its computation is more challenging and depends on the media through which the time-aware systems are connected to.

All time-aware systems must be connected in a point-to-point basis, not necessarily physically, but at least logically. That is, the gPTP information sent by a port is received by another gPTP port at the other end of the link. Whether the links are physically point-to-point or only logically connected is media-dependent (GARNER; RYU, 2011), i.e. depends on the media being used. The standard defines the following possible medias:

- IEEE 802.3 Ethernet using full-duplex point-to-point links;
- IEEE 802.3 Ethernet Passive Optical Network (EPON);
- IEEE 802.11 wireless;
- Generic coordinated shared networks (Coordinated Shared Networks (CSN), e.g., MoCA and G.hn).

Since some computations on the protocol are media dependent, the protocol architecture divides a time-aware system into a media dependent layer and a media independent layer. Each of them is responsible for specific tasks, which will be discussed during this chapter. Besides these two layers, there is also a higher layer for application interfaces. This layer can either provide the time information that will be transmitted to other nodes, or use time information received from other nodes. Figure 17 illustrates a description of the gPTP protocol layers.

Before moving on to the details of the protocol operation, it is enlightening to observe how time-aware systems connect and communicate to each other. Figure 18 exhibits a set of time-aware systems connected to each other by a local area network, forming a gPTP domain. Note that different medias can coexist in the same domain. The gPTP domain defines one of its time-aware systems as a grandmaster, which sends time information to all the other systems and to which they should all synchronize to. In case the connection with the grandmaster is lost, another node in the domain is defined as the grandmaster.

Figure 19, on the other hand, shows in more detail the communication between time-aware systems ports. Each time-aware system contains at least one port, which can have one out of four possible roles: Master, Slave, Disabled and Passive. The Master Port is any port of the time-aware system that is closer to the grandmaster than any other port of the gPTP communication path connected to the port. The Slave Port is the one port of the time-aware system that is closest to the grandmaster. The Disabled Port is any port of the time-aware system for which the port is not enabled or the time-aware system is not capable of running the gptp protocol. The Passive Port is any port of the time-aware system whose port role is not Master, Slave or Disabled (IEEE..., 2011). Time information is then passed from the grandmaster until the end stations through bridges in a way that master ports send the time information and slave ports connected to them receive it.

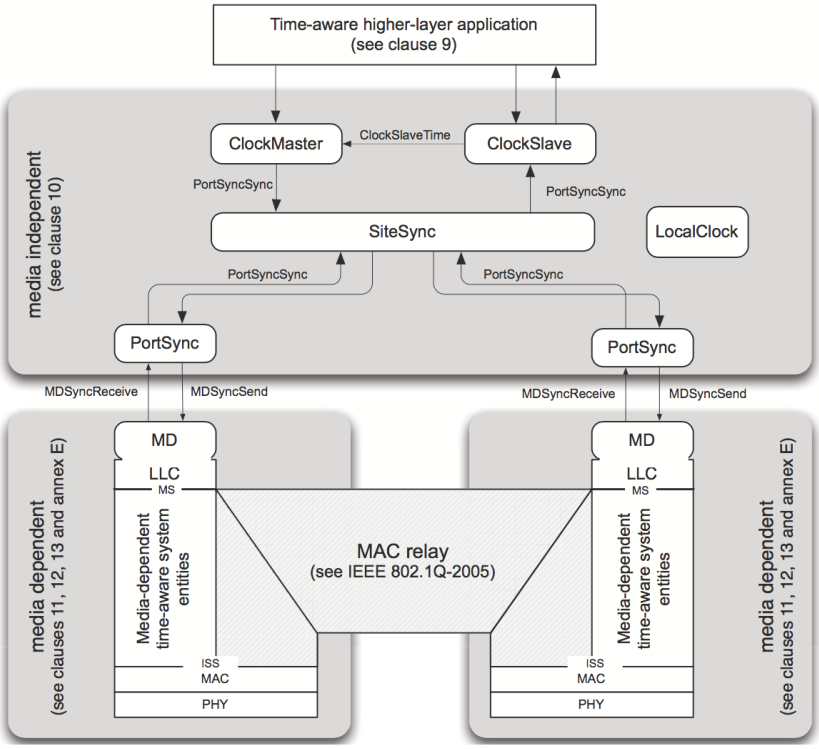

Figure 17 – Time-aware system model (IEEE..., 2011)

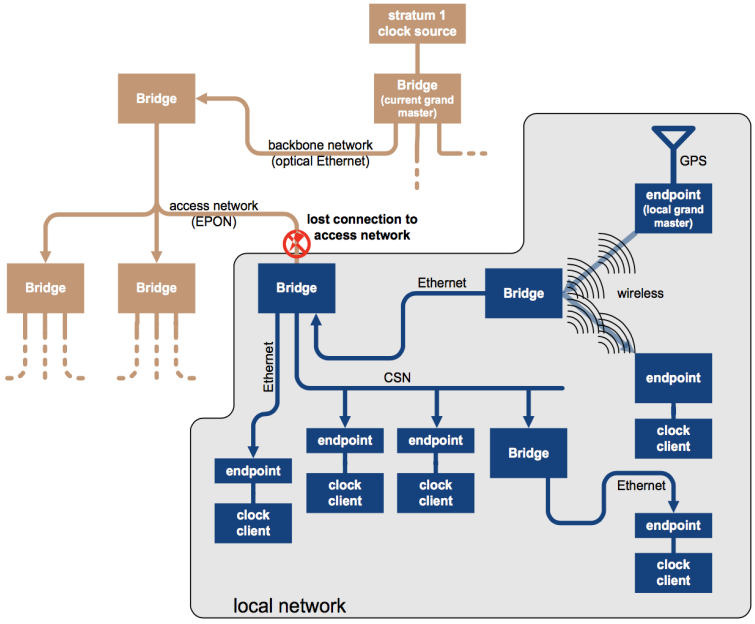

Figure 18 – gPTP domain (IEEE..., 2011)

4.2.2 Protocol Operation

As seen before, IEEE 802.1AS establishes a gPTP domain in which one time-aware system is defined as the grandmaster and all the others receive information from the grandmaster and synchronize to it (IEEE..., 2011). This process, however, is not so straightforward, but it can be divided into four steps for an easier understanding:

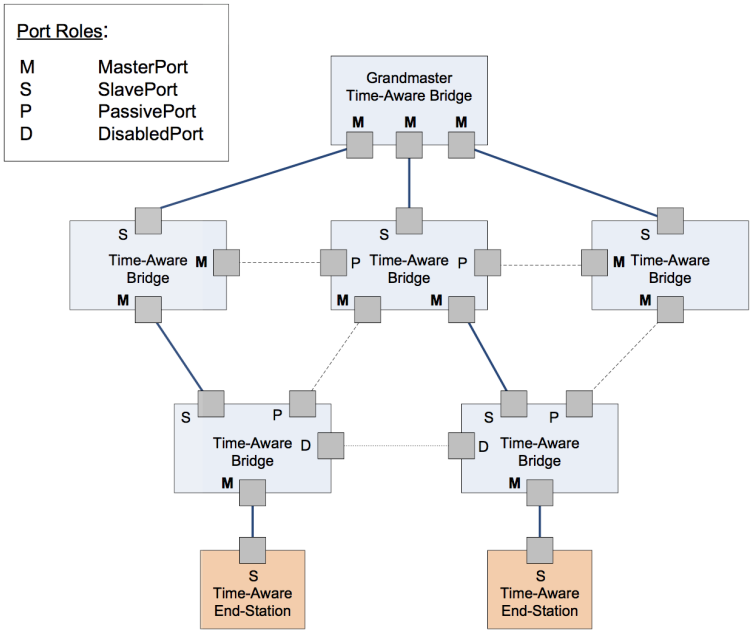

Figure 19 – Time-aware system port communication (LIM et al., 2011)

1. Determination of whether the peer is capable of supporting the gPTP protocol and interoperate with other nodes using it;
2. Determination of the path delay and the rate of the peer's clock;
3. Selection of the best master clock and establishing a synchronization spanning tree;
4. Transportation of time-synchronization information and synchronization of each node to the master time source.

4.2.2.1 Steps 1 and 2: the Peer Delay Mechanism

A global per-port boolean variable called *asCapable* is used to indicate whether the peer is capable of supporting the gPTP protocol. That is, this variable is set to TRUE if, and only if, the time-aware system to which this port belongs to and the time-aware system at the other end of the link attached to the port communicate to each other and can interoperate by using the IEEE 802.1AS protocol. The *asCapable* variable is then used to enable most of the state machines in the media independent layer, which performs the clock synchronization process. In this way, the clock synchronization process is performed only when the *asCapable* variable is set to TRUE and the peer supports the gPTP protocol and can interoperate with the peer at the other end of the link by using this protocol.

Even though *asCapable* is used in the media independent layer, its determination depends on the media and is then attributed to state machines in the media depended layer. For full-duplex Ethernet ports, it is a result of the Peer Delay Mechanism, as described in the IEEE 1588 protocol.

A *peer delay request* message is periodically sent by a node, called Node 1, to the node at the other end of the link, called Node 2. Then, Node 1 simply waits for a *peer delay response* and a *peer delay response follow up* message to be sent from Node 2. Node 1, the peer delay initiator, timestamps the instant t_1 at which the *peer delay request* message is sent. In the same way, Node 2, the peer delay responder, timestamps the instant t_2 at which the *peer delay request* message is received. It then prepares a *peer delay response* message carrying the time t_2 and sends it to Node 1. The time t_3 at which the response is sent is also timestamped and sent to Node 1 in a *peer delay response follow up* message. A last timestamp t_4 is taken by Node 1 upon the receipt of the *peer delay response* message. Figure 20 illustrates a diagram of this process.

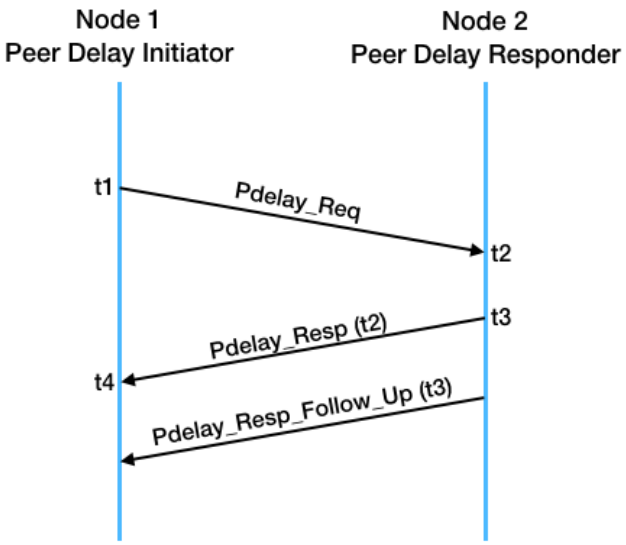

Figure 20 – Peer delay mechanism

The peer delay initiator has thus all four timestamps taken: t_1, t_2, t_3 and t_4. Note these values are constantly updated, since the peer delay mechanism exchanges messages periodically. It then computes the *neighborRateRatio* (r) and the *neighborPropDelay* (D) variables, which are used to compute the synchronized grandmaster time.

The *neighborRateRatio* variable corresponds to the measured ratio of the local clock frequency from the time-aware system at the other end of the link and the local clock frequency of this time-aware system. It is computed by equation 4.1,

$$r = \frac{t_3 - t_{30}}{t_4 - t_{40}} \tag{4.1}$$

in which t_{30} and t_{40} are the timestamps t_3 and t_4 taken in the first exchange of peer delay messages.

On the other hand, the *neighborPropDelay* variable measures the propagation delay on the link attached to this port, expressed in the time base of the time-aware system at

the other end of the link. It is computed by equation 4.2,

$$D = \frac{r(t_4 - t_1) - t_3 + t_2}{2} \tag{4.2}$$

in which r is the *neighborRateRatio* calculated by equation 4.1.

The peer delay mechanism is always running, so the propagation delay in the link to be always known to all ports of all links that run the protocol. Thus, in case of a grandmaster or network topology change, a faster reconfiguration is possible (MANN et al., 2013).

Once this mechanism is up and running, a peer is determined as capable of supporting the gPTP protocol and *asCapable* is, consequently, set to 1 for that port if:

- The port is exchanging peer delay messages with its neighbor;
- The measured propagation delay does not exceed a threshold value;
- The port only receives one *peer delay response* and *peer delay response follow up* messages for each peer delay request sent;
- If the *peer delay response* and *peer delay response follow up* messages received do not come from the time-aware system the port belongs to.

4.2.2.2 Step 3: Best Master Clock Selection and Synchronization Spanning Tree

The whole point of the IEEE 802.1AS protocol is to synchronize every time-aware system in the same gPTP domain to the same clock. This is done by selecting the best clock from the time-aware systems in the domain as a reference to which all the others have to synchronize to.

The standard specifies the Best Master Clock Algorithm (BMCA) for autonomously select the best available clock as the grandmaster and construct a time-synchronization spanning tree with the grandmaster as the root. The spanning tree works as a synchronization hierarchy for the time-aware systems and defines the path the synchronization information should travel through. Besides this, the BMCA also assigns the role each port should have, i.e., if they should be a master, a slave, a passive or a disabled port. Figure 19 shown before represents a grandmaster and its synchronization spanning tree resulted from the BMCA algorithm.

In the beginning, every time-aware system sends to the others *announce* messages containing information about its own clock along with a spanning tree. Upon the receipt of announce messages, a time-aware system evaluates whether the best clock is its own clock or the one indicated by the *announce* message received. In the latter case, it keeps the information about who is the best clock along with the associated spanning tree, and stops transmitting *announce* messages.

At some point, there will be only one time-aware system sending *announce* messages. This "last survivor" is the best clock and thus the grandmaster. It keeps periodically transmitting *announce* messages in case some time-aware system with better clock enters the domain. In this case, the new time-aware system would compare and evaluate its own clock as better than the one received and then indicate that its own clock should be the grandmaster by sending *announce* messages.

4.2.2.3 Step 4: Transport of Synchronization Information and Node Synchronization

IEEE 802.1AS defines a grandmaster and a synchronization hierarchy within an AVB network through the spanning tree resulted from the BMCA. The grandmaster, located at the root of the spanning tree, then sends synchronization information to its immediate children, i.e., to the time-aware systems it communicates with directly, with no need of a bridge. The time-aware system that receives the synchronization information then synchronizes itself to the grandmaster using the received information. This means the time-aware system is able to compute the grandmaster time corresponding to any desired local clock time.

Moreover, in case the time-aware system is a time-aware bridge, it also applies corrections to the information received and sends new synchronization information to its immediate children. This process goes on until every time-aware system receives the synchronization information. In Figure 19, it is possible to see the path the synchronization information takes from the grandmaster until each end station, passing through each bridge. It is also important to notice that the information is sent by master ports and received by slave ports.

To perform time synchronization, the synchronization information in sent by the grandmaster and received by all the other time-aware systems periodically. This information consists of a grandmaster time and a corresponding local clock time (MANN et al., 2013). The time synchronization information is sent through *Sync* and *Follow Up* messages. The correspondence between the grandmaster and the local clock time is obtained using the timestamped information from the upstream time-aware system carried in the *Follow Up* message and the timestamped of the arrival of the *Sync* message.

In order to synchronize itself to the grandmaster, a time-aware system receives from the *Sync* and *Follow Up* messages the values of *preciseOriginTimestamp*, *followUpCorrectionField* and *rateRatio* attributes. The *preciseOriginTimestamp* attribute is the source time of the grandmaster when the received time-synchronization information was sent by the grandmaster. The *followUpCorrectionField* attribute contains the accumulated time since the *preciseOriginTimestamp* was acquired by the grandmaster. It corresponds to the elapsed time between the time the grandmaster sent the received time-synchronization information and the time at which the time-synchronization information was sent by the immediate upstream time-aware system. Finally, the *rateRatio* attribute is the ratio of

the grandmaster frequency to the frequency of this time-aware system local clock.

Using these values, along with other ones obtained from the peer delay mechanism explained before, it is possible to compute the clock slave time corresponding to the grandmaster time according with the equation 4.3.

$$clockSlaveTime = preciseOriginTimestamp + followUpCorrectionField + \frac{neighborPropDelay}{neighborRateRatio * rateRatio} + dA \tag{4.3}$$

Note that the third term of the equation divides the *neighborPropDelay* by the *neighborRateRatio* and the *rateRatio*, so the propagation delay in the link is expressed in the grandmaster time base. Besides that, the last term in the equation, dA, corresponds to the delay asymmetry. It is calculated with the timestamps from the peer delay mechanism and is used to compensate for any asymmetry in the link.

4.2.3 Future perspectives for the IEEE 802.1AS standard

Future trends regarding the IEEE 802.1AS include the IEEE 802.1AS-Rev mentioned in the last section (P802..., 2018) and a new draft of the IEEE 1588 (IEEE..., 2018).

IEEE 802.1AS-Rev, which was in its Draft 5.0 by June 2017, is a revision for the IEEE 802.1AS standard and intends to improve it. It brings modifications in order to provide:

- Clock Redundancy;
- Support for more than just bridges, like routers;
- Support for aggregated links and accuracy enhancement in Wi-Fi.

On the other hand, the new IEEE 1588 version being drafted does not refer directly to the gPTP protocol. However, since the IEEE 802.1AS is based on the IEEE 1588, the advances brought by this new draft could essentially also be applied to the gPTP protocol. One of the main advances on this new draft regards to a new Annex S, addressing security aspects for the protocol (IEEE..., 2018).

5 GPTP PROTOTYPE PROPOSAL AND IMPLEMENTATION

Despite the IEEE 802.1AS standard has been published in 2011, there are not so many implementations of it available for the industry. Moreover, most are proprietary technologies restricted for some companies and some do not offer the precision required for synchronization at 1000BASE-T1 Ethernet networks.

In this chapter, the design requirements for the AVB synchronization protocol are presented along with known implementations for that protocol. Then, considerations are made on the decision for the development of a new implementation of the protocol instead of using a third-party one. After that, the implementation proposed is presented along with the details and adopted strategies, and the chosen hardware platform.

5.1 DESIGN REQUIREMENTS AND OTHER IMPLEMENTATIONS

In order to design and produce AVB end points or bridges, it is essential to understand and use the IEEE standards in which AVB stands on. One can choose between using a previous implementation of the standards or develop his own, as long as the standards specifications are obeyed. This is imperative for the correct functioning of the time-aware systems and for achieving interoperability among nodes with different implementations.

This work aims to provide an implementation for the IEEE 802.1AS protocol that achieves the nanosecond precision and can be used by AVB end-stations as well as to serve as a study platform for researchers to improve the protocol. The targeted end-station are for the automotive domain and slave-only devices, such that no master feature like the transmission of *Announce* messages need to be supported. Thus, the following points must be taken into consideration when choosing between a previous third-party or new own implementation of the protocol.

1. Fulfillment of standard requirements;
2. Protocol particularities for slave-only automotive systems;
3. Protocol particularities for end-point devices;
4. Costs of licensing x Costs of development;
5. Time-to-market;
6. Availability of technical implementation details.

IEEE 802.1AS, as explained before, is responsible for synchronizing all nodes in an AVB network to a common time reference. According with the standard, two end nodes, with fewer than 7 AVB nodes between them, are required to synchronize within a $500ns$

precision (IEEE..., 2011). Therefore, direct neighbor nodes must synchronize with a nanosecond precision (MATHEUS; KÖNIGSEDER, 2017). In order to achieve the required precision, certain constraints are made on the responsiveness and accuracy of time-aware systems. The local clock of each node is required to have a granularity less than or equal to $40ns$. The duration of time between the receipt of a time synchronization event message and the sending of the next subsequent time synchronization event message on another port, defined as Residence Time, must be within $10ms$. The duration of time between the receipt of a *peer delay request* message by a port of a time-aware system, and the sending of the corresponding *peer delay response* message, defined as pdelay turnaround time, must be within $10ms$ (IEEE..., 2011).

Besides the protocol requirements, it is important to note that automotive Ethernet devices execute gPTP operations with some differences to what is described in the standard. Automotive original equipment manufacturer (OEM) usually configure the gPTP domain and its spanning tree statically. This means that the grandmaster and all port roles from all nodes are assigned prior to the system startup and are not supposed to change. Therefore, there is no need to support the BMCA state machines (BECHTEL et al., 2015). Their functions must not be executed, so no modification is applied to the grandmaster or port roles' previous configuration. Since *announce* messages are used only for BMCA purposes, they are also neither required nor expected in an automotive network. Finally, as the spanning tree is static, time-aware bridges may perform delay measurements in only one direction. They are not required to initiate peer delay requests on its ports whose role is Master.

Added to the particularities imposed by the automotive environment targeted, the aimed implementation is for slave-only end points. Therefore, all matters regarding only bridges can be left out. This implies even more simplifications on the desired implementation. Node ports are no longer required to send *sync* and *follow up* messages, but only to receive them.

Upon the gPTP requirements and all those simplifications made, one could decide for acquiring an 802.1AS third-party implementation. This could definitely reduce the end-point time-to-market, even though licensing costs could eventually overcome development costs in the long run. However, there are not this many implementations out there for purchasing or licensing. Despite IEEE contributes to joint solutions for industry common problems, the automotive industry is so competitive; a company is not likely to allow its opponents to use implementations developed by them.

Some commercial implementations are in AVB end points from XMOS and NXP. They are already commercial end points, not standalone implementations of the gPTP protocol. Being a commercial product, implementation details are not available or possible to be modified for investigating and proposing improvements to the protocol. Thus, they would not be useful for researching the protocol itself, but only network aspects. On the other

hand, the work in (HERBER; SAEED; HERKERSDORF, 2015) is one of few references found that discusses implementation details. The authors of this research implement their own AVB end-point, so hardware resource requirements can be evaluated and timing measures can be directly embedded into the hardware for precise assessment of performance metrics.

The end point designed in (HERBER; SAEED; HERKERSDORF, 2015) extends an Ethernet MAC into an AVB controller, which includes the implementation of the IEEE 802.1Qat, IEEE 802.1Qav and IEEE 802.1AS standards. The Xilinx Zynq 7000 System on Chip (SoC) is chosen as the target hardware so it is possible to partition the protocols implementation between the software of a processing system (PS) and the Field Programmable Gate Array (FPGA) hardware of a programmable logic (PL). In addition, a Linux based operating system is used.

Time-critical portions of the protocols are best suited for a hardware implementation in the PL, since a better performance can be achieved. On the other hand, if the required performance of some portion can be achieved by the PS, this portion is implemented in software since hardware resources are scarce and valuable as well as its implementation is more complex and time consuming. The authors of the mentioned work decided for a hardware implementation for IEEE 802.1Qat and IEEE 802.1Qav standards, as flow control and traffic shaping are highly time-critical and multiple streams must be handled in parallel. However, the gPTP protocol, apart from the required real time clock (RTC), is implemented in software at the expense of a reduced precision compared with hardware realizations. They argue, supported by the work in (MAHMOOD; EXEL; SAUTER, 2014), that software based time-synchronization could achieve synchronization precision within the microseconds range and a precision lower than that would represent extra complexity for limited benefits to their product.

Contrary to their decision, our desire in this dissertation is for an end-point product capable to synchronize neighbor nodes with a nanosecond precision, as specified in the gPTP protocol. The implementation must be able to support not only 100BASE-T1 Ethernet networks, but also the recently standardized 1000BASE-T1 automotive Ethernet technology, in which the nanosecond precision is indispensable due to the higher data rates. That said, upon the few available realizations of the IEEE 802.1AS protocol, their limitations regarding nanosecond synchronization precision and the lack of implementation details, the decision is made is for developing our own gPTP protocol implementation.

5.2 DESIGN IMPLEMENTATION

Once the decision is for developing our own IEEE 802.1AS implementation, it is necessary to have in mind this must be done mostly in hardware, so the desired synchronization precision is achieved. Therefore, the platform chosen for the implementation is the same Xilinx Zynq 7000 SoC used in (HERBER; SAEED; HERKERSDORF, 2015). This device has a programmable logic and a dual-core ARM Cortex-A9 processors combining great pro-

cessing power with flexibility. Nevertheless, it has been largely used in a wide range of embedded applications including multi-camera driver assistance systems and 4K2K Ultra-HDTV. The Processing System (PS) programming is done in C language while the Programmable Logic (PL) programming is done in VHDL. The communication between them is through AXI the LogiCORE™ IP AXI4-Lite IP Interface (IPIF), which provides a point-to-point bidirectional interface between a user IP core and the Xilinx LogiCORE IP AXI Interconnect core.

In addition, it is also necessary to consider the simplifications made for the slave-only automotive end-point device to be designed:

- BMCA shall not be implemented - automotive systems/slave-only devices limitation;
- *Announce* messages shall not be supported - automotive systems/slave-only devices limitation;
- *Sync* and *Follow up* messages shall not be sent (only received) - end-point limitation (the implementation is not for bridges).

The proposed IEEE 802.1AS implementation will be part of a node in an AVB network. While in this chapter we dive in the explanation about how the protocol is implemented, in Chapter 6 we discuss how to use this implementation as part of an AVB node.

5.2.1 Entities and State Machines

The first thing to do in the implementation is to define the system architecture in functional blocks, i.e. define what components constitute a time-aware system, in our case, a time-aware end-station. According with the specification, each time-aware system is composed by a Port Structure Entity for each port of the system added to a single Local Clock Entity, Site Sync Sync Entity and Clock Slave Entity. The Port Structure Entity, in turn, is formed by other two entities, a Port Sync Entity and a Media Dependent Entity. As one can deduce, the Media Dependent Entity includes all state machines from the Media Dependent (MD) Layer. On the other hand, the Media Independent (MI) Layer is compounded by the Site Sync Sync Entity, the Clock Slave Entity and the Port Sync Entity. Figure 21 shows a representation of the time-aware system entities and layers. Each of these entities is implemented for the design and will be explained in detail in this section.

5.2.1.1 Local Clock

The whole purpose of the gPTP protocol is to synchronize all nodes in a gPTP domain to a common time reference. Thus, each node in a gPTP domain has its own local clock and needs to convert a time measure in the node time base to the grandmaster time base. Therefore, every time-aware system has a single Local Clock Entity that corresponds

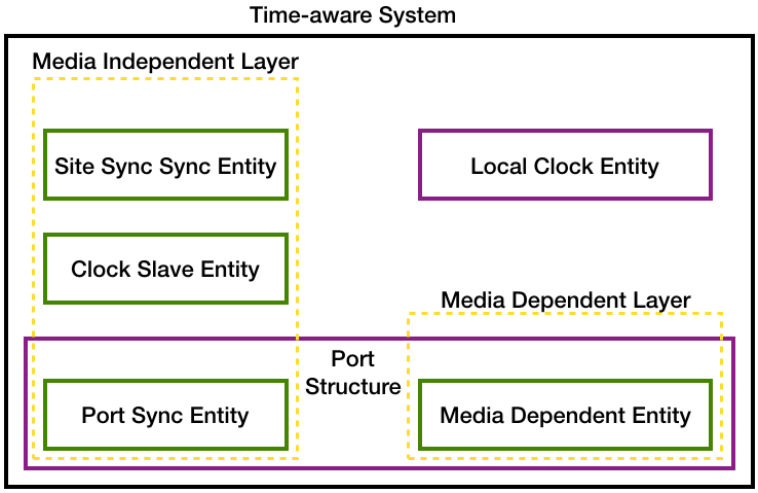

Figure 21 – Entities and Layers of a time-aware system

to a real time clock used for all state machines and operations within the time-aware system, and a Clock Slave Entity responsible for the time base conversion performed with values received from other state machines, as will be discussed later. As said before, the protocol specification requires a granularity of less than or equal to $40ns$. In this project, a granularity of $10ns$ is chosen so every clock cycle takes $10ns$ to be executed.

5.2.1.2 Peer Delay Mechanism

The peer delay mechanism is responsible for computing the *neighborRateRatio* and the *neighborPropDelay* as well as determining whether a port is capable of communicating to the port attached at the other end of the link using the gPTP protocol. This mechanism is performed by two state machines in the PL, which belong to the media dependent entity in the port structure instantiated for each port: *MD_PDelay_Resquest* state machine and *MD_PDelay_Response* state machine.

MD_PDelay_Resquest periodically sends a *peer delay request* message to the port attached at the other end of the link, and then waits for the *peer delay response* and *peer delay response follow up* messages. By default, the wait period is $1s$, but it can configurable. The time in which the *peer delay request* is sent is timestamped as t_1, while the time in which the *peer delay response* is received is timestamped as t_4.

At the port in the other end of link, the *MD_PDelay_Response* state machine from this other time-aware system waits for a *peer delay request*. Upon its receipt, it constructs a *peer delay response* message and sends it to the port from which the request came from. Within this message, the time in which the request was received, timestamped as t_2, is carried. Then, it constructs a *peer delay response follow up* message and also sends it to the port from which the request came from. Within this message, time in which the response message was sent, timestamped as t_3, is carried. Figure 22 shows a diagram of the peer delay mechanism state machines and messages involved in the peer delay mechanism.

Figure 23 shows the same diagram of Fig. 20, in which the timestamps and messages from the peer delay mechanism are represented.

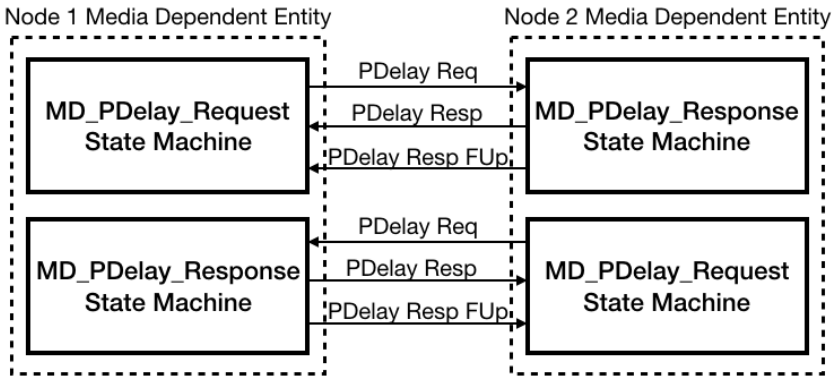

Figure 22 – Peer Delay Mechanism state machines and messages

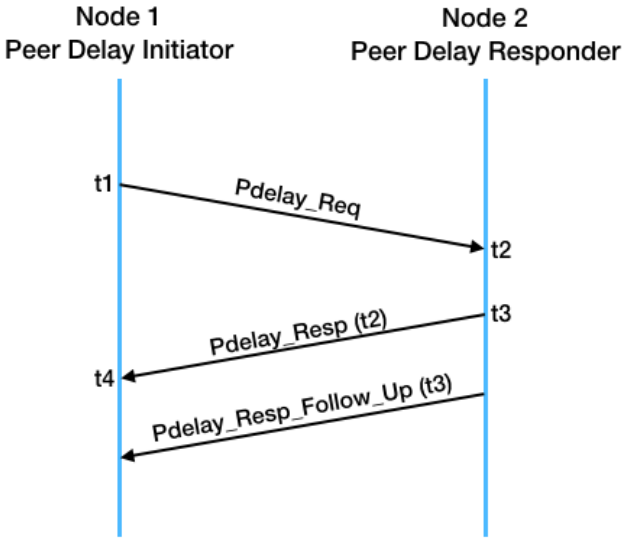

Figure 23 – Peer delay mechanism

Upon the receipt of timestamps t_3 and t_4, the *MD_PDelay_Resquest* state machine has all four timestamps needed for computing $neighborRateRatio$ and $neighborPropDelay$, as defined by equations 4.1 and 4.2. The *neighborPropDelay* D and each timestamped time are composed by a 48-bit field for seconds and a 32-bit field for nanoseconds, while the *neighborRateRatio* r is a double precision floating point type with 64 bits. The double precision floating point combined with the division in the equations are not easily performed in the PL. This type of data and the division operation by some value other than a power of 2 increases the utilization of the FPGA resources and require significant extra execution time. Since all operations must be executed within one clock cycle time, which is only $10ns$, timing can be compromised.

An alternative approach is needed and the strategy is to take advantage of the combined PL and PS resources in the SoC. Thus, the *MD_PDelay_Resquest* state machine

sends all operands in equations 4.1 and 4.2 and makes a calculation request for the PS. Then, the PS receives the operands values, performs the requested computation using the processor resources and sends the results back to the PL. The only concern about this approach is whether the time to send the operands to the PS, execute the operation and get the results back may surpass the period of sending the next *peer delay request* message. Experimental tests were performed in this work and this approach was verified to consume only a few microseconds. Since the period between peer delay requests is usually around 1s, the strategy adopted is well suited. Once the propagation delay is received from the PS, the PL simply compares it with a previously configured threshold value and attributes *TRUE* to the *asCapable* variable if the propagation delay does no overcome the threshold. Figures 24 and 25 show the *MD_PDelay_Resquest* and *MD_PDelay_Response* state machine diagrams.

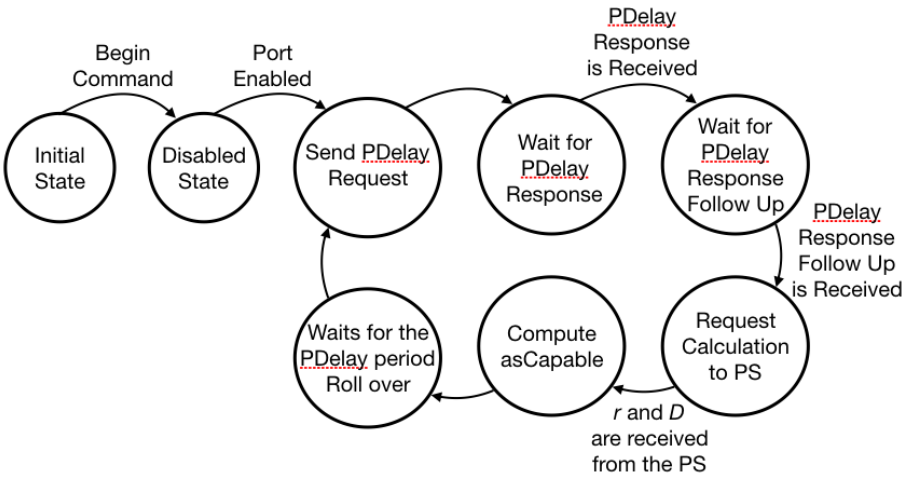

Figure 24 – *MD_PDelay_Resquest* state machine diagram

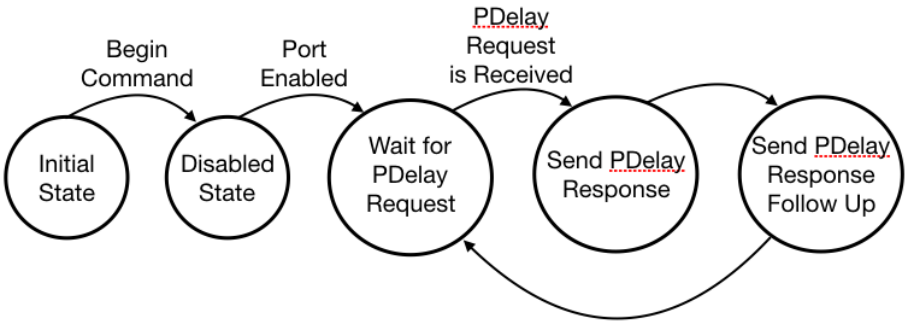

Figure 25 – *MD_PDelay_Response* state machine diagram

These two state machines cover the first two steps defined in 4.2.2. The third step corresponds to the best master clock selection and establishment of a synchronization spanning tree, referring to the BMCA. However, since automotive systems do not need to support it, we concentrate on the fourth step of the list.

5.2.1.3 Transport of Time Synchronization

The transport of time-synchronization information involves six state machines, four from the media independent layer and two from the media dependent layer. The latter two are the *MD_SyncReceive* and *MD_SyncSend* state machines in the MD Entity. The former four are the *PortSyncSyncReceive* and *PortSyncSyncSend* state machines in the PortSync Entity, the *SiteSyncSync* state machine in the SiteSync Entity and the *ClockSlave* state machine in the Clock Slave Entity.

The *MD_SyncReceive* state machine receives *Sync* and *Follow Up* messages and sends the time-synchronization information carried in these messages to the *PortSyncSyncReceive* state machine, in the Port Sync entity of the same port, using a Sync Receive Structure. The *sync* message supplies the Sync Receive Structure with the *port identity* information that identifies the port from which the synchronization information was originated from. The *port identity* contains an 8-byte MAC address in the IEEE EUI-64 format to identify the time-aware system and a *port number* value to identify the port within the time-aware system (IEEE..., 2011). It goes into the *sourcePortIdentity* field of the Sync Receive Structure. Besides this information, the *sync* message also brings to the structure a *logMessageInterval* value, used for determining how long the state machine should wait for the *follow up* message to arrive.

The main attributes acquired from the *Follow Up* message to the synchronization process are the *preciseOriginTimestamp*, the *correctionField* and the *cumulativeScaledRateOffset*. The *preciseOriginTimestamp* is the grandmaster time when the received time-synchronization information was sent. This attribute is directly passed to the Sync Receive Structure. The *correctionField* corresponds to corrections for fractional nanoseconds, it is chosen so that its sum with the *preciseOriginTimestamp* is the grandmaster time that corresponds to the local time of when the *Sync* message was sent. It is used for computing the *followUpCorrectionField* of the Sync Receive Structure, which corresponds to the elapsed time between the time the grandmaster sent the received time-synchronization information and the time at which the time-synchronization information was sent by the immediate upstream time-aware system. The *cumulativeScaledRateOffset* is a scaled rate offset accumulated from the time-aware systems on the spanning tree from the grandmaster towards the immediate upstream node. It is used for computing the *rateRatio* attribute from the Sync Receive Structure, that is the ratio of the grandmaster frequency to the frequency of the LocalClock entity of the time-aware system at the other end of the link attached to this port.

Another important information attached to the Sync Receive Structure is the *upstreamTxTime* attribute, that corresponds to the time in which the *Sync* message was received minus the propagation time on the link attached to this port divided by *neighborRateRatio*. In order to calculate it, the moment in which the *Sync* message is received is timestamped, while the propagation delay on the link (*neighborPropDelay*) and the

neighborRateRatio are already computed by the peer delay mechanism. Also, in case the link is not symmetric, a delay asymmetry factor *dA*, obtained from the timestamps taken in the peer delay mechanism, has to be taken into consideration for compensating the asymmetry. The value of *dA* is then divided by *rateRatio* and subtracted from the *Sync* message timestamp.

Just as in the *MD_PDelay_Resquest*, the *rateRatio* and *upstreamTxTime* computations involve multiplications and divisions that affect the implementation timing and performance. Thus, the same strategy of sending the operands and performing the necessary calculations on the PS in adopted here. This strategy is also well suited for the synchronization process because the default period between *sync* messages is $125ms$, so it is affordable to spend a few microseconds in the calculations. Observe in table 9 the main attributes in a Sync Receive Structure and in Fig. 26 a simplified state machine diagram for *MD_SyncReceive* state machine.

Table 9 – Sync Receive Structure Main Attributes

Sync Receive Structure Main Attributes
sourcePortIdentity
logMessageInterval
preciseOriginTimestamp
followUpCorrectionField
rateRatio
upstreamTxTime

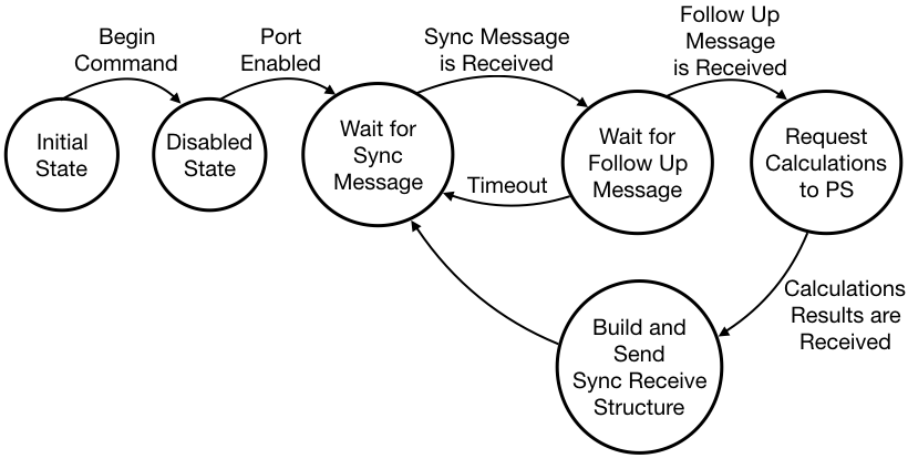

Figure 26 – *MD_SyncReceive* state machine diagram

The *PortSyncSyncReceive* state machine, in the Port Sync Entity of the same port, receives the Sync Receive Structure and uses the information carried in it to compute the accumulated *rateRatio* and the *syncReceiptTimeoutTime*. Then, a Port Sync Sync Structure is built, using these computed values and others from the Sync Receive Structure, and sent to the *SiteSyncSync* state machine.

The *rateRatio* received from Sync Receive Structure is the ratio of the grandmaster frequency to the frequency of the LocalClock entity of the time-aware system at the other end of the link attached to this port. On the other hand, the *neighborRateRatio* is the measured ratio of the local clock frequency of the time-aware system at the other end of the link attached to this port, to the local clock frequency of this time-aware system. Then, the accumulated *rateRatio* adds these both values and subtracts one to obtain the ratio of the grandmaster frequency to the frequency of the current time-aware system.

The *syncReceiptTimeoutTime* is computed from the logMessageInterval by using the PS for performing calculations. It corresponds to the time interval expected between subsequent *Sync* messages, i.e. the time at which sync receipt timeout occurs if a subsequent time-synchronization event message is not received by that time.

The *sourcePortIdentity*, *logMessageInterval*, *preciseOriginTimestamp*, *followUpCorrectionField* and *upstreamTxTime* attributes from the Sync Receive Structure are directly passed to the Port Sync Sync Structure. Observe in Table 10 the main attributes from Port Sync Sync Structure and in Fig. 27 a block diagram for the state machines seen so far regarding the synchronization process.

Table 10 – Port Sync Sync Structure Main Attributes

Port Sync Sync Structure Main Attributes
sourcePortIdentity
logMessageInterval
preciseOriginTimestamp
followUpCorrectionField
upstreamTxTime
accumulated rateRatio
syncReceiptTimeoutTime

Following the synchronization process, the single *SiteSyncSync* state machine in the time-aware system receives a Port Sync Sync Structure from every port, as each port structure receives *Sync* and *Follow Up* messages and has a *MD_SyncReceive* and a *PortSyncSyncReceive* state machine. The *SiteSync* state machine then forwards back to the *PortSyncSyncSend* state machine of all Port Entities the Port Sync Sync Structure received by the time-aware system slave port. Remember that only one port in a time-aware system can have a slave port role. Thus, the *SiteSyncSync* state machine can be thought as a multiplexer that receives synchronization information from every port and forwards back to them only the information coming from the slave port.

The *PortSyncSyncSend* state machine receives the time-synchronization information, received at the time-aware system slave port, from the Site Sync Entity in a Port Sync Sync structure. Then, it builds a Sync Send Structure that has the same fields of the Sync Receive structure, so its main attributes are the same ones displayed in table 9. This structure shall be sent to the MD Entity if, and only if, the present port has a master role. Therefore, the main purpose of the *PortSyncSyncSend* state machine is to guarantee

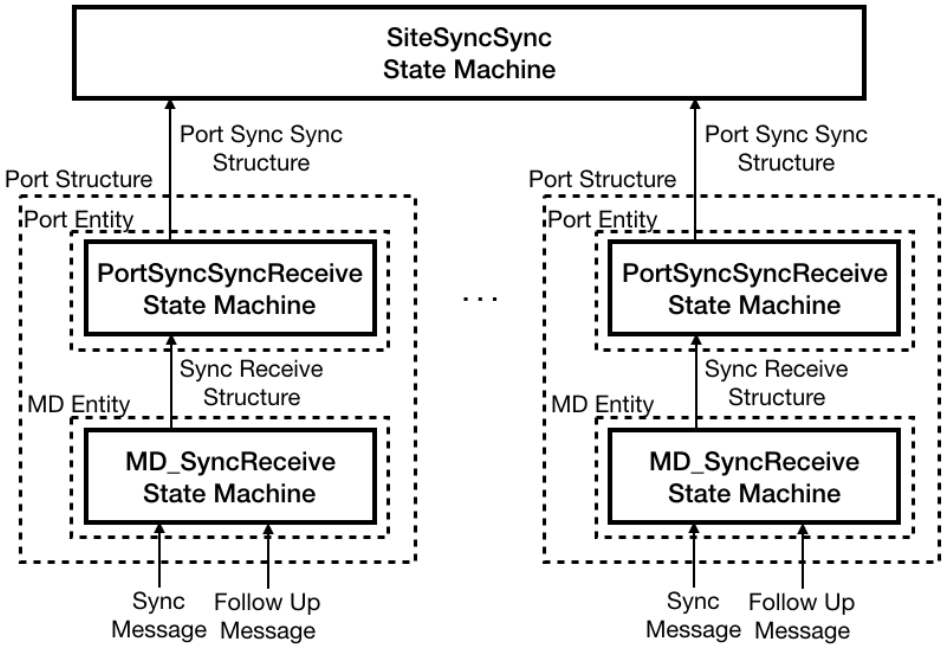

Figure 27 – Synchronization process partial block diagram

the synchronization information is only forwarded to the MD Entity of master ports, as they are the only ports *sync* and *follow up* messages should be sent from.

The *MD_SyncSend* state machine in the MD entity is the one who receives the Sync Send Structure containing the synchronization information. This state machine does the opposite of the *MD_SyncReceive* state machine. It receives the Sync Send Structure and uses the information contained in it to build a *sync* and a *follow up* message, which are sent to the MAC layer. In this process, calculations involving double precision floating points are required to compute some of the *follow up* message attributes, such as the *cumulativeScaledRateOffset*. Like before, the PS is used for accomplishing these calculations. Observe in Fig. 28 the *MD_SyncSend* state machine diagram.

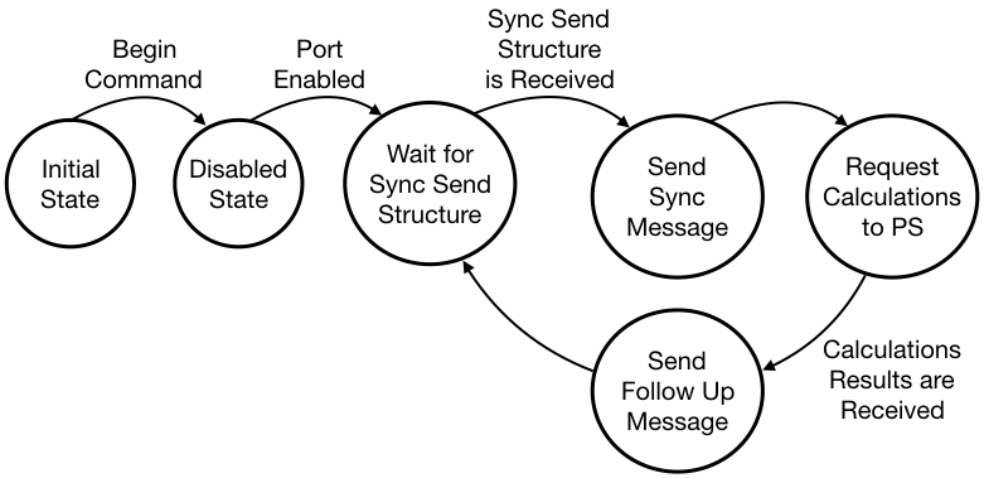

Figure 28 – *MD_SyncSend* state machine diagram

Observe in Fig. 29 the synchronization process message flow discussed so far. Note the SiteSyncSync state machine forwards to all port structures the Port Sync Sync Structure received from the slave port. Also note only master ports send *sync* and *follow up* messages, as only them send a a Sync Send Structure from the *PortSyncSyncSend* state machine to the *MD_SyncSend* state machine.

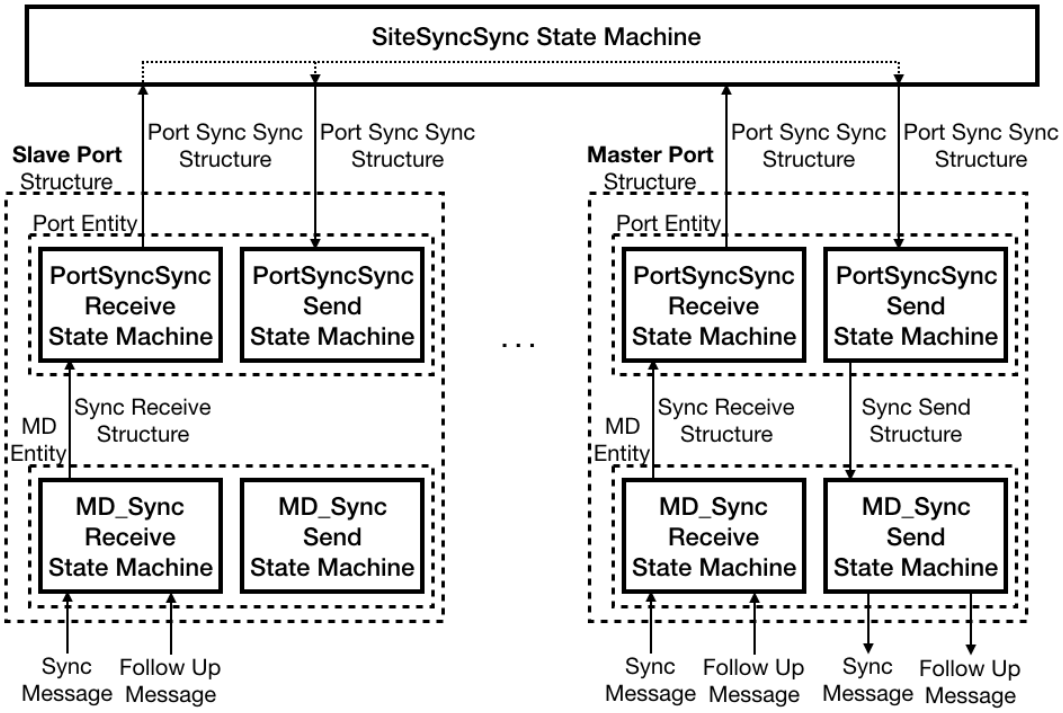

Figure 29 – Synchronization process message flow

Now, the *ClockSlave* in the Clock Slave Entity is the only state machine left to be explained regarding the transport of time-synchronization. This state machine is the one responsible for the actual synchronization of the time-aware system to the grandmaster clock. In order to do so, it receives the Port Sync Sync Structure from the *SiteSyncSync* state machine every time a *sync* and a *follow up* message have been received by the slave port. Then, it performs calculations to obtain a *clockSlaveTime* and a *clockSlaveTime_ready* signal. The former, compounded by a 48 bits seconds field and a 32 bis nanoseconds field, corresponds to the actual timestamp used for synchronizing the slave node to the grandmaster clock. The latter is simply a 1 bit signal that indicates whenever the *clockSlaveTime* is updated.

The *clockSlaveTime* is computed according with the equation 5.1. In this equation, the terms *preciseOriginTimestamp*, *followUpCorrectionField* and *rateRatio* are received from the Port Sync Sync Structure, while *neighborPropDelay*, *neighborRateRatio* and *dA* are obtained from the peer delay mechanism. It is executed every time a *sync* and a *follow up* message arrives in a slave port and, consequently, a new Port Sync Sync Structure is received at the *ClockSlaveSM* state machine from the *SiteSyncSync* state machine. Also,

every time this computation is performed, a pulse is sent on the *clockSlaveTime_ready* signal, so the node knows when to re-synchronize itself to the grandmaster clock.

$$clockSlaveTime = preciseOriginTimestamp + followUpCorrectionField+$$

$$(neighborPropDelay/neighborRateRatio) * rateRatio + dA; \tag{5.1}$$

This concludes the transport of time-synchronization information and node synchronization processes, fourth step in the list 4.2.2. Observe in Fig. 30 a diagram exhibiting all state machines involved in the time-synchronization process and in the peer delay mechanism. Note that the *LinkDelaySyncIntervalSetting* state machine is the only one in the diagram that has not been explained yet. This state machine is responsible for configuring the *pdelayReqInterval* and *syncInterval* signals, which corresponds to the time interval between *Peer Delay Request* messages and *Sync* messages, respectively. In other words, it configures the periodicity in which *Peer Delay Request* messages and *Sync* messages are sent by the time-aware system. For this purpose, a *Signaling* message that carries information about these periods is received and then used for making the requested configurations.

In spite of all state machines represented in Fig. 30 have been implemented, some of them are not needed by slave-only AVB end stations. AVB end stations do not need to forward any synchronization information through *sync* and *follow up* messages. Therefore, even though the *PortSyncSyncSend* and *MD_SyncSend* state machines have been implemented, they are only needed for bridges.

5.2.2 MAC Layer Interface

The proposed implementation for the gPTP protocol has already been detailed explained with regard to the protocol operation. The peer delay mechanism, transport of time synchronization and node synchronization were presented by means of the state machines implemented and messages' structures involved in these processes. However, it is also necessary to discuss how the gPTP Core, designed in Fig. 30, interfaces with the MAC Layer, i.e. how the messages are received from and sent to the Ethernet MAC Layer and, consequently, to the PHY and the other nodes in the network.

First, we summarize all messages received and sent by the gPTP Core from Fig. 30. Table 11 presents the purpose and flow of the gPTP messages, while Table 12 describes their size in bytes as well as the main fields of each message. In addition to the six gPTP messages in these tables, the gPTP protocol also specifies an *Announce* message. This message is used by the BMCA state machines and brings information related to the grandmaster selection and spanning tree formation. Since, those are static and preconfigured for our case, there is no need to support *Announce* messages in the design.

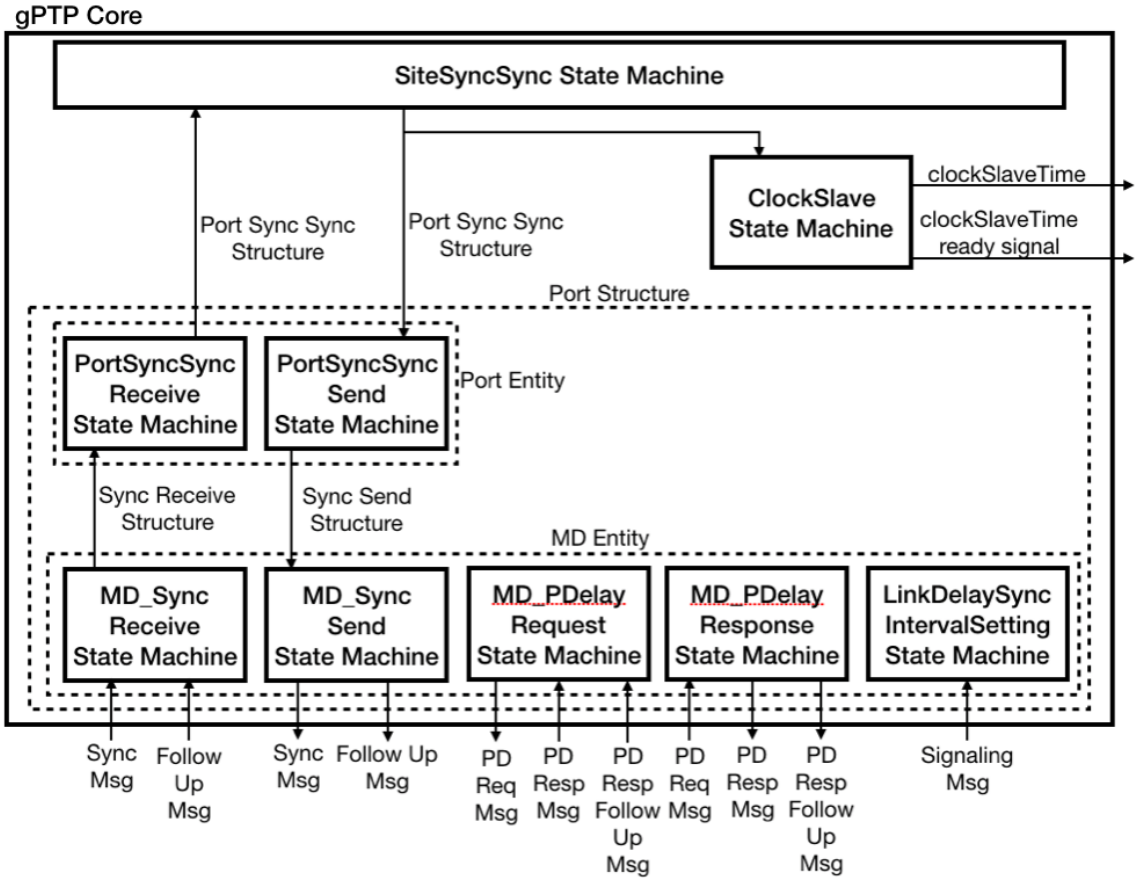

Figure 30 – gPTP Core

Table 11 – gPTP messages traffic flow in slave-only AVB end stations

gPTP Message	**Sent from state machine**	**Received by state machine**	**Purpose**
Signaling	-	*LinkDelaySyncIntervalSetting*	Configure Sync and PDelay Request messages periods
Sync	*MD_SyncSend*	*MD_SyncReceive*	Forward/Receive Synchronization Information
Follow Up	*MD_SyncSend*	*MD_SyncReceive*	Forward/Receive Synchronization Information
PDelay Request	*MD_PDelayRequest*	*MD_PDelayResponse* state machine	Peer Delay Mechanism
PDelay Response	*MD_PDelayResponse*	*MD_PDelayRequest*	Peer Delay Mechanism
PDelay Response Follow Up	*MD_PDelayResponse*	*MD_PDelayRequest*	Peer Delay Mechanism

Table 12 – gPTP messages

gPTP Message	**Message Length**	**Message Fields**		
Signaling	60 bytes	Header (34 bytes)	*targetPortIdentity* (10 bytes)	*messageIntervalRequestTLV* (16 bytes)
Sync	44 bytes	Header (34 bytes)	Reserved (10 bytes)	
Follow Up	76 bytes	Header (34 bytes)	*preciseOriginTimestamp* (10 bytes)	*Follow_Up_information_TLV* (32 bytes)
PDelay Request	54 bytes	Header (34 bytes)	Reserved (10 bytes)	Reserved (10 bytes)
PDelay Response	54 bytes	Header (34 bytes)	*requestReceiptTimestamp* (10 bytes)	*requestingPortIdentity* (10 bytes)
PDelay Response Follow Up	54 bytes	Header (34 bytes)	*responseOriginTimestamp* (10 bytes)	*requestingPortIdentity* (10 bytes)

The gPTP messages contain information from the concern of only the state machines within the gPTP core, while the MAC layer (in which they are supposed to be sent to and received from) only supports Ethernet frames whose format is exposed in Fig. 31. Thus, the interface between the gPTP core and MAC layer must handle a format conversion by encapsulating/extracting gPTP messages in/from Ethernet frames. For this purpose, a Build Frames and a Build Messages modules are also designed in the SoC programmable

logic.

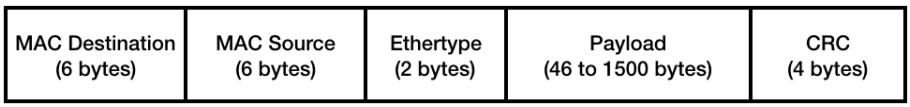

Figure 31 – Ethernet frame format

The Build Frames module receives the messages to be sent to the MAC layer from the gPTP Core and encapsulate them to Ethernet frames by adding the gPTP message to the Ethernet frame payload field, and filling the other fields in the frame as follows. The MAC Destination has the fixed hexadecimal value 01-80-C2-00-00-0E; the MAC source receives the MAC address of the node; the Ethertype has the fixed hexadecimal value 88F7; and the cyclic redundancy check (CRC) is filled with an appropriated error-detecting code. Once the Ethernet frames are built, they are sent to First-in First-out (FIFO) structure connected to the MAC layer. In this way, the gPTP messages received by the Build Frames module are encapsulated to Ethernet frames one by one and sent to the FIFO, which sends the frames to the MAC layer.

The Build Messages module works in the opposite way from the Build Frames. A FIFO receives Ethernet frames containing gPTP messages from the MAC layer and forwards these frames to the Build Messages module, which extracts the encapsulated gPTP messages. Once a message is extracted, the Build Messages module forwards it to the appropriate state machines in the gPTP Core, as summarized in Table 11, and receives another Ethernet frame from the FIFO. For example, a received *sync* message is forwarded to the *MD_SyncReceive* state machine, while a received *Peer Delay Response* message is forwarded to the *MD_PDelayRequest* state machine.

The FIFOs have a 64-bit width, such that the Ethernet frames are sent and received in pieces of eight bytes. This means the Build Frames and Build Messages modules not only encapsulate and extract gPTP messages, but also splits and puts together Ethernet frames in 8 bytes packets. In order to identify which Ethernet frames contain gPTP messages and to define how many eight bytes packets form the Ethernet frame, eight extra bytes are included in the beginning of the Ethernet frame. These bytes are constituted by an identification code, with hexadecimal value of 0x5555AAAA, and the length of the Ethernet frame.

Considering the interface with the MAC layer, the final design is offered in the form of an Intellectual Property (IP) that can be included by any project in need of a IEEE 802.1AS implementation for slave-only end stations. A diagram for this IP is exhibited in Fig. 32.

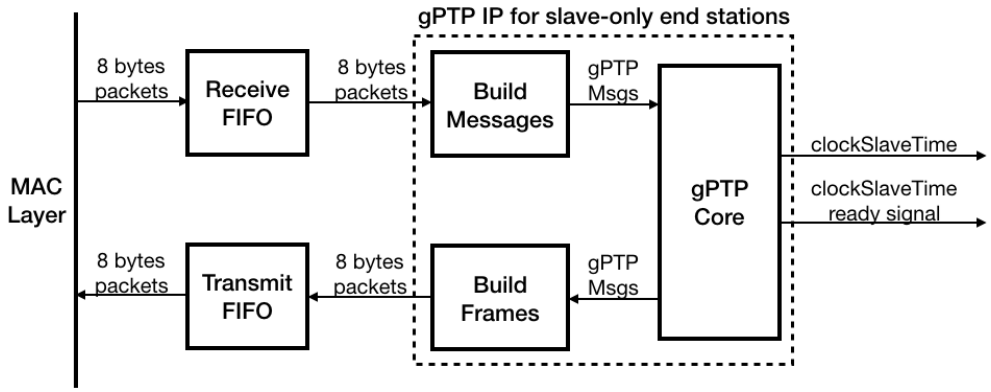

Figure 32 – gPTP IP design

6 GPTP PROTOTYPE EVALUATION

After its implementation, the IEEE 802.1AS protocol design proposed in chapter 5 needs to be tested and validated with regard to its feasibility and behavior. The implementation feasibility needs to be verified in a hardware platform to ensure that timing and utilization limitations are satisfied. For this purpose, reports are acquired from the Xilinx Vivado tool used and then analyzed. Furthermore, in order to verify the implementation behavior, i.e. if the design works in the way it should, an experimental setup is established along with a testing methodology.

This chapter then presents and analyzes timing and utilization reports for the design implementation on Chapter 5. Then, it proposes a testing methodology based on simulation and hardware tests as well as the their results. The intention is to prove that the developed implementation fulfills the IEEE 802.1AS standard requirements and works properly.

6.1 DESIGN ANALYSIS AND ITS USE AS A COMPONENT

In Chapter 5, the gPTP IP implementation for slave-only end-stations was detailed considering all its modules and state machines, as exhibited in Fig. 32. In order to test this implementation, it is necessary to create a larger project that uses the IP and also other elements needed for this IP to work, such as the FIFOs that interface with the MAC layer. Observe in Fig. 33 the register-transfer level (RTL) diagram obtained in Vivado for this larger project.

It is composed by nine modules, in which one of them it the gPTP IP to be evaluated, named as gptp3_0, which has seven inputs and five outputs as described in Tables 13 and 14, respectively. Note that, apart from the `clockSlaveTime` and `clockSlaveTime_ready` outputs, highlighted in Fig. 32, most of the other gPTP IP inputs and outputs are used for interfacing with two FIFOs and exchanging data with the processor system. As seen before, the FIFOs are used for interfacing the gPTP IP with the MAC layer and the processor system is used to perform calculations required by the IEEE 802.1AS standard.

Table 13 – gptp3_0 IP Inputs

Input/Output	**Description**
S00_AXI	AXI interface connection for performing configurations and calculations using the processor.
RTC_input	Real time clock used in the timestamps operations.
receive_ff_Empty	Indicates if the receive FIFO (FWFT_FIFO2_0) is empty.
receive_ff_DataOut	Data that comes from the receive FIFO (FWFT_FIFO2_0) to the gPTP IP.
transmit_ff_Full	Indicates if the transmit FIFO (FWFT_FIFO2_1) is full.
s00_axi_aclk	clock that runs all state machines and modules in the project.
s00_axi_aresetn	asynchronous reset used for all state machines and modules in the project.

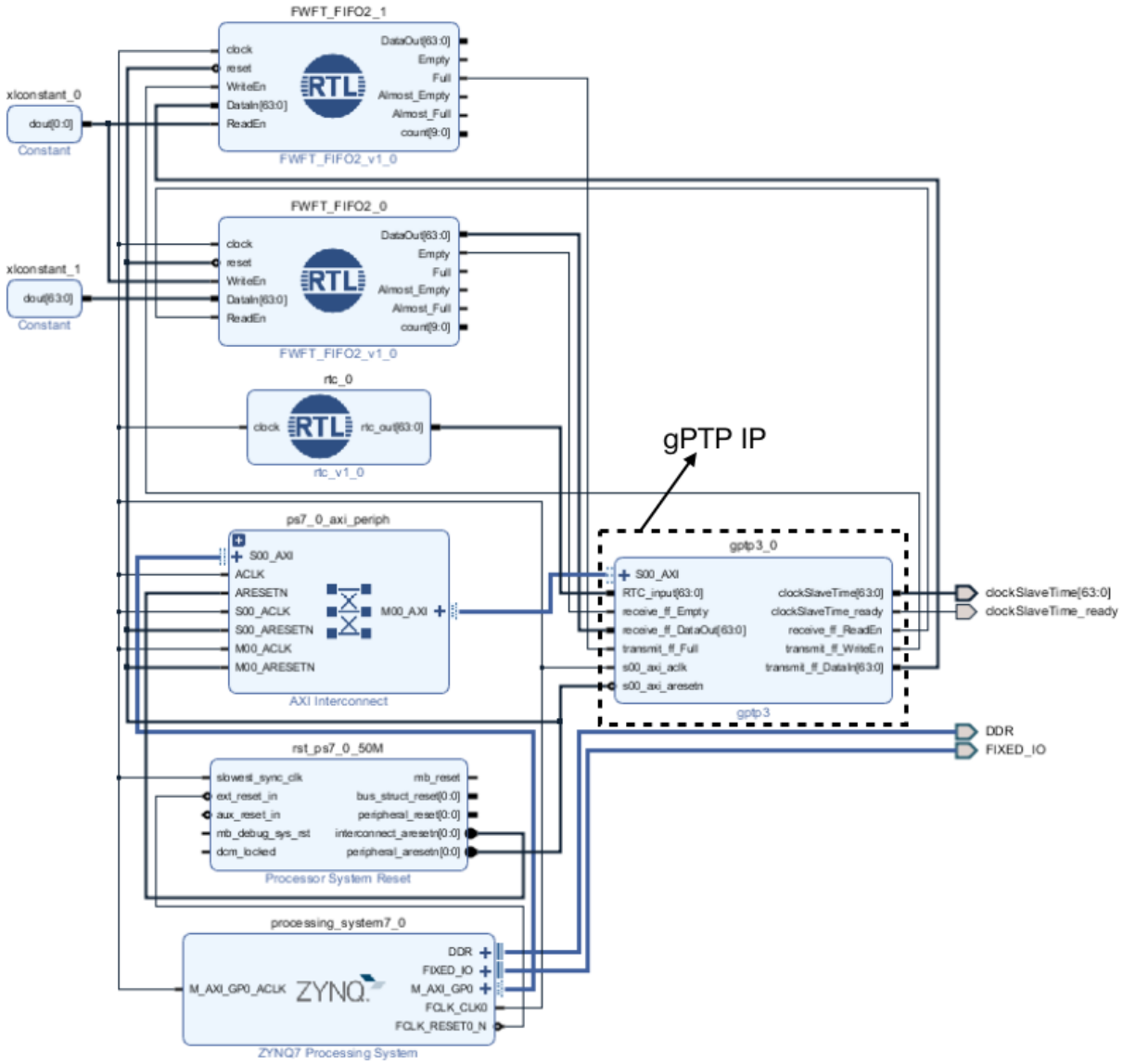

Figure 33 – RTL diagram of project that uses the gPTP IP designed

Table 14 – gptp3_0 IP Outputs

Input/Output	Description
clockSlaveTime	Timestamp used for synchronizing the slave node to the grandmaster clock.
clockSlaveTime_ready	Indicates whenever clockSlaveTime is updated.
receive_ff_ReadEn	Receive FIFO (FWFT_FIFO2_0) read enable signal used for requesting data to be read.
transmit_ff_WriteEn	Transmit FIFO (FWFT_FIFO2_1) write enable, used for requesting data to be written.
transmit_ff_DataIn	Data that goes from the gPTP IP to the transmit FIFO (FWFT_FIFO2_1).

Two of the other eight modules presented in Fig. 33 are FIFOs used to interface the gPTP IP with the MAC layer. The FIFO FWFT_FIFO2_1 corresponds to the transmit FIFO exhibited in Fig. 32. It receives 8-byte packets containing the Ethernet frames that encapsulate the gPTP messages to be transmitted to the MAC layer. The 8-byte packets are received in the input `DataIn`, while the output `DataOut` (not connected) would be connected to the MAC layer. On the other hand, the FIFO FWFT_FIFO2_0 corresponds

to the receive FIFO exhibited in Fig. 32. It receives from the MAC layer 8-byte packets containing the Ethernet frames that encapsulate the gPTP messages and transmit them to the gPTP IP. The FIFO input `DataIn`, connected to a Constant module, would actually be connected to the MAC layer, while its output `DataOut` is connected to the gPTP IP, so the 8 bytes packets are transmitted. As described in Tables 13 and 14, the FIFOs modules have some status signals reporting whether the FIFO is full or empty. Those are used by the FIFOs control, so there is to attempt to write in a FIFO that is full or to read from a FIFO that is empty.

Other two modules are simply Constant modules connected to the FIFOs. They are used only for obtaining the design utilization and timing reports, as no input can be left unconnected. The xlconstant_0 writes 1 to the FWFT_FIFO2_1 (transmit FIFO) `read enable` and to the FWFT_FIFO2_0 (receive FIFO) `write enable`, so the transmit FIFO can always read data coming from the gPTP IP and the receive FIFO can always send data to the gPTP IP. On the other hand, xlconstant_1 simulates the input that comes from the MAC layer. The use of a constant value does not affect the utilization and timing reports, as the actual connection with the MAC layer is not evaluated, only its interface.

The rtc_0 module is a real time clock used as time reference by the gPTP IP Local-Clock entity. It is a 64-bit signal with resolution of 10 nanoseconds. The gPTP modules inside the IP perform format conversions when timestamps are taken, since the standard uses a timestamp format with a 48-bit field for seconds and a 32-bit field for nanoseconds.

The processing_system7_0 corresponds to the processor present in the Zynq chip. It is used by the gPTP IP in order to perform configurations and calculations whenever necessary, as explained in Chapter 5. Combining hardware IP modules with the processing system is the great advantage of using a system-on-chip (SoC) architecture. It allows the complex calculations required in the gPTP protocol to be performed by the processor, while the FPGA hardware handles all the data flow and calculation requests.

The last two modules are also related to the processing system. The ps7_0_axi_periph is the AXI Interconnect IP core used as interface between the gPTP IP (gptp3_0) and the processing system (processing_system7_0). It handles all data transfer between these two modules. The rst_ps7_0_50M module, on the other hand, is responsible for the processor system reset. One of its outputs is also used for resetting the gPTP IP and FIFOs modules.

6.1.1 Design Reports

Field Programmable Gate Arrays (FPGAs) are pre-fabricated silicon devices that can be electrically programmed in the field to become almost any kind of digital circuit or system (FAROOQ; MARRAKCHI; MEHREZ, 2012). Normally, they are constituted by Configurable Logical Blocks (CLBs) that implement logic functions, I/O blocks that make off-chip connections, and programmable routing that interconnect all these elements. A CLB element contains a pair of slices, each comprised by Look-up Tables (LUTs) and storage elements.

The former are tables that determine what should be the output for any given input, they implement logical functions. The latter can be Look-up Table RAM (LUTRAM), which are distributed RAM memories; Flip-Flop (FF) elements; or registers. In order to save resources, distributed RAM can be replaced by Block RAM (BRAM) units, which are dedicated blocks of RAM memory; and carry logic can be moved to Digital Signal Processing (DSP) slices, which are dedicated digital signal processing units. This is just a brief explanation about FPGA's resources, so one can understand better the reports to be presented in this subsection. Further information can be found in manuals and user guides from FPGA manufacturers.

The design described in Fig. 33 is synthesized and implemented in Vivado, such that utilization and timing reports are obtained. An Integrated Logic Analyzer (ILA) is also included in the design, so signals can be probed and inspected to verify the design correct functioning. The ILA works as another module in the design, such that it also utilizes resources from the chip. All reports obtained from Vivado and displayed in this chapter consider the xc7z020clg400-1 part as Xilinx Zynq chip and also the resources needed for the ILA.

Figures 34 and 35 show a summary of the chip resources utilization for the whole project in Fig. 33 and the ILA. Figures 36 and 37 bring a more detailed utilization report in absolute and percentage values of the utilization, respectively. Note that the design fits the chosen chip. If it demanded more resources than the chip can offer, either the design would have to be improved or the chip would have to be replaced by a bigger (and more expensive) one.

Resource	Utilization	Available	Utilization %
LUT	15133	53200	28.45
LUTRAM	1588	17400	9.13
FF	16351	106400	15.37
BRAM	7.50	140	5.36
DSP	8	220	3.64
IO	65	125	52.00

Figure 34 – Project from Fig. 33 utilization report

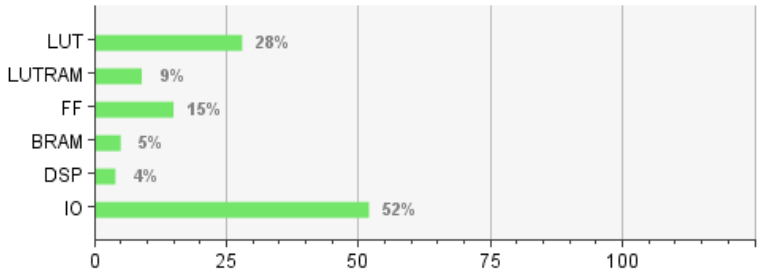

Figure 35 – Project from Fig. 33 utilization report

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	LUT Flip Flop Pairs (53200)	Block RAM Tile (140)	DSPs (220)	Bonded IOB (125)
gptp_wrapper	15133	16351	423	80	6316	13545	1588	5049	7.5	8	65
dbg_hub (dbg_hub_CV)	413	697	1	0	215	389	24	248	0	0	0
gptp_i (gptp)	14055	14459	416	80	5769	12617	1438	4453	0	8	0
FWFT_FIFO2_0 (gptp_F...	985	106	64	0	284	297	688	106	0	0	0
FWFT_FIFO2_1 (gptp_F...	986	106	64	0	287	298	688	106	0	0	0
gptp3_0 (gptp_gptp3_0_0)	11702	13628	288	80	5094	11702	0	4014	0	8	0
processing_system7_0 (...	0	0	0	0	0	0	0	0	0	0	0
ps7_0_axi_periph (gptp_...	366	466	0	0	162	305	61	205	0	0	0
rst_ps7_0_50M (gptp_rst...	15	25	0	0	11	14	1	13	0	0	0
rtc_0 (gptp_rtc_0_0)	1	128	0	0	23	1	0	1	0	0	0
xlconstant_0 (gptp_xlcon...	0	0	0	0	0	0	0	0	0	0	0
xlconstant_1 (gptp_xlcon...	0	0	0	0	0	0	0	0	0	0	0
u_ila_0 (u_ila_0_CV)	665	1195	6	0	377	539	126	322	7.5	0	0

Figure 36 – RTL diagram of project that uses the gPTP IP designed

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	LUT Flip Flop Pairs (53200)	Block RAM Tile (140)	DSPs (220)	Bonded IOB (125)
gptp_wrapper	28.45%	15.37%	1.59%	0.60%	47.49%	25.46%	9.13%	9.49%	5.36%	3.64%	52.00%
dbg_hub (dbg_hub_CV)	0.78%	0.66%	<0.01%	0.00%	1.62%	0.73%	0.14%	0.47%	0.00%	0.00%	0.00%
gptp_i (gptp)	26.42%	13.59%	1.56%	0.60%	43.38%	23.72%	8.26%	8.37%	0.00%	3.64%	0.00%
FWFT_FIFO2_0 (gptp_F...	1.85%	0.10%	0.24%	0.00%	2.14%	0.56%	3.95%	0.20%	0.00%	0.00%	0.00%
FWFT_FIFO2_1 (gptp_F...	1.85%	0.10%	0.24%	0.00%	2.16%	0.56%	3.95%	0.20%	0.00%	0.00%	0.00%
gptp3_0 (gptp_gptp3_0_0)	22.00%	12.81%	1.08%	0.60%	38.30%	22.00%	0.00%	7.55%	0.00%	3.64%	0.00%
processing_system7_0 (...	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
ps7_0_axi_periph (gptp_...	0.69%	0.44%	0.00%	0.00%	1.22%	0.57%	0.35%	0.39%	0.00%	0.00%	0.00%
rst_ps7_0_50M (gptp_rst...	0.03%	0.02%	0.00%	0.00%	0.08%	0.03%	<0.01%	0.02%	0.00%	0.00%	0.00%
rtc_0 (gptp_rtc_0_0)	<0.01%	0.12%	0.00%	0.00%	0.17%	<0.01%	0.00%	<0.01%	0.00%	0.00%	0.00%
xlconstant_0 (gptp_xlcon...	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
xlconstant_1 (gptp_xlcon...	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
u_ila_0 (u_ila_0_CV)	1.25%	1.12%	0.02%	0.00%	2.83%	1.01%	0.72%	0.61%	5.36%	0.00%	0.00%

Figure 37 – RTL diagram of project that uses the gPTP IP designed

In the same way the utilization reports were obtained, the timing report exhibited in Fig. 38 was also acquired from Vivado. It shows that there is no failing endpoint in the design, i.e. that there is enough time for the signals to go through their paths. It is important to highlight here the timing impact of when there are many flip-flops (or other logical units) in a signal path. The strategy of making calculation requests for the PS to perform the computations and getting the results back from it was not pointless. If those computations were performed in the FPGA itself, timing would simply not be met due to the amount of logic resources required in the signals path.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.586 ns	Worst Hold Slack (WHS):	0.017 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	42254	Total Number of Endpoints:	42254	Total Number of Endpoints:	18081

All user specified timing constraints are met.

Figure 38 – RTL diagram of project that uses the gPTP IP designed

6.2 TESTING METHODOLOGY AND RESULTS

In order to validate the gPTP IP, it is necessary to test it as a component of a larger design, such the one described in Fig. 33. In the last section, it was already verified that the hardware utilization and and timing requirements are met. Now, we need to verify the design behavior, i.e. if it works as expected. For this reason, a simulation is performed for obtaining a proof of concept. Then, tests in utilizing the Zync SoC hardware take place.

6.2.1 Simulation Tests

The same software used for the design development, Vivado, also provides a simulation tool that is very useful for verifying the system behavior. Basically, a test bench is written in hardware description language to simulate all input signals of a design, while its output signals waveforms are observed.

The design in Fig. 33 corresponds to a node in an AVB network. To test this node behavior, especially regarding the gPTP protocol operation, it is necessary to simulate an AVB network with, at least, two nodes, so one of them is the slave-only end station designed and the other is a grandmaster. Moreover, it is also necessary to simulate the MAC layer through which the nodes communicate to each other. Then, the intended simulation must minimally have the designed end station node (that is the unit under test), a grandmaster node and something that works as a MAC layer. Observe a diagram of this scenario in Fig. 39.

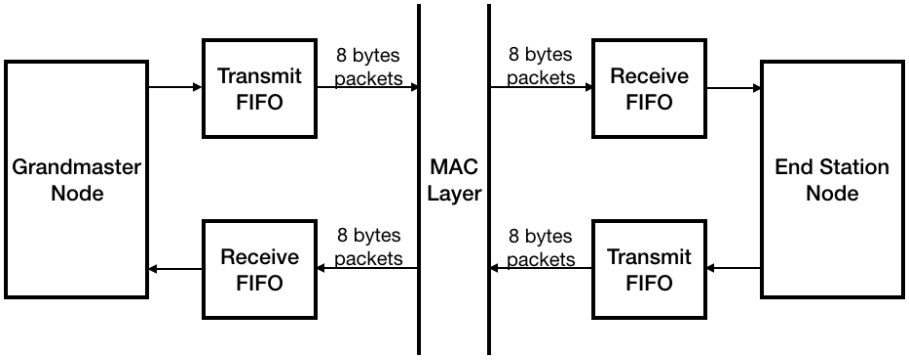

Figure 39 – Testing scenario

The MAC layer represented in Fig. 39 has the only objective of exchanging packets from one node to the other. Since each node has two FIFOs as interface, it is possible to abstract the MAC layer by transferring packets from the grandmaster node transmit FIFO directly to the end station node receive FIFO and also from the end station node transmit FIFO to the grandmaster node receive FIFO. Just as described in Fig. 40. However, the existence of two FIFOs simply forwarding packets to each other does not make much sense. It is just a waste of resources. The MAC layer can be better abstracted by keeping

only two FIFOs between the end station and the grandmaster node, as exhibited in Fig. 41.

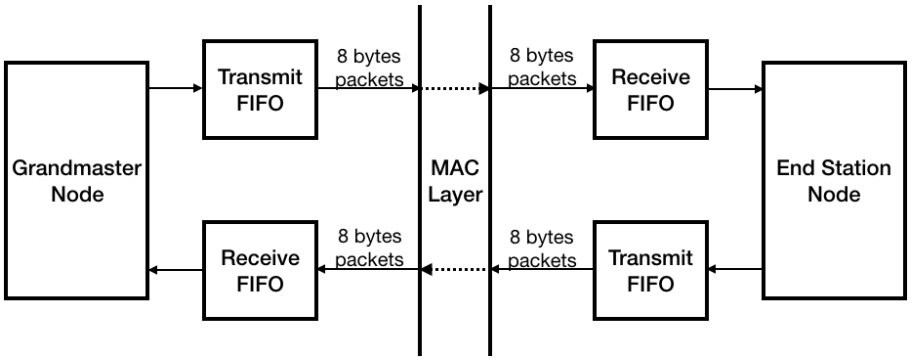

Figure 40 – Testing scenario

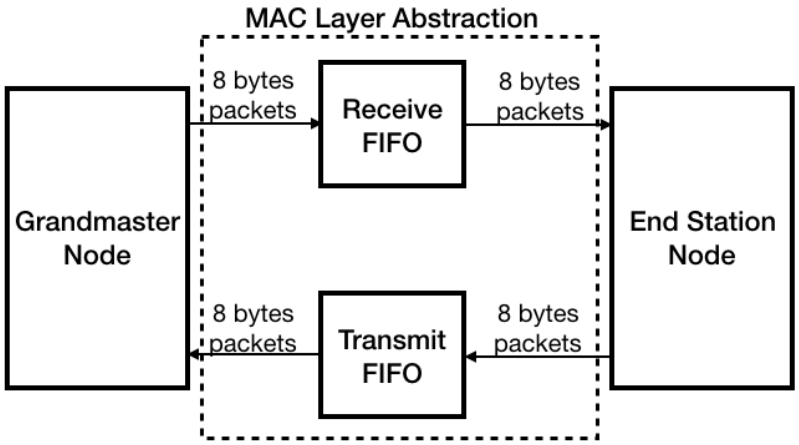

Figure 41 – Testing scenario

In Fig. 42, the end station node that corresponds to gPTP IP is represented along with the modules that compound it: Build Messages, Build Frames and the gPTP Core. All of them, already explained in detail. These modules and their interaction with each other are exactly our object of study. They represent the final product of this dissertation and thus need to be verified for their perfect functioning. On the other hand, the grandmaster node is created only for testing the end station designed and has the same architecture of the end station with a Build Messages, a Build Frames and a gPTP Core modules. Besides these, however, it also has an extra module, called Message Generator, which periodically creates and transmits signaling, *sync* and *follow up* messages.

The Message Generator module ensures the end station node receives *signaling*, *sync* and *follow up* messages, so the gPTP Core behavior upon the receipt of these messages can be observed. Moreover, the end station itself periodically sends to the grandmaster node *peer delay request* messages. The behavior of gPTP Core within the grandmaster is then observed in order to verify that the *peer delay response* and *peer delay response follow up* messages are correctly transmitted back to the end station. If this is the case,

the end station behavior upon the receipt of these two types of messages can also be observed. Therefore, in this way, it is possible to verify the behavior of all state machines in the gPTP core as well as of the Build Messages and Build Frames modules.

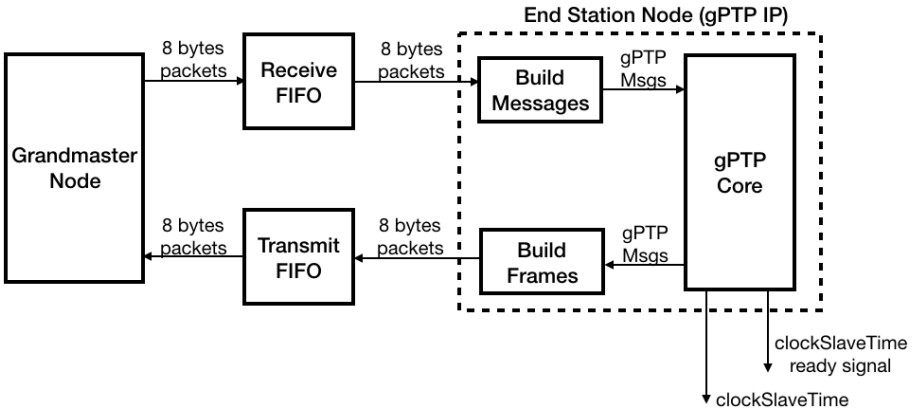

Figure 42 – Testing scenario

In this way, the simulation used to validate the gPTP IP for the end station node is expected to verify the points below. Note that the numbers 2, 3 and 4 correspond to the peer delay mechanism; the numbers 5, 6 and 7 represent the transport of time-synchronization information and the number 8 is the computation needed for the actual node synchronization.

- Extraction of Messages with the Build Messages Module
- Functioning of the gPTP Core
    1. Receipt of Signaling Messages and Configuration of time periods
    2. Creation and Dispatch of Peer Delay Request Messages
    3. Receipt of Peer Delay Response and Peer Delay Response Follow Up Messages
    4. Computation of Neighbor Rate Ratio, Propagation Delay and asCapable
    5. Receipt of Sync and Follow Up messages
    6. Creation and Dispatch of Sync Receive Structure
    7. Creation and Dispatch of Port Sync Sync Structures
    8. Computation of clockSlaveTime and clockSlaveTime_ready outputs
- Encapsulation of Messages with the Build Frames Module

Before diving into the gPTP Core operation itself, lets first validate the extraction and encapsulation of gPTP messages along with its interaction with the receive and transmit FIFOs. In Fig. 43 it is possible to observe the transmission of a *peer delay request* message by the end point to the grandmaster and also the receipt of the correspondents *peer delay*

response and *peer delay response follow up* messages sent by the grandmaster to the end point.

The *txSendPDelayReq* signal is a flag that goes high whenever a *peer delay request* message is to be sent by the end station, which occurs periodically according with the period set in the LinkDelaySyncIntervalSetting state machine. When this happens, the Build Frames module gets the *peer delay request* message in the *txSendPDelayReqMessage* signal and creates a sequence of 8 bytes packets. This sequence corresponds to the message and is sent to the transmit FIFO by being written to the *transmit_ff_DataIn* signal (in blue) while this FIFO's write enable signal *transmit_ff_WriteEn* (in blue) is high. Since the transmit FIFO along with the receive FIFO are an abstraction of the MAC layer, as soon as the transmit FIFO holds data, this data is extracted by the Build Messages module in the grandmaster node. Then, it is transmitted from the transmit FIFO to the Build Messages module by the *transmit_ff_DataOut* signal (in red) while the *transmit_ff_ReadEn* (in red) is high.

In the opposite way, the *receive_ff_WriteEn* and *receive_ff_DataIn* (in red) in Fig. 43 represent the transmission of a *peer delay response* and a *peer delay response follow up* messages from the grandmaster node to the end point. As soon as the receive FIFO holds data, this data is extracted by the Build Messages module in the end point so the actual messages are built. Observe the 8 bytes packets received by the Build Messages module in the *receive_ff_DataOut* signal (in blue) while the *receive_ff_ReadEn* is high, and in the *rcvdPDelayRespMessage* and *rcvdPDelayR...wUpMessage* signals, the peer delay response and *peer delay response follow up* messages built. Note that, once these messages are built, the signals *rcvdPDelayResp* and *rcvdPDelayRespFollowUp* go to high to indicate the messages were already received and are ready to be used. In Fig. 44 it is possible to observe a zoom in from Fig. 43, so the 8 bytes packets that form the *peer delay response* message received are better displayed.

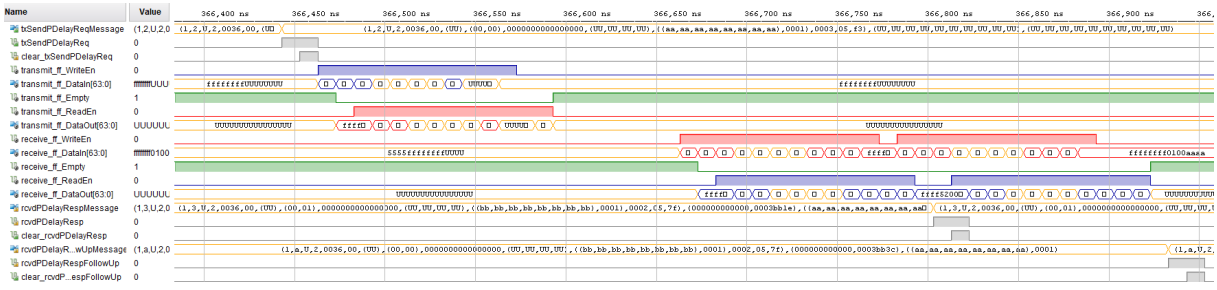

Figure 43 – Message Encapsulation and Extraction Waveform

In Fig. 45 it is possible to observe the operation of the *MD_PDelayRequest* state machine, responsible for the transmission of *peer delay request* messages and the receipt of *peer delay response* and *peer delay response follow up* messages. This state machine periodically builds *peer delay request* messages in the *txSendPDelayReqMessage* signal, and then writes a logical one to the *txSendPDelayReq* signal. This is done so the Build

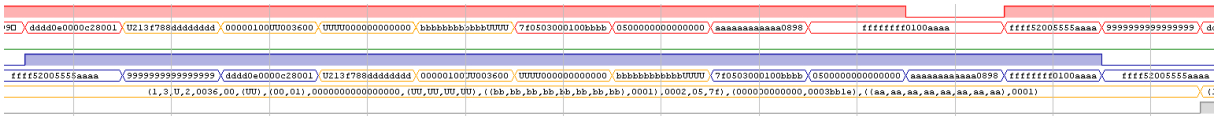

Figure 44 – Message Encapsulation and Extraction Waveform Zoom in

Frames module knows that the gPTP message is ready to be sent to the MAC layer. Also, since the *peer delay request* message is about to be sent, a timestamp t_1 (*timestamp1* signal in green) is taken to record the event time.

This process is repeated every time the *MD_PDelayRequest* state machine is in state s2 (*cur_state* signal in blue, not explicitly written, but between states s5 and s3). Then, the current state signal changes to s3 and stays there while the appropriated *peer delay response* message does not arrive. Upon its receipt, indicated by the pulse in the *rcvdPDelayResp* signal, the timestamp t_4 (*timestamp4* in green) is taken and the timestamp t_2 (*timestamp2* in green) is extracted from the *peer delay response* message in the *rcvdPDelayRespMessage* signal. After this, the current state changes to s4 and waits for the *peer delay response follow up* message to arrive, which is indicated by the pulse in the *rcvdPDelayRespFollowUp* signal. Then, the timestamp t_3 (*timestamp3* in green) is extracted from the arrived message in the *rcvdPDelayR...wUpMessage* signal.

Once all four timestamps t_1, t_2, t_3 and t_4 are acquired, the current state changes to s7 and a calculation request is made to the PS by sending a pulse in the *PS_calc_request* signal in red. The PS handles the required computations for the neighbor rate ratio and propagation delay, and sends the results back to the PL, while indicates they are ready by sending a pulse to the *PS_calc_ready* signal (in red). The computed propagation delay is then compared with a threshold, so the asCapable signal is set if the propagation delay is less than the threshold, or cleared otherwise. Observe, also in Fig. 45, the neighbor propagation delay (*D_PS*) and the *asCapable* signal (both in red).

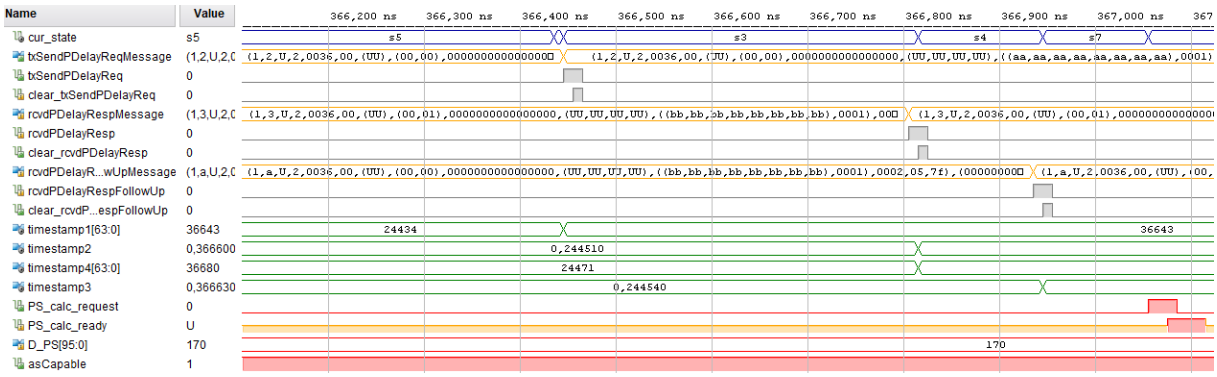

Figure 45 – *MD_PDelayRequest* state machine operation

At the grandmaster side, the *MD_PDelayResponse* state machine receives the *peer delay request* message sent by the end station and sends the *peer delay response* and *peer*

delay response follow up messages received from the end station, that are exhibited in Fig. 45. Observe in Fig. 46 that the receipt of the *peer delay request* message is signaled by the *rcvdPDelayReq* signal, while the transmission of the peer delay mechanism response messages are signaled by the *txPDelayResp* and *txPDelayRespFollowUp* signals. Note that the timestamps t_2 and t_3, received in the end station *MD_PDelayRequest* state machine, are taken in the grandmaster *MD_PDelayResponse* state machine, as exhibited by the *t2_pdelayReq_ingress* and *t3_pdelayResp_egress* signals (in red).

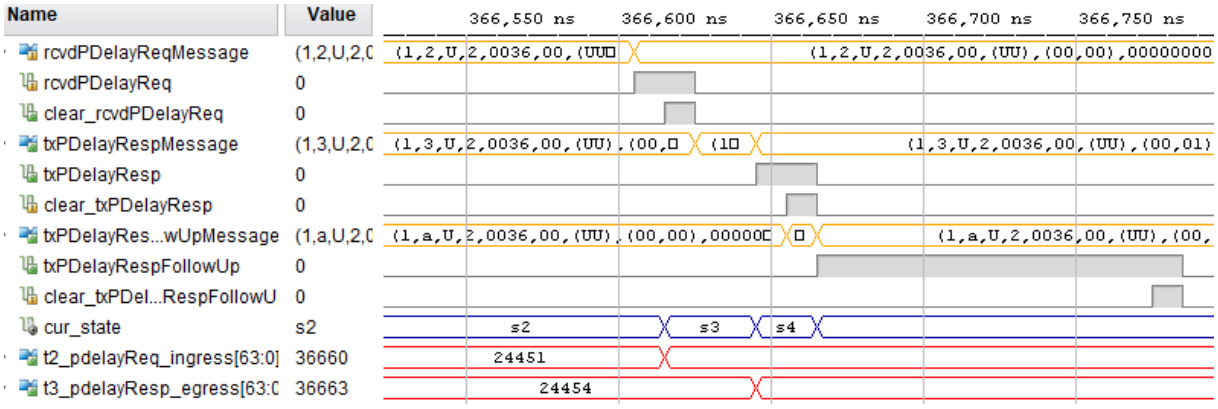

Figure 46 – *MD_PDelayResponse* state machine operation

This concludes the simulation tests of the peer delay mechanism operation, that corresponds to the numbers 2, 3 and 4 from the list of points to be verified in the implementation. Carrying on with the verification, the waveforms related to the transport of time-synchronization information and the computation of the clockSlaveTime and clockSlaveTime_ready signals, used to synchronize the node, are displayed in Fig. 47.

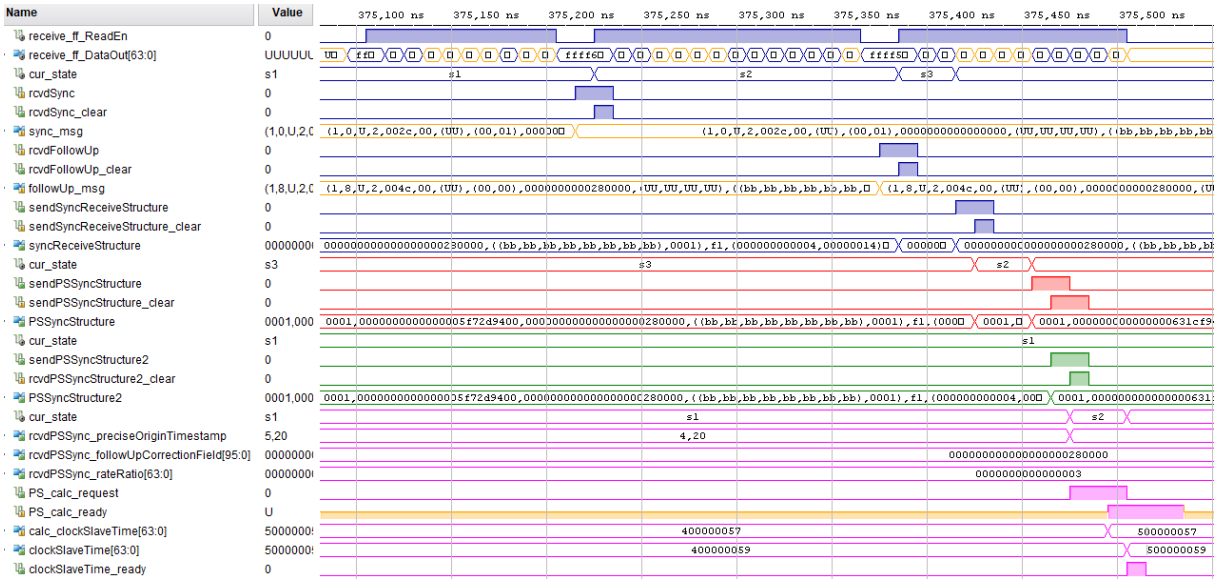

Figure 47 – Synchronization Process Waveforms

The pulses in signals *rcvdSync* and *rcvdFollowUp* (in blue) in Fig. 47 indicate the receipt of a sync and a follow up by the *MD_SyncReceive* state machine, which use the information carried in these messages to perform some calculations and build the Sync Receive Structure in signal *syncReceiveStructure*. As soon as this structure is built, the *sendSyncReceiveStructure* signal goes high to notify the *PortSyncSyncReceive* state machine, which then reads the structure and builds the correspondent Port Sync Sync Structure in the *PSSyncStructure* signal (in red). Once this other structure is ready, the *SiteSyncSync* state machine is notified by the *sendPSSyncStructure* signal (in red) and then receives it. This happens for each port in the end station, as the *SiteSyncSync* state machine receives a Port Sync Sync Structure from each port. The *SiteSyncSync* state machine then identifies which structure received came from the slave port, so it forwards this structure in the *PSSyncStructure2* (in green) signal to the *ClockSlave* state machine.

Upon the receipt of the Port Sync Sync Structure, the *ClockSlave* state machine extracts from it the *preciseOriginTimestamp*, the *followUpCorrectionField* and the *rateRatio* values in the signals *rcvdPSSync_preciseOriginTimestamp*, *rcvdPSSync_followUpCorrectionField* and *rcvdPSSync_rateRatio* (in magenta). Then, it sends a pulse to the *PS_calc_request* signal (in magenta) in order to request the PS to perform the needed computations for the clockSlaveTime output. When the results are ready, the PL is notified by the pulse in the *PS_calc_ready* signal (in magenta) and then clock cycles are added to the *calc_clockSlaveTime*, so the *clockSlaveTime* output compensates for the time spent by the PS in the calculations. Also, when the *clockSlaveTime* output is updated, a pulse is sent to the *clockSlaveTime_ready* output.

At this point, besides the Build Messages and Build Frames modules, the behavior of the *MD_PDelayRequest*, *MD_PDelayResponse*, *MD_SyncReceive*, *PortSyncSyncReceive*, *SiteSyncSync* and *ClockSlave* state machines in the gPTP Core were already verified by the simulation results. Despite the *PortSyncSyncSend* and *MD_SyncSend* state machines are implemented, they are not used in end station devices, so no simulation is done for them. Therefore, the only state machine left to be tested is the *LinkDelaySyncIntervalSetting*, responsible for configuring the periods in which *peer delay request* and *sync* messages are sent. In our case, only the former period does actually matter, since *sync* messages are not sent by end station devices.

The *LinkDelaySyncIntervalSetting* state machine operation is verified by the simulation waveforms in Fig. 48, which exhibits the receipt of a *Signaling* message in the *rcvdSignalingMsg1* signal (in blue). The state machine waits for a *signaling* message to arrive in state s2, then, upon its receipt, the *signalingMsg1_linkDelayInterval_temp* and *signalingMsg1_timeSyncInterval_temp* signals are updated with values carried in the *signaling* message. As their names indicate, these signals hold period values for the *peer delay request* and *sync* messages, respectively.

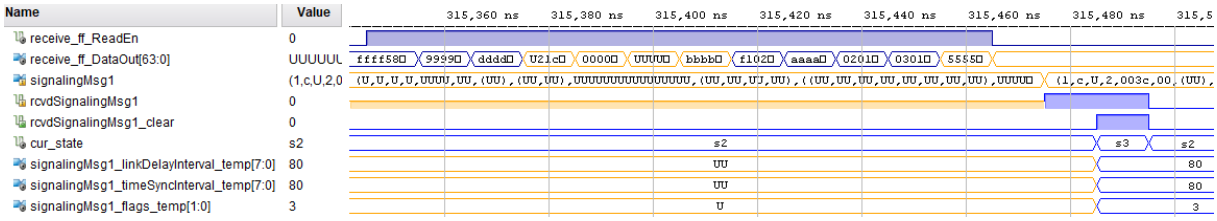

Figure 48 – *LinkDelaySyncIntervalSetting* Operation

6.2.2 Hardware Tests

Once the simulation results are analyzed and the behavior of the end station node regarding the gPTP protocol is successfully verified, we have a proof of concept that the design works. In order to categorically state that our implementation works and satisfies all requirements specified in the last chapter is to have it tested in a hardware platform.

For this purpose, we have available a prototype board that has the xc7z020clg400-1 part of the Xilinx Zynq chip, the same one used for acquiring the utilization and timing reports. Thus, a hardware bitstream for the design and the included ILA is generated in Vivado and deployed to the board's memory. Besides this, the board's flash memory is programmed with the processor C code using the Xilinx SDK software. At this point we have the design in Fig. 33, which corresponds to the slave-only AVB end station with only the gPTP protocol implemented, deployed to the actual hardware.

Ideally, two boards would be necessary for testing the gPTP implementation by communicating an end point with a grandmaster, just as done in the simulation. However, there was only one board available and other arrangements had to be made. We programmed the processor side of the SoC to, not only perform the calculations required by the gPTP IP state machines, but to also emulate an AVB node (regarding only the IEEE 802.1AS standard). The emulated node creates and sends gPTP messages directly to the receive FIFO in Fig. 33 design by sending the appropriated set of 8 bytes packets. In the same way, it also receives packets from the transmit FIFO in Fig. 33 design and converts them to gPTP messages, so these messages can be verified.

Following this approach, such as with the simulation, the gPTP IP behavior was tested upon the receipt of *peer delay request*, *peer delay response*, *peer delay response follow up*, *sync*, *follow up* and *signaling* messages. Besides this, the Build Messages and Build Frames modules were tested and verified to correctly encapsulate and extract every gPTP message, no matter its type. Finally, the clockSlaveTime and clockSlaveTime_ready outputs from the gPTP IP, that correspond to the actual time information used to synchronize the node to the grandmaster were also validated according with the expected values for the messages transmitted. Waveforms similar to the ones from the simulation were obtained with the ILA for the hardware tests, such that all results are successful and consistent. Furthermore, the required nanosecond precision was achieved, such that all time require-

ments defined in the IEEE 802.1AS standard were satisfied.

7 CONCLUSION

Today, the exchange of information between ECUs in cars typically uses the Controller Area Network (CAN) and the Controller Area Network with Flexible Data Rate (CAN FD). CAN and CAN FD protocols need to operate perfectly and in a reliable way, such that the communication between ECUs is not compromised. Moreover, in the context of a car connected to the exterior world, it is necessary to provide security mechanisms to prevent, or at least detect, cyber-attacks to the intra-vehicular networks.

This dissertation proposes a novel Error Injection Technique for CAN and CAN FD that enables the smash of any bit, from the first DLC bit (in CAN frames) or the BRS bit (in CAN FD frames), within frames from CAN/CAN FD networks using only a single device connected to the bus. By directly acting on the bus and adulterating specific bits, the tool provides efficiency and flexibility in customizing errors to be injected in the network. This mechanism can be very helpful for checking error conditions in real-world scenarios that affect the safety of vehicles. Thus, the tool can be a powerful ally for validating the correct functioning of automotive components and the safety requirements at the system level.

Then, this dissertation proposes a responsive intrusion detection system for CAN networks that can be deployed in low-cost hardwares, such as in a Raspberry Pi, and embedded in vehicles. The technique uses the one-class Support Vector Machine and the Isolation Forest algorithms to detect intrusion frames, possibly resulted from a cyber-attacks. Six models are defined by using these algorithms with different amount of the CAN frames data bytes as features. An experimental evaluation conducted and shows that the iForest models outperform the OCSVM models by achieving mean accuracy rates higher than 99% for the different amount of data bytes used as features. Moreover, it shows that the more data bytes are used as features the larger the accuracy rate mean and the smaller the accuracy rate standard deviation.

The rising of new and more complex automotive applications that require much higher bandwidths, such as ADAS and Infotainment systems, contributed for the emergence of the Automotive Ethernet. Despite many of these new applications required determinism, Ethernet was not originally conceived to support deterministic traffic. Thus, a few strategies to overcome the lack of determinism needed to be proposed.

The AVB standards were developed in order to provide time-synchronized streaming of audio and video using IEEE 802.3 Ethernet. One of these standards, the IEEE 802.1AS, defines the gPTP protocol that is responsible for synchronizing the nodes within an AVB network. Furthermore, the gPTP protocol can also be applied to others time-sensitive applications for synchronizing nodes. It can be used for controlling the wheels of an electric car, for example. Even though this protocol has a great importance and directly

affects a system's capacity of accomplishing synchronized activity, there are not many implementations of it available for the industry. Most are proprietary technologies that offer no implementation details and others do not offer the demanded nanosecond precision for some systems with strict timing requirements.

This dissertation provides a hardware implementation of the gPTP protocol utilizing a low-cost SoC platform. The implementation is tested with simulation and tests in hardware and satisfies the expected behavior for the protocol. The nanosecond precision specified by the IEEE 802.1AS standard, which is demanded for applications with strict timing requirements, is fulfilled. Moreover, implementation details are provided to help with future researches on the protocol.

Beyond the contributions and scope of this dissertation, a few future works are proposed. The combined use of the proposed Error Injection and IDS techniques for CAN networks is suggested, so they can work together as an intrusion prevention system capable of detecting and corrupting intrusion frames. The combination of this techniques has, however, rigid time requirements. In order to corrupt some intrusion frame, such that the ECUs would discard it, it is necessary to classify the frame as regular or intrusion fast enough so it is possible to corrupt an intrusion frame before the it is over. It is then necessary to measure the detection time for each frame and to ensure this time is lower than a established time limit. Moreover, the redesign of the IDS system using a lower level programming language, such as C, and also a Real Time Operating System (RTOS) could contribute for guaranteeing a deterministic detection behavior.

Furthermore, the details provided for the proposed gPTP implementation allows one to modify specifications and features of the protocol in order to investigate and propose enhancements to the IEEE 802.1AS standard. For instance, a work front currently investigates enhancements to the gPTP protocol that include grandmaster redundancy (P802..., 2018). Moreover, another work front currently investigates security on the IEEE 1588 standard (IEEE..., 2018). Since the IEEE 802.1AS includes a profile of the IEEE 1588, these security aspects could essentially be extended to the gPTP protocol. These investigations can make use of the proposed implementation and are left as suggestions of future works.

REFERENCES

AVNU FAQs. 2018. <http://avnu.org/faqs/>. [Online; accessed 04-June-2018].

BECHTEL, G.; GALE, B.; KICHERER, M.; DAVE, O. Automotive Ethernet AVB Functional and Interoperability Specification. In: AVNU ALLIANCETM. [S.l.], 2015.

BELLO, L. L. Novel trends in automotive networks: A perspective on Ethernet and the IEEE Audio Video Bridging. In: IEEE. *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. [S.l.], 2014. p. 1–8.

CHANDRAN, D.; GUDDETI, J.; SADASHIVAIAH, S. Automotive microcontroller interface protocol validation in post-silicon using on-the-fly error injector. In: IEEE. *Embedded Computing and System Design (ISED), 2016 Sixth International Symposium on*. [S.l.], 2016. p. 152–157.

CHARETTE, R. N. This car runs on code. *IEEE spectrum*, IEEE, v. 46, n. 3, p. 3, 2009.

CHECKOWAY, S.; MCCOY, D.; KANTOR, B.; ANDERSON, D.; SHACHAM, H.; SAVAGE, S.; KOSCHER, K.; CZESKIS, A.; ROESNER, F.; KOHNO, T. et al. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In: SAN FRANCISCO. *USENIX Security Symposium*. [S.l.], 2011.

D'AGOSTINO, R.; PEARSON, E. S. Tests for departure from normality. empirical results for the distributions of b^2 and $\sqrt{b^1}$. *Biometrika*, Oxford University Press, v. 60, n. 3, p. 613–622, 1973.

DEMŠAR, J. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, v. 7, n. Jan, p. 1–30, 2006.

FAROOQ, U.; MARRAKCHI, Z.; MEHREZ, H. *Tree-based Heterogeneous FPGA Architectures Application Specific Exploration and Optimization*. [S.l.]: Springer Science Business Media, 2012.

GARNER, G. M.; OUELLETTE, M.; TEENER, M. J. Using an IEEE 802.1 AS network as a distributed IEEE 1588 boundary, ordinary, or transparent clock. In: IEEE. *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2010 International IEEE Symposium on*. [S.l.], 2010. p. 109–115.

GARNER, G. M.; RYU, H. Synchronization of audio/video bridging networks using IEEE 802.1 AS. *IEEE Communications Magazine*, IEEE, v. 49, n. 2, 2011.

GESSNER, D.; BARRANCO, M.; BALLESTEROS, A.; PROENZA, J. sfican: A Star-Based Physical Fault-Injection Infrastructure for CAN Networks. *IEEE Transactions on Vehicular Technology*, IEEE, v. 63, n. 3, p. 1335–1349, 2014.

GMBH., R. B.; REIF, K.; DIETSCHE, K.-H. *Automotive handbook*. [S.l.]: Robert Bosch GmbH, 2014.

HANK, P.; MÜLLER, S.; VERMESAN, O.; KEYBUS, J. V. D. Automotive Ethernet: in-vehicle networking and smart mobility. In: EDA CONSORTIUM. *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.], 2013. p. 1735–1739.

HARTWICH, F.; BASSEMIR, A. The configuration of the CAN bit timing. In: *6th International CAN Conference*. [S.l.: s.n.], 1999. p. 2–4.

HERBER, C.; SAEED, A.; HERKERSDORF, A. Design and Evaluation of a Low-Latency AVB Ethernet Endpoint Based on ARM SoC. In: IEEE. *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*. [S.l.], 2015. p. 1128–1134.

IEEE 1588 Working Group Public. 2018. <https://ieee-sa.imeetcentral.com/1588public/FrontPage>. [Online; accessed 05-June-2018].

IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks. *IEEE Std 802.1AS-2011*, p. 1–292, March 2011.

IEEE Std 802.3bp-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE Std 802.3bw-2015, IEEE Std 802.3by-2016, and IEEE Std 802.3bq-2016) - IEEE Standard for Ethernet Amendment 4: Physical Layer Specifications and Management Parameters for 1 Gb/s Operation over a Single Twisted-Pair Copper Cable. *IEEE*, IEEE, 2016.

KANG, M.-J.; KANG, J.-W. A novel intrusion detection method using deep neural network for in-vehicle network security. In: IEEE. *Vehicular Technology Conference (VTC Spring), 2016 IEEE 83rd*. [S.l.], 2016. p. 1–5.

KIM, H. K. *Car-Hacking Dataset for the intrusion detection*. 2018. <https://sites.google.com/a/hksecurity.net/ocslab/Datasets/CAN-intrusion-dataset>. [Online; accessed 04-June-2018].

KOPETZ, H.; ADEMAJ, A.; GRILLINGER, P.; STEINHAMMER, K. The time-triggered Ethernet (TTE) design. In: IEEE. *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*. [S.l.], 2005. p. 22–33.

LEE, Y.; PARK, K. Meeting the real-time constraints with standard Ethernet in an in-vehicle network. In: IEEE. *Intelligent Vehicles Symposium (IV), 2013 IEEE*. [S.l.], 2013. p. 1313–1318.

LIM, H.-T.; HERRSCHER, D.; VÖLKER, L.; WALTL, M. J. Ieee 802.1 AS time synchronization in a switched Ethernet based in-car network. In: IEEE. *Vehicular Networking Conference (VNC), 2011 IEEE*. [S.l.], 2011. p. 147–154.

LIM, H.-T.; WECKEMANN, K.; HERRSCHER, D. Performance Study of an In-Car Switched Ethernet Network without Prioritization. In: SPRINGER. *Nets4Cars/Nets4Trains*. [S.l.], 2011. p. 165–175.

LIU, F. T.; TING, K. M.; ZHOU, Z.-H. Isolation forest. In: IEEE. *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. [S.l.], 2008. p. 413–422.

LIU, F. T.; TING, K. M.; ZHOU, Z.-H. Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, ACM, v. 6, n. 1, p. 3, 2012.

LIU, J.; ZHANG, S.; SUN, W.; SHI, Y. In-vehicle network attacks and countermeasures: Challenges and future directions. *IEEE Network*, IEEE, v. 31, n. 5, p. 50–58, 2017.

LUO, F.; MO, M.; LIU, C.; HUANG, Z. Can disturbances generator development. In: IEEE. *Vehicle Power and Propulsion Conference, 2009. VPPC'09. IEEE*. [S.l.], 2009. p. 1587–1591.

MAHMOOD, A.; EXEL, R.; SAUTER, T. Impact of hard-and software timestamping on clock synchronization performance over ieee 802.11. In: IEEE. *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*. [S.l.], 2014. p. 1–8.

MANN, E.; PEARSON, L.; ELDER, A.; HALL, C.; GUNTHER, C.; KOFTINOFF, J.; BUTTERWORTH, A.; UNDERWOOD, D. Avb Software Interfaces and Endpoint Architecture Guidelines. In: AVNU ALLIANCETM. *AVnu AllianceTM Best Practices, 2013*. [S.l.], 2013.

MARCHETTI, M.; STABILI, D. Anomaly detection of CAN bus messages through analysis of id sequences. In: IEEE. *Intelligent Vehicles Symposium (IV), 2017 IEEE*. [S.l.], 2017. p. 1577–1583.

MATHEUS, K.; KÖNIGSEDER, T. *Automotive Ethernet*. [S.l.]: Cambridge University Press, 2015.

MATHEUS, K.; KÖNIGSEDER, T. *Automotive Ethernet*. [S.l.]: Cambridge University Press, 2017.

MATSUMOTO, T.; HATA, M.; TANABE, M.; YOSHIOKA, K.; OISHI, K. A method of preventing unauthorized data transmission in Controller Area Network. In: IEEE. *Vehicular Technology Conference (VTC Spring), 2012 IEEE 75th*. [S.l.], 2012. p. 1–5.

MILLER, C.; VALASEK, C. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, Mandalay Bay, Las Vegas, NV, USA, v. 2015, 2015.

MOSTAFA, M.; SHALAN, M.; HAMMAD, S. Fpga-based low-level CAN protocol testing. In: IEEE. *System-on-Chip for Real-Time Applications, The 6th International Workshop on*. [S.l.], 2006. p. 185–188.

NAVET, N.; SIMONOT-LION, F. *In-vehicle communication networks-a historical perspective and review*. [S.l.], 2013.

NOVÁK, J. Flexible approach to the Controller Area Networks test and evaluation. In: IEEE. *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2009. IDAACS 2009. IEEE International Workshop on*. [S.l.], 2009. p. 44–48.

NOVAK, J.; FRIED, A.; VACEK, M. Can generator and error injector. In: IEEE. *Electronics, Circuits and Systems, 2002. 9th International Conference on*. [S.l.], 2002. v. 3, p. 967–970.

P802.1AS-REV – Timing and Synchronization for Time-Sensitive Applications. 2018. <https://1.ieee802.org/tsn/802-1as-rev/>. [Online; accessed 05-June-2018].

PIMENTEL, M. A.; CLIFTON, D. A.; CLIFTON, L.; TARASSENKO, L. A review of novelty detection. *Signal Processing*, Elsevier, v. 99, p. 215–249, 2014.

PRADEEP, Y. CAN-FD and Ethernet create fast reliable automotive data buses for the next decade. *Automotive Compilation*, v. 10, 2013.

ROBERT BOSCH GMBH. *CAN Specification version 2.0*. [S.l.], 1991.

ROBERT BOSCH GMBH. *CAN with Flexible Data-Rate Specification Version 1.0*. [S.l.], 2012.

SONG, H. M.; KIM, H. R.; KIM, H. K. Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In: IEEE. *Information Networking (ICOIN), 2016 International Conference on*. [S.l.], 2016. p. 63–68.

SRIDHARAN, K. *Investigation of time-synchronization over Ethernet In-Vehicle Networks for automotive applications*. Dissertação (Mestrado) — Technische Universiteit Eindhoven, the Netherlands, 2015.

STANDARD, I. *IEEE Std 802.3bw-2015 (Amendment to IEEE Std 802.3-2015, title=IEEE Standard for Ethernet Amendment 1: Physical Layer Specifications and Management Parameters for 100 Mb/s Operation over a Single Balanced Twisted Pair Cable (100BASE-T1)*, p. 1–88, March 2016.

STEINBACH, T.; KORF, F.; SCHMIDT, T. C. Comparing time-triggered Ethernet with FlexRay: An evaluation of competing approaches to real-time for in-vehicle networks. In: IEEE. *Factory Communication Systems (WFCS), 2010 8th IEEE International Workshop on*. [S.l.], 2010. p. 199–202.

TUOHY, S.; GLAVIN, M.; HUGHES, C.; JONES, E.; TRIVEDI, M.; KILMARTIN, L. Intra-vehicle networks: A review. *IEEE Transactions on Intelligent Transportation Systems*, IEEE, v. 16, n. 2, p. 534–545, 2015.

WEBER, M.; KLUG, S.; SAX, E.; ZIMMER, B. Embedded hybrid anomaly detection for automotive CAN communication. In: *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*. [S.l.: s.n.], 2018.

WOO, S.; JO, H. J.; KIM, I. S.; LEE, D. H. A practical security architecture for in-vehicle CAN-FD. *IEEE Transactions on Intelligent Transportation Systems*, IEEE, v. 17, n. 8, p. 2248–2261, 2016.

WOO, S.; JO, H. J.; LEE, D. H. A practical wireless attack on the connected car and security protocol for in-vehicle CAN. *IEEE Transactions on Intelligent Transportation Systems*, IEEE, v. 16, n. 2, p. 993–1006, 2015.

ZHANG, M.; XU, B.; GONG, J. An anomaly detection model based on one-class SVM to detect network intrusions. In: IEEE. *Mobile Ad-hoc and Sensor Networks (MSN), 2015 11th International Conference on*. [S.l.], 2015. p. 102–107.

ZINNER, H.; NOEBAUER, J.; GALLNER, T.; SEITZ, J.; WAAS, T. Application and realization of gateways between conventional automotive and IP/Ethernet-based networks. In: ACM. *Proceedings of the 48th Design Automation Conference*. [S.l.], 2011. p. 1–6.