Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
2019

António Luís Ferreira Marques

Métodos Eficientes para Recuperação de Erros no Domínio Temporal em Sistemas Flexible Time-Triggered

Efficient Time-Domain Error Recovery Methods for Flexible Time-Triggered Systems

António Luís Ferreira Marques

Métodos Eficientes para Recuperação de Erros no Domínio Temporal em Sistemas Flexible Time-Triggered

Efficient Time-Domain Error Recovery Methods for Flexible Time-Triggered Systems

Tese apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Doutor em Engenharia Eletrotécnica, realizada sob a orientação científica do Professor Doutor Paulo Bacelar Reis Pedreiras, Professor Auxiliar do Departamento de Electrónica, Telecomunicacões e Informática da Universidade de Aveiro e do Professor Doutor Luís Miguel Pinho de Almeida, Professor Associado do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Este trabalho foi apoiado por:
Fundação para a Ciência e a Tecnologia, que me concedeu uma Bolsa ao abrigo do Programa de Apoio à Formação Avançada de Docentes do Ensino Superior Politécnico (PROTEC) - ref. SFRH/PROTEC/49837/2009

Universidade de Aveiro, que me proporcionou as condições técnicas e humanas para a prossecução dos trabalhos realizados no âmbito desta tese.

Instituto de Telecomunicações de Aveiro, que apoiou financeiramente a minha participação em conferências internacionais para apresentação de resultados parciais obtidos no âmbito desta tese.

Instituto Superior de Engenharia de Coimbra/Coimbra Engineering Academy que me proporcionou as condições técnicas e humanas para a prossecução dos trabalhos realizados no âmbito desta tese, assim como apoio financeiro para apresentação de resultados parciais obtidos no âmbito desta tese.

Apoio financeiro da FCT e do FSE no âmbito do III Quadro Comunitário de Apoio.

o júri / the jury

presidente / president

Professor Doutor Vitor José Babau Torres
Professor Catedrático, Universidade de Aveiro
(por delegação do Reitor da Universidade de Aveiro)

vogais / examiners committee

Professor Doutor Petru Ion Eles
Professor Catedrático, Linkoping University

Professor Doutor Julián Proenza Arenas
Professor Associado, Universitat de les Illes Balears

Professor Doutor António Casimiro Ferreira da Costa
Professor Associado, Faculdade de Ciências da Universidade de Lisboa

Professor Doutor Valter Filipe Miranda Castelão da Silva
Professor Adjunto, Universidade de Aveiro

Professor Doutor Paulo Bacelar Reis Pedreiras
Professor Auxiliar, Universidade de Aveiro (**Orientador**)

agradecimentos / acknowledgements

This work would not be accomplished without the support and contribution of many people and institutions. To all of them, I would like to express my deepest gratitude.

First and foremost, I would like to thank my advisor Professor Paulo Pedreiras and co-advisor Professor Luís Almeida for their support, fruitful exchange of ideas, patience, friendship and constant encouragement over the years.

To the Coimbra Institute of Engineering in particular to the Department of Electrotechnical Engineering for the excellent work environment they provide me and for the motivation to finish this work. To my colleagues, especially those who worked in VEIL project - João Trovão, Paulo Pererinha, Marco Silva and Frederico Santos.

I want to thank my whole family for their love and support.

The last thoughts go to my beloved wife and daughter, who have always been my safe haven and who shared all the ups and downs of this long journey.

Palavras chave

Sistemas Distribuídos, CAN, Flexible Time-Triggered, Recuperação de Erros, Fiabilidade, Eficiência no uso de Largura de banda

Resumo

Atualmente, há um uso omnipresente de sistemas embebidos distribuídos, controlando todos os tipos de equipamentos, máquinas ou processos, que têm um impacto positivo no nosso dia-a-dia, e.g. carros, transportes públicos, equipamentos médicos e sistemas HVAC. Dependendo da natureza das tarefas executadas, juntamente com os requisitos de desempenho, podem ainda existir questões relativas à segurança que também devem ser tidas em consideração.

Devido à natureza distribuída desses sistemas, os nós de computação necessitam trocar mensagens de modo ao sistema executar as tarefas pretendidas. Isto é conseguido usando uma rede de comunicação subjacente, que implementa protocolos específicos para o tipo de tráfego que tem requisitos de pontualidade rigorosos, sendo estes herdadas dos requisitos de tempo real ao nível do sistema.

O ambiente em que a rede é implementada nunca está isento de interferências, por exemplo ruído devido a interferência eletromagnética. Estas podem corromper as mensagens, o que pode levar à degradação do desempenho ou até mesmo impedindo o funcionamento correto do sistema distribuído e, no limite, produzir resultados catastróficos no caso de sistemas de segurança crítica.

Claramente, este problema tem de ser resolvido, existindo diversos modos de o alcançar, geralmente através da utilização de métodos tolerantes a falhas. A tolerância a falhas pode ser obtida usando redundância temporal, redundância espacial ou uma combinação de ambas, onde a escolha depende de requisitos diversos, como por exemplo custo, consumo de energia, peso, espaço ocupado, complexidade ou tipo de falhas a serem toleradas.

Para lidar com sistemas de tempo real de segurança crítica, a escolha mais comum é utilizar redes baseadas no paradigma Time-Triggered, que usam um escalonamento das comunicações definidas na fase de projeto, pelo que são estáticas.

Estas redes permitem uma deteção rápida de erros, mas a latência na recuperação dos erros só pode ser minimizada pela reserva excessiva de largura de banda, transmitindo proactivamente várias instâncias da mesma mensagem, apresentando então um uso ineficiente da largura de banda disponível.

Uma alternativa possível é a utilização de redes baseadas no paradigma Flexible Time-Triggered (FTT), pois este usa escalonamento on-line do tráfego, possibilitando uma reacção rápida aos erros, integrando o escalonamento das retransmissões com o das mensagens regulares Time-Triggered, apresentando o potencial para usar significativamente menos largura de banda do que a opção Time-Triggered.

De facto, esta tese defende que é possível garantir níveis elevados de fiabilidade na transmissão de mensagens em redes Time-Triggered através da recuperação de erros no domínio temporal, utilizando um mecanismo de escalonamento das retransmissões controlado centralmente e dinâmico.

Para apoiar a tese, propõe-se um método direcionado para recuperar mensagens com transmissão falhada numa rede CAN Time-Triggered, com base no paradigma Flexible Time-Triggered, ou seja, FTT-CAN.

O mecanismo de recuperação de erros é executado no nó Master da rede FTT-CAN, ao qual deve ser adicionado um módulo detector de erros, um servidor para gestão de erros e a integração dos pedidos de retransmissão no escalonador on-line.

A escolha do servidor de erros, que implica a escolha de tipo, capacidade e período, é fundamental para atingir os objectivos de fiabilidade pretendida e também limitar a interferência em mensagens regulares Time-Triggered. No projeto deste, são considerados os cenários de erro de pior caso, que são limitados probabilisticamente por um processo de Poisson que modela a taxa de chegada de falhas.

A avaliação da proposta é feita por um estudo de simulação que utiliza diversos benchmarks de sistemas reais e outros gerados aleatoriamente, utilizando diferentes cenários de falhas, primeiramente utilizando um limite de uma falha por ciclo elementar e depois removendo essa restrição. Em qualquer caso, mostra uma redução de duas ordens de grandeza no valor médio da largura de banda utilizada obtido pelo mecanismo de recuperação de erros proposto, quando comparado com as abordagens tradicionais disponíveis na literatura e que são baseados na colocação de slots adicionais de transmissão definidos estaticamente.

A aplicabilidade da tese não se encontra restrita a redes FTT-CAN, pelo que é analisada a aplicação a outros protocolos Time-Triggered, em particular a TTCAN e FlexRay, mostrando melhor eficiência na utilização da largura de banda do que as versões Time-Triggered clássicas.

Keywords Distributed Systems, CAN, Flexible Time-Triggered, Error recovery, Reliability, Bandwidth Efficiency

Abstract

Nowadays there is an ubiquitous use of distributed embedded systems (DES), controlling all kinds of equipment, machinery or processes, that have a positive impact on our daily lives, e.g. cars, public transportation, medical equipment and HVAC systems. Depending on the nature of the tasks performed, along with performance requirements, there could be safety issues that must also be taken in consideration.

Due to the distributed nature of these systems, the computing nodes must exchange information/messages for the system to perform its intended function. They achieve this using an underlying communication network that implements specific protocols for the type of traffic that exhibits stringent timeliness characteristics, which are inherited from real-time requirements at the system level.

The environment where the network is deployed is never free of interferences, e.g. EMI noise. These can corrupt the messages, which leads to performance degradation or even preventing correct functioning of the corresponding distributed system and, in the limit, produce catastrophic results in case of safety critical systems.

Clearly this problem must be tackled, and there are diverse ways to achieve it, commonly through use of fault tolerant techniques. Fault tolerance can be obtained using temporal redundancy, spatial redundancy or a combination of both, and the choice depends on different requirements, as for instance cost, power, weight, space, complexity or type of faults to handle.

When dealing with safety-critical real time systems the most common choice is to use Time-Triggered networks, which use communication schedules defined at design-time, being therefore static. These networks present prompt error detection, but the latency in error recovery can only be minimized by bandwidth overprovisioning, proactively transmitting multiple instances of the same message, which makes it bandwidth inefficient.

A possible alternative is to use networks based on the Flexible Time-Triggered (FTT) paradigm, since it uses online traffic scheduling, which enables a prompt reaction to errors, integrating retransmissions among the regular Time-Triggered messages and having the potential to use significantly less bandwidth than the common Time-Triggered option.

In fact, the thesis states that it is possible to guarantee high-reliability of message transmission in Time-Triggered networks by performing error recovery in the time domain, using a centrally controlled and dynamically scheduled retransmission mechanism.

To support the thesis we propose a method tailored to recover failed message transmissions in a time-triggered Controller Area Network (CAN), based on the Flexible Time-Triggered paradigm, namely FTT-CAN. The error recovery mechanism will run in the FTT Master node, which must be enhanced with an error detector, a server for error management and integrating the retransmission requests in the online scheduler.

The error handling server design, implying the choice of type, capacity and period, is crucial for attaining the intended reliability goal and also limit the interference on regular Time-Triggered messages. In the design process the worst case error scenarios are considered, which are bounded probabilistically by a Poisson process that models the fault arrival rate.

The assessment of the proposal is done by an extensive simulation study that uses several practical benchmarks and randomly generated ones, within different fault scenarios, firstly using a limit of one fault per Elementary Cycle and afterwards removing this restriction. In any case it shows a reduction of two orders of magnitude in the average bandwidth taken by the proposed error recovery mechanism, when compared with traditional approaches available in the literature based on adding extra pre-defined transmission slots.

To show that the thesis is not solely applicable to FTT-CAN networks, the applicability to other Time-Triggered protocols is also discussed, in particular, to TTCAN and FlexRay, showing better bandwidth efficiency than pure Time-Triggered versions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The way we live and the well-being in the twenty first century society is dependent on automatic systems and machines, ranging from very simple ones, e.g. toasters, to complex ones, as for instance cars, airplanes or trains. The control of these systems depends heavily in computing devices, that often have a distributed nature, termed distributed embedded systems (DES). Then, the nodes of these systems need to exchange messages in order that they cooperate and the system can full-fill its intended function, being this accomplished by an underlying communication network.

Many of these systems have real-time constraints, so the top level requirements for system performance translate to requirements on the communication network, which must assure a reliable and timely delivery of the messages, to achieve correct system functioning. As interferences are always present in real systems then, at the network level, the existence of mechanisms to guarantee that the messages are timely delivered in the presence of faults, facilitates the implementation of fault tolerant mechanisms at system higher-levels.

1.1 Problem Statement

The communication network is subject to interferences, that can induce failed transmissions, which depending on critically level, can produce catastrophic failures at the system level, in economic terms or, in the limit, in human life's losses. To cope with the unavoidable faults, one (or more) fault-tolerant mechanism must be deployed, to guarantee adequate system performance and reliability levels.

A common way to implement networks in high-reliability DES is to use the Time-Triggered paradigm, where the message transmission is controlled by the passage of time, being transmitted in precise time instants. By using a clock with high precision, they present deterministic message transmit times and very low jitter. They typically use a static schedule, that is determined before the system startup, being therefore not operational flexible. Also, the integration of event-triggered traffic limits bandwidth efficiency severely, as an overprovision of reserved bandwidth/slots is necessary to attain prompt reaction to these events, which due to their nature can happen in any time instant.

The occurrence of errors happens in random instants of time, having then an event nature. Its recovery can be attained using spatial, time (or both) redundancy, having each one their questions relating the necessary resources and the efficiency in their use. In what concerns error-recovery using time redundancy, which is a way to guarantee message transmission reliability, the common approach is to use extra windows/slots, which are unused when there are no errors (most of the time), presenting then a low bandwidth efficiency. This limits severely the maximum utilization bandwidth, being this more notorious when the available bandwidth is already scarce for the current systems, e.g. CAN in automobiles, and is previewed an increase in future generation of DES.

This dissertation presents a method to obtain fault-tolerance in DES networks, which show very good bandwidth utilization efficiency in error recovery, in the time-domain, obtaining adequate level of message transmission reliability, with prompt error recovery and giving also guarantees on real-time timeliness.

The method is firstly presented for the FTT-CAN protocol and uses a module in the Master node that listens to the messages sent in each EC, detects failed messages, inserts this information in a error server queue, being the dynamic scheduling for the next cycle done jointly with the regular TT traffic.

The developed mechanism is afterwards generalized and shown that is applicable to other Time-Triggered protocols, namely TTCAN and FlexRay, still presenting considerable bandwidth efficiency gains.

1.2 Thesis and Contributions

The thesis supported by the present dissertation argues that:

It is possible to guarantee high-reliability of message transmission in Time-Triggered networks, combining an error detector with a server for online error management and message re-scheduling. This method results in small error recovery latency and a significant gain in bandwidth efficiency when compared to traditional time domain methods in common Time-Triggered networks.

The major contributions of this thesis are as follows:

- New method for error recovery in the time domain in FTT-CAN networks that allows meeting a desired reliability goal with small recovery latency and significantly lower associated bandwidth when compared to static pro-active retransmissions;
- Error injector and simulator for FTT-CAN networks, including error pattern generator;
- Optimizer for LSW parameter, considering multiple replica retransmission and global reliability target, in FTT-CAN networks;
- Application of same method to obtain high-reliability in TTCAN and FlexRay networks, using active time redundancy;

- Generalization of the method and abstraction of the network technology to obtain high-reliability in a time and bandwidth-efficient way in any Time-Triggered network that supports online traffic scheduling or a dynamic traffic phase.

1.3 Published works

The contributions in the scope of this dissertation, briefly presented in the previous section, were published in the following conference proceedings and journal article:

- L. Marques, V. Vasconcelos, P. Pedreiras, L. Almeida, and V. Silva. Towards Efficient Transient Fault Handling in Time-Triggered Systems. In *Proceedings of the INFORUM11 - Simpósio de Informática*, Coimbra, Portugal, September 2011.

- L. Marques, V. Vasconcelos, P. Pedreiras, and L. Almeida. Tolerating Transient Communication Faults with Online Traffic Scheduling. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT2012)*, pages 396-402, Athens, Greece, March 2012.

- L. Marques, V. Vasconcelos, P. Pedreiras, and L. Almeida. Error Recovery in Time-Triggered Communication Systems Using Servers. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES13)*, Porto, Portugal, June 2013.

- L. Marques, V. Vasconcelos, P. Pedreiras, and L. Almeida. Schedulability Analysis of Server-Based Error-Recovery Mechanisms for Time-Triggered Systems. In *Proceedings of the IEEE 18th Conference on Emerging Technologies and Factory Automation (ETFA'13)*, Cagliary, Italy, September 2013.

- L. Marques, V. Vasconcelos, P. Pedreiras, and L. Almeida. Comparing Scheduling Policies for a Message Transient Error Recovery Server in a Time-Triggered Setting. In *Proceedings of the IEEE Conference in Emerging Technology and Factory Automation (ETFA14)*, Barcelona, Spain, September 2014.

- L. Marques, V. Vasconcelos, P. Pedreiras, L. Almeida, and V. Silva. Efficient Transient Error Recovery in FlexRay Using The Dynamic Segment. In *Proceedings of the IEEE Conference in Emerging Technology and Factory Automation (ETFA14)* , Barcelona, Spain, September 2014.

- L. Marques, V. Vasconcelos, P. Pedreiras, and L. Almeida. Reactive Error Recovery in Time-Triggered Networks using Online Traffic Scheduling. In *Proceedings of Inforum 17*, Aveiro, October 2017.

- L. Marques, V. Vasconcelos, P. Pedreiras, and L. Almeida. Error Recovery in the Time-Triggered Paradigm with FTT-CAN. *Sensors Journal*, 18(1), January 2018.

1.4 Thesis Outline

The thesis supported by this dissertation is organized as follows:

Chapter 2 is a generic and diversified chapter, intended as a compact introduction to several topics that are fundamental to understand distributed and real-time systems, which will be used in this and later chapters. It starts with the communication paradigms, the physical and logic topologies and the types of medium access control available in communication protocols. Following, real-time systems are briefly characterized, along with scheduling policies and ways to evaluate the system schedulability, namely the response-time analysis. Servers definition and characterization and the fault models are also described in this chapter, along with the main aspects on dependability.

In what concerns protocols used in real-time systems, **Chapter 3** firstly describes thoroughly the CAN protocol, along with the new CAN FD version. Other protocols that are contending to dominate the future DES are presented next, including AFDX, TT-Ethernet, and also the ones that use the FTT paradigm - FTT-Ethernet, FTT-SE, HaRTES and FT4FTT. Protocols aiming at flexible Time-Triggered systems are afterwards presented. An in depth description of the FTT-CAN protocol, including all protocol details and the schedulability analysis for real-time functioning, referring also complementary works in FTT-CAN (e.g. multiple bus). The TTCAN and FlexRay protocols are presented here, as they will be used in Chapter 7, when the adaptation of our proposal to these protocols is presented.

In **Chapter 4** a detailed presentation of other works that use time redundancy is done, with a small introduction to spatial one. These works are the most relevant in what concerns error-recovery in Time-Triggered systems, that will be used later for comparison with the proposed method.

Chapter 5 describes the proposed error recovery method, firstly the single replica version, including the rational behind it, how it integrates failed transmissions in the scheduling process, how to obtain the server parameters and the response time of messages, along with their algorithms. To overcome the limitation on achievable transmission reliability when single replica is used, a second version is then presented, that essentially repeats the previous steps, but now defining new error and recovery scenarios, the number of replicas per error detected, with the ultimate goal of obtaining a global transmission reliability goal. The description of the optimization process to choose the LSW FTT-CAN key parameter finalizes this chapter.

Chapter 6 starts by presenting the simulator and the used benchmarks (*Updated_SAE, PSA* and VEIL) used in the simulation study. Correct functioning, server type choosing and server policies are assessed. The multiple replica version is also simulated using the same benchmarks and afterwards the chapter presents a comparison of the bandwidth efficiency of our proposal with the methods referenced in Chapter 4. This chapter concludes with an analysis and tests of some critical implementation issues in the Master node, namely the modifications necessary to implement the method proposed in this thesis.

Chapter 7 presents first the abstracted generic model of the proposed method, that

retains the essential characteristics. It follows an analysis on its applicability to two Time-Triggered protocols - TTCAN and FlexRay - discussing the adaptations necessary and presenting some practical considerations and values on the achievable bandwidth efficiency use.

Finally, **Chapter 8** concludes the document, resuming the dissertation, presenting the main conclusions and also pointing possible ways for future work.

In **Appendix A** a more detailed description of benchmarks *SAE*, *Updated_SAE*, *PSA* and *VEIL* are presented, followed by **Appendix B** with additional results from the simulation work. In **Appendix C** a proposal is presented that tries to detect and resolve inconsistent message omission scenarios in the Master node.

Chapter 2

Background

2.1 Real-time Systems

The control of all kind of physical functions is increasingly done by real-time systems, being these systems ubiquitous in so many areas of modern societies. These span from industrial process control, automotive, aviation or robotics, just to name a few.

A real-time system can be described as a computer system where the correctness of a computation is dependent on both the logical results of the computation and the time at which these results are produced [Kop11].

In real-time systems the timeliness of the results is mandatory, otherwise it is considered that a system failure occurred. If a result produced after its deadline has some utility to the system (no impact on the safety), the system is said to be soft real-time. On the other side, if a deadline miss can cause catastrophic consequences, then its called hard real-time system. So, when the consequences of system failure are severe or catastrophic, in terms of loss of value or even human lives, these are termed safety-critical systems.

2.1.1 Distributed Systems and Networks

A Distributed System (DS) is defined in [ST16] as "a collection of autonomous computing elements that appears to its users as a single coherent system". So a distributed system is composed by at least two computing elements, generally referred as **nodes**, where each one may behave independently of the others and, to the system user, are viewed as a single entity. So, the nodes need to cooperate to perform the intended function, doing that by exchanging messages through and underlying communication network. A Distributed Embedded System (DES) is a DS that has an embedded nature. They typically have real-time requirements, and are widespread in all fields and industries.

2.1.1.1 Physical Topologies

The network topology refers to the arrangement of the elements (e.g. nodes, links) of a communication network. The basic topologies, from the physical point of view, are repre-

sented in figure 2.1, being referred as Bus, Star, Ring and Mesh. Depending on deployment details, some distributed systems can use mixed topologies, that combines two or more of the basic topologies.

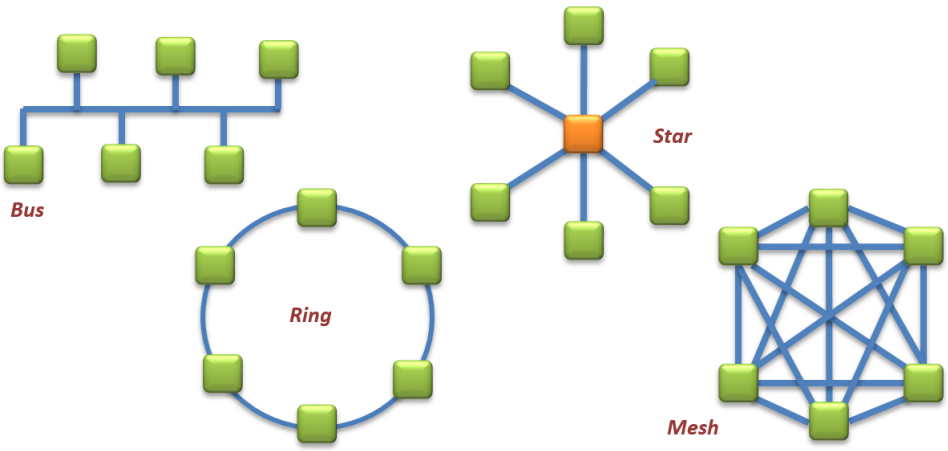

Figure 2.1: Physical topologies: Bus, Ring, Star and Mesh.

The bus uses a shared transmission medium, where all participants may listen all transmitted messages. This topology allows easy insertion and removal of nodes and lowers cabling length.

In the ring topology each message is passed from node to node, until reaching destination. In single rings the messages travels in one direction and with double ring messages can travel simultaneously in opposite directions. This can be used to increase throughput or to implement spatial redundancy.

The star topology uses a central equipment that has a dedicated link (half-duplex or duplex) to each node. This way, any communication between nodes always traverses the central equipment.

The used links are typically copper cables, the air or optical fibers. In this thesis the focus is in cabled networks and bus topology.

2.1.1.2 Medium Access Control - MAC

A key aspect, namely in topologies that use a shared transmission medium, is how they are allowed to transmit. The medium access control defines a set of rules that regulate when the nodes can transmit any pending messages. The most common are:

- **CSMA/CD** - Carrier Sense Multiple Access with Collision Detection - any node that finds the medium free can start transmitting, listening simultaneous to the electrical levels on the bus and if a collision is detected the node transmits a jam signal to guarantee that all nodes also detect signal corruption and stops transmission; afterwards it refrains from trying the transmission for a period of time (depends on specific implementation or protocol), and the process repeats again, until successful transmission; an example of a protocol that uses this MAC is (Shared) Ethernet [MB76];

- **CSMA/CR** or /CA (or /DCR)- Carrier Sense Multiple Access with Collision Resolution or Collision Avoidance (or Deterministic Collision Resolution) - the arbitration method consists in writing bit-by-bit the message ID and simultaneous listen to the resultant bit in the bus; then if the value is different from the written one, the node stops transmitting and the arbitration proceeds to the next bit with the nodes that have listen same value; this process repeats until only one node is a transmitter, that then proceeds and sends the message; an example of a protocol that uses this MAC is CAN [BG91], which will be detailed in Section 3.1;

- **TDMA** - Time-Division Multiple Access - it uses a global clock that permits each node to know the time instant where it is allowed to transmit; the time is divided in slots or windows, being each one assigned to a specific node; the sequence of pairs node/slot corresponds to the transmission schedule, that is fixed at system startup and repeats in cycles; as only one node has access to the medium in a particular time instant, the message transmission is done without collisions, being the transmission deterministic; examples of protocols that uses this MAC are TTP/C [KB03] and FlexRay (in the Static Segment) [Par07];

- **FTDMA** - Flexible TDMA - each node allocates one or more slots, referred as minislots due to the small duration (when compared with normal message) and when the slot counter of a particular node coincides with an allocated slot then it has permission to transmit the message, using the time of several minislots; examples of protocols that uses this MAC are Byteflight [BPG00] and FlexRay (in the Dynamic Segment).

2.1.2 Communication Paradigms

Messages, depending on the nature of their transmission instants, can be classified as [Obe05]:

- **periodic** - there is a constant time interval between successive message transmissions;

- **sporadic** - the transmission times are not known, but it is known that a minimum time interval exists between successive transmissions;

- **aperiodic** - the transmission times are not known nor a minimum time interval between successive transmissions exists.

In the **Time-Triggered** communication paradigm all the communication activities take place in precise time instants. Typically the time is divided in cycles and each cycle has a known number of windows or slots, being the schedule global and obtained off-line, there it is static. As advantages, it is deterministic and predictable, since that at any time instant it is known what message is being transmitted, which also facilitates the detection of transmission failures. Also the message jitter is minimum.

The integration of event-triggered traffic, e.g. alarms, is possible but poses efficiency limitations in bandwidth usage. This kind of traffic can be regarded as sporadic messages with a large average time between message transmissions, but much lower minimum time between messages. In this case the network slots to transmit these messages have to be configured for the minimum interval, but they will be empty most of the time, thus the low bandwidth efficiency. There are also some inflexibility to system evolution, as for instance when messages need to be added or removed. If flexibility in regards to add new messages (new system functionalities) is sought, then this must be previewed in the global schedule, adding spare slots to the schedule, that remains unused until the system is updated, which represents then wasted bandwidth.

Clock synchronization is mandatory, since each message must be transmitted in precise time instants. One good example of this paradigm implementation is the TTP/C protocol [KB03].

In the **Event-Triggered** paradigm, the messages are sent based on their occurrence and not in predefined instants in time. Being so, a mechanism must exist to resolve the conflict when two or more messages try to access the communication medium at the same time. Two well-known implementations are Ethernet [MB76] and CAN [BG91] protocol, for instance.

Due to the importance to support both types of traffic in DES networks more efficiently, protocols have emerged that divide the communication cycle in different segments or windows, as is for instance the TTCAN [LH02] and FlexRay [Fle05] protocols. Nevertheless, the time-triggered part is still static and defined offline. These protocols will be described further in Chapter 3.

The **Flexible Time-Triggered** paradigm tries to reconcile the previous paradigms where the event-triggered and time-triggered traffic share the available bandwidth. Bandwidth sharing between traffic type is done by splitting each cycle, termed elementary cycle, in two disjoint windows (or phases) - Synchronous and Asynchronous Window. Moreover, the time-triggered traffic is scheduled centrally in a special node, the Master, that sends a specific message, Trigger Message, that triggers the transmission of Time-Triggered messages for the current cycle. As the schedule is performed online, the scheduler can use any desired scheduling policy. Moreover, taking advantage of the dynamic scheduling, it is possible to change the messages and/or they characteristics, still guaranteeing message schedulability. This last characteristic, grants the protocol a operational flexibility that is not available in other Time-Triggered protocols, where the schedule is static.

The FTT paradigm has been implemented over CAN, the FTT-CAN protocol [APF02], shared Ethernet [PAG02] and switched Ethernet [Mar09], [San11]. The FTT-CAN protocol is presented in detail in Chapter 3, Section 3.3.2.

2.2 Fault Tolerance and Reliability

2.2.1 Dependability

In real-time systems the dependability requirements are of utmost importance, since failing to provide services in a timely and predictable manner may cause important economic losses or even put human life in risk.

Such systems must be dependable, i.e., it must be possible to place justifiable reliance on the service they deliver [Lap95].

As described by Aviziënis [ALRL04], **Dependability** is an integrating concept that includes what is described in the dependability tree (Figure 2.2).

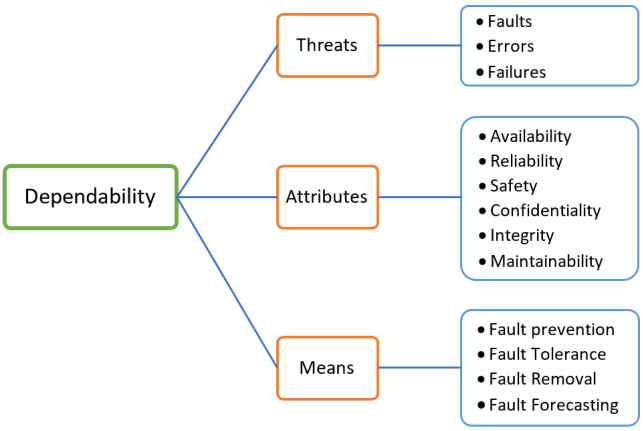

Figure 2.2: Dependability tree (adapted from [ALRL04]).

2.2.1.1 Threats

There is a causal sequence defined for the threats, defined as

... fault $\longrightarrow$ error $\longrightarrow$ failure ...

that can be defined at various-levels, as a failure in a low-level can constitute the fault for the next level.

Laprie in [Lap95] defines that a system **failure** occurs when the delivered service no longer complies with the specification, the later being an agreed description of the system's expected function and/or service. An **error** is that part of the system state that is liable to lead to subsequent failure; an error affecting the service is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a **fault**. This can also be put in the following form:

- **fault** - is a defect within the system (error cause);
- **error** - refers to difference between actual output and expected output;
- **failure** - it is the inability of a system or component to perform required function according to its specification.

Fault types

Faults can be classified according to their persistence in:

- **Permanent faults** - remain in the system until they are repaired; e.g., a broken wire or a software design error;
- **Transient faults** - starts at a particular time, remains in the system for some period and then disappears, e.g. EMI;
- **Intermittent faults** - are transient faults that occur from time to time, e.g. a hardware component that is heat sensitive, it works for a time, stops working, cools down and then starts to work again.

2.2.1.2 Attributes

Dependability includes the following attributes [Lap95]:

- **Reliability** - continuity of correct service;
- **Availability** - readiness for correct service;
- **Safety** - absence of catastrophic consequences on the user(s) and the environment;
- **Integrity** - absence of improper system alterations;
- **Maintainability** - ability for a process to undergo modifications and repairs;
- **Confidentiality** - absence of unauthorized disclosure of information.

More precisely **Reliability** can be defined as the probability that a system can perform its intended function, under given conditions, for a given time interval. In fact reliability comprises three aspects: hardware reliability, software reliability and communication reliability, being the system reliability obtained by multiplying these factors.

2.2.1.3 Means

Fault Prevention aims to prevent the introduction of faults, e.g., by constraining the design processes by means of rules.

Fault Tolerance aims to ensure that the presence of faults does not lead to system failure. Fault tolerance relies primarily on error detection and error correction, with the latter being either backward recovery (e.g., retry), forward recovery (e.g., exception handling) or compensation recovery (e.g., majority voting).

Fault Removal tolerates faults without compromising correct functioning.

Fault Forecasting aims to quantify the confidence that can be attributed to a system. Measures of the forecasted dependability can be obtained, for example, through stochastic modelling or through extrapolation of field experience from previously deployed systems.

2.2.1.3.1 Fault-Tolerance Fault tolerance [ALRL04] is intended to preserve the delivery of correct service in the presence of active faults. It is generally implemented by error detection and subsequent system recovery.

Redundancy types

In fact a fault tolerant system can be obtained including some sort of redundancy in the system. Redundancy can be classified in several types: spatial, time, informational (presentation, version). In all these types, redundancy does not necessarily mean identical functionality, but just performing the same work.

Redundancy can also be classified as static or dynamic. **Static redundancy** implements fault masking, meaning that the fault does not show up, since it is transparently removed. Examples are voting mechanisms, correcting codes, N-modular redundancy (NMR), (4-2) concept, TMR with duplex. In **Dynamic redundancy** after fault detection, the system is reconfigured to avoid a failure. Examples include backup sparing, duplex and share, pair and spare.

Hybrid approaches are also possible.

Redundancy has always a cost. For instance:

- Hardware: additional components, area, power consumption, shielding;
- Software: development costs, maintenance costs;
- Information: extra hardware for decoding / encoding;

2.2.1.4 Reliability in the message transmission subsystem

Let's concentrate our focus on the message transmission and how to obtain the reliability in the defined time period of working.

Message reliability in a mission time, MT, can be obtained by assessing the probability of success of sending each message in that period of time.

The message transmission can be regarded as a Bernoulli process, where each bit transmitted successfully has probability $(1 - ber)$ and fail probability ber, in an environment with known bit error rate ber. So, the success probability of sending one time the message with n_{bits}, can be obtained using Equation (2.1),

$$p_{success} = (1 - ber)^{n_{bits}} \tag{2.1}$$

and the corresponding failure probability of message i is

$$p_i = 1 - p_{success} = 1 - (1 - ber)^{n_{bits}} \tag{2.2}$$

Considering single-shot transmission and independent faults, then the success probability of sending all instances of a particular message, in the considered mission time - MT -, with

a total number of $\frac{MT}{T_i}$ messages sent, is obtained by the product of the success probabilities, as expressed in equation (2.3), where T_i represents the message i period.

$$p_{sucess}(MT) = \prod_{k=1}^{\frac{MT}{T_i}} (1 - p_i) = (1 - p_i)^{\frac{MT}{T_i}} \tag{2.3}$$

Extending this reasoning to all messages in the set, allows us to write an equation that reflects the transmission reliability of all N messages in the set, in the MT.

$$R_M = \prod_{i=1}^{N} (1 - p_i)^{\frac{MT}{T_i}} \tag{2.4}$$

Fault tolerant transmission

We are specifically interested in obtaining fault tolerance using time redundancy, by transmitting a predefined number of message replicas in the time domain, so the probability of a failure in message transmission, using m copies per message period, is given by Equation (2.5), where p_i as in Equation (2.2). Here, it is considered a successful transmission when at least one copy is delivered to the receiver or equivalently a message transmission failure occurs if and only if all instances of the message are not delivered.

$$p_i^{FT} = p_i \cdot p_i \cdot \ldots \cdot p_i = (p_i)^m \tag{2.5}$$

Then this equation corresponds to the new failure probability of message i with the fault-tolerant message set, obtaining finally Equation (2.6), simply substituting p_i by p_i^{FT}.

$$R_M^{FT} = \prod_{i=1}^{N} \left(1 - p_i^{FT}\right)^{\frac{MT}{T_i}} \tag{2.6}$$

As the R_M should be a value near unity, to have a better insight is preferable to use $(1 - R_M)$ instead, since is more "informative" to refer a value of $1 \cdot 10^{-6}$ of unreliability against referring a reliability equal to 0.999999. It is also easier to make comparisons this way.

2.3 Fault Models

2.3.1 Deterministic Model

A fault model for an event-triggered CAN network is presented in [TB94]. Firstly, during t seconds exactly one burst of errors with size n_{error} happens. Except for this burst, errors happen with period equal to T_S seconds, according to Equation (2.7).

$$tot_{errors}(t) = n_{error} + \lceil \frac{t}{t_{error}} \rceil - 1 \tag{2.7}$$

This allows to obtain the worst case response time of each message recovered and presents

an error recovery overhead that is directly proportional to the number of errors in the time interval t. This model is referred as deterministic since it is a model in which a bounded worst-case scenario is characterized.

In [PHN00] a generalization of this model is presented, that includes several interference sources,which using single source corresponds to the model presented in Tindell [TB94].

This is a simple model to use as it defines a concrete number of error per time interval, but gives pessimistic results as is normally used an upper bound for the error number.

2.3.2 Probabilistic Model

The logic behind probabilistic models is that the fault level can be low, with long inter-arrival times (most of the time) but sometimes the system experiences much higher loads with much lower inter-arrival time. A random distribution, like the Poisson distribution, as in Equation (2.8), seems to be a good match for the model. This distribution presents an average equal to $1/\lambda$ arrivals per second, where smaller distance between arrivals can occur, but with a much lower probability, so they occur less frequently.

$$P_\lambda(k;\tau) = e^{-\lambda\tau}\frac{(\lambda\tau)^k}{k!} \tag{2.8}$$

Navet in [NSS00] presents a probabilistic fault model that considers both fault frequency and gravity. It uses a generalized Poisson process where the faults can be single-bit faults or a burst (more than one single bit sequence) according to a random distribution. Then deadline failure probabilities are then computed.

Navas in [BBRN02] also use a Poisson distribution for the fault arrival, providing worst case response times for message frames, not as a single value, but as a probability distribution. The probabilities are obtained using a probability tree of scenarios that are pruned when branch probabilities are below a threshold. Due to computational cost, in [BBRN04], an enhanced version was presented, and also a comparison between CAN and TTCAN was performed, analyzing the probability of successful delivery.

2.3.3 Experimental BER characterization in CAN networks

The work reported in [FAFF04] have characterized the BER in diverse types of environments. Measures were performed in a factory with arc-welding machines two meters away (Agressive environment). The Normal environment was a factory production line, and lastly the Benign environment a laboratory at the University of Aveiro. The experimental setup used a CAN network configured with a bit rate of 1000 kbps.

The values reported are the ones presented in Table 2.1 that will be used in this dissertation to guide the parameter choosing in the proposed method.

Table 2.1: Bit Error Rate (BER) measurements in Controller Area Network (CAN)

Environment	BER
Benign	$3.0 \cdot 10^{-11}$
Normal	$3.1 \cdot 10^{-9}$
Agressive	$2.6 \cdot 10^{-7}$

2.4 Scheduling Algorithms

The process of choosing the next task to execute is termed scheduling. There are diverse ways to make these choices, which depends on particular characteristics of the tasks or the relation between them. In the following description it is assumed that the tasks are preemptive and the time taken to exchange tasks is negligible. The task set is given by Equation (2.9).

$$\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\} \tag{2.9}$$

where each task is characterized by its Period, Deadline, Offset and Execution Time - T_i, D_i, O_i and C_i.

Depending on the task periodicity, in real-time systems we can define three tasks types. The **periodic task** has many instances or iterations and a fixed period exists between two consecutive instances of the same task. A **sporadic task** has zero or more activations, and a minimum interval must elapse before other activation occurs. The **aperiodic task** can occur at any instant in time and the instant of next activation is not defined, with no known minimum interarrival value.

2.4.1 Scheduling Policies

One of the simplest algorithms is the **Fixed Priority (FP)**, where a priority is assigned to each message. If the priority is assigned according to the period, with the biggest priority attributed to the fastest task, than this policy is called **Rate Monotonic (RM)**. Using the task deadline, in an analog way to RM, the **Deadline Monotonic (DM)** policy is defined.

The **Earliest Deadline First (EDF)** assigns the priority according to the distance to the message deadline, implying that at each moment the task with the shortest distance to its deadline is the one chosen to be executed. **Least Laxity First (LLF)** is a dynamic priority scheduling algorithm that assigns priorities according to the laxity at any instant, the least laxity the greater the priority.

2.4.1.1 Shedulability Bounds

If all the deadlines are met then the task set is said to be schedulable. This can be evaluated using different techniques, as for instance bounds. These are based on task utilization, that is the ratio between execution time and period. In the FP case, Liu and Layland in their seminal work [LL73], proved that a safe bound to guarantee the scheduling of a task set using

RM policy is given by Equation (2.10). This equation is sufficient only. Another simple test for RM is the Hyperbolic Bound [BBB03], as given by Equation (2.11). Any of these tests can be used to rapidly evaluate the task set schedulability, offline as online, for instance as an acceptance test to accept new tasks. For DM policy, with $D_i \leq T_i$, the Equation (2.10) can be used substituting T_i by D_i, gives guarantees in schedulability.

$$U_{LL(RM)} = \sum_{i=1}^{n} \frac{C_i}{T_i} < n \cdot \left(2^{1/n} - 1\right) \tag{2.10}$$

$$U_{Hyper(RM)} = \prod_{i=1}^{n} \left(\frac{C_i}{T_i} + 1\right) \leq 2 \tag{2.11}$$

The *EDF* is a dynamic scheduling technique, being the bound given simply by Equation (2.12) [LL73].

$$U_{EDF} = \sum_{i=1}^{n} \frac{C_i}{T_i} < 1 \tag{2.12}$$

2.4.1.2 Response Time Analysis

Considering a system with a FP scheduling policy and n preemptive tasks, as defined by Equation (2.9), the response time can be obtained using Equation (2.13), where R_i is the response time of task i, T_k is the period of task k and $hp(i)$ is the set of tasks with higher priority than task i.

$$R_i = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil \cdot C_K \tag{2.13}$$

Since the term R_i appears in both sides of this equation, then an iterative resolution can be applied, as in Equation (2.14), since the R_i is monotonically non-decreasing. Applying this to each message, the process stops when $R_i^{m+1} = R_i^m$ or R_i^{m+1} is greater than the task deadline. In the latter case, the system is non-schedulable.

$$R_i^{m+1} = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i^m}{T_k} \right\rceil \cdot C_K \tag{2.14}$$

2.4.1.3 Processor Demand Test for EDF

The processor demand in interval $[0, L]$ is the total time needed for completing all jobs with deadlines no later than L, given by Equation (2.15) [But11].

$$C_p(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor \cdot C_i \tag{2.15}$$

To guarantee schedulability of all messages under EDF scheduling, then for all $L \geq 0$,

Equation (2.16) must be verified.

$$L \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{T_i} \right\rfloor \cdot C_i \tag{2.16}$$

2.5 Servers

Servers are software entities that act as proxies for associated aperiodic requests, shaping their arrival pattern and allowing for their integration in periodic/sporadic systems. Many server types can be found in the literature [But11], being typically characterized by a certain capacity C_S that can be provided over a given interval T_S to serve arriving requests. However, they differ in the rules on how and when their capacity can be used and replenished.

2.5.1 Polling Server

The Polling Server (PS) [But11] becomes active with periods equal to T_S and serves any aperiodic pending requests, with a maximum value equal to its capacity C_S. Any pending request must wait for next activation, so there is no aperiodic activity. Also, any unused capacity is lost. In terms of schedulability, its interference is bounded as a strictly periodic message of period T_S and execution time C_S.

A working example is given in Figure 2.3, which uses the task set of Table 2.2, having sporadic tasks arriving at instants 2, 9 and 12.5 which require 3, 1 and 1 time units of execution time. The server uses intermediate priority.

Table 2.2: Example for Polling Server

	T_i	C_i
τ_1	4	1
τ_2	8	2
PS	5	2

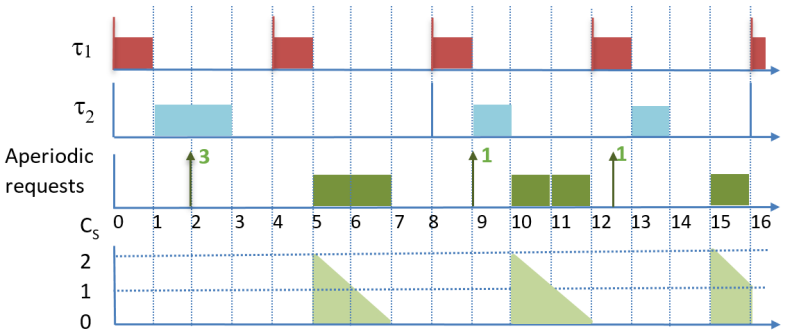

Figure 2.3: Polling Server (adapted from [But11]).

2.5.2 Deferrable Server

A Deferrable Server (DS) [LSS87, SLS95] replenishes its capacity strictly periodically and allows for consuming its remaining capacity at any point of its period. The server is marked as ready and scheduled whenever it has pending requests to serve and has enough capacity. The capacity is decremented by the exact amount of requested execution time that was actually served. Despite presenting a penalization in terms of the schedulability of lower-priority periodic tasks, when compared to other servers, like Polling and Sporadic Servers, the simplicity, small overhead, and responsiveness of DSs make them a good practical option as referred in [BB99]. Figure 2.4 presents the response of a DS, using the previous message set and server with equal T_S and C_S.

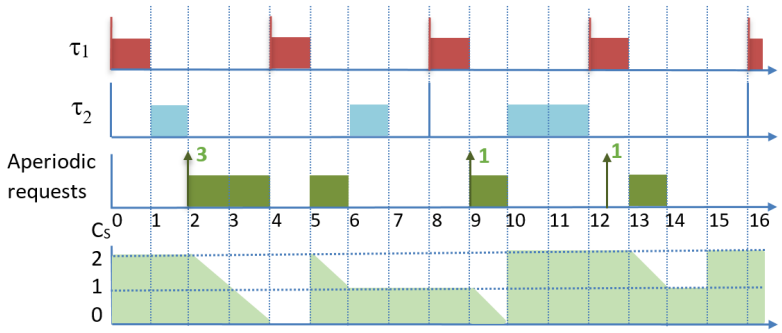

Figure 2.4: Deferrable Server (adapted from [But11]).

2.5.3 Sporadic Server

The Sporadic Server (SS), proposed in [SSL89], presents similar values in the average response time of aperiodic tasks when compared with the DS, but without the penalization in schedulability bound due to possible back-to-back execution that this last server presents.

The SS algorithm is a preserving capacity one that can use available capacity when need. The replenishment rule, forces that capacity replenished should be of equal value of the used one and should occur in T_S time. This way, the capacity is used and replenished in chunks, and from the schedulability point of view the server never uses more capacity than C_S in any considered period of time.

Table 2.3: Message set used in SS example

	T_i	C_i
τ_1	5	1
τ_2	15	4
SS	10	5

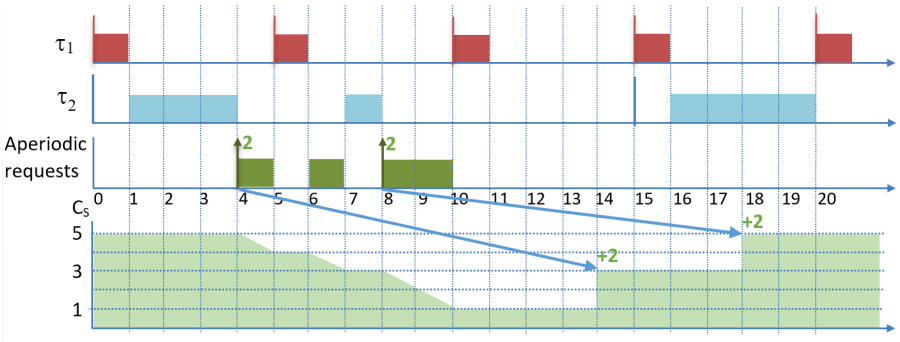

Figure 2.5: Sporadic Server (adapted from [But11]).

2.5.3.1 Summary of Servers

Table 2.4 is adapted from the one presented in [But11] that compares the various servers, according to the criteria presented there. First line corresponds to Background Service, which schedules the aperiodic or sporadic tasks only when there is no periodic task executing (uses only free time), presenting tipically long response times and no guarantees on schedulability of aperiodic messages that it serves.

Table 2.4: Fixed-Priority servers comparison

	Performance	*Computational Complexity*	*Memory Requirement*	*Implementation Complexity*
Background Service	☹	☺	☺	☺
Polling Server	☹	☺	☺	☺
Deferrable Server	😐	☺	☺	☺
Sporadic Server	😐	😐	😐	😐

A final remark to point out that similar algorithms exists for dynamic scheduling policies, e.g. the Dynamic Priority Exchange Server, Dynamic Sporadic Server or the Earliest Deadline Late Server, which are thoroughly described in reference [But11].

2.6 Summary

In this chapter general background information was presented that intended to give a briefly presentation on diverse topics related to real-time systems.

Chapter 3

Networks for Embedded Systems

This chapter starts by presenting in detail the CAN protocol, followed by diverse protocols based on Ethernet. Afterwards it presents in detail three protocols intended to obtain operational flexibility in Time-Triggered settings, namely TTCAN, FTT-CAN and FlexRay.

3.1 Controller Area Network

3.1.1 Introduction

With more than 1000 million CAN nodes installed in 2016, CAN is the dominant network in the auto industry[Zel17]. It is also used in medical equipment, in military vehicles, boats, planes and also in all types of industrial machinery. An example of a CAN network is depicted in Figure 3.1. CAN is still one of most used protocols, namely in the automotive industry, and functionalities that need more bandwidth can be tackled using CAN FD [BG12], prolonging this way the CAN protocol dominance in cars.

There are several options for the Application layer implementation, e.g. CANOpen [PAK08], [iA15], DeviceNet [Law13] or J1939 [Vos08], just to name a few.

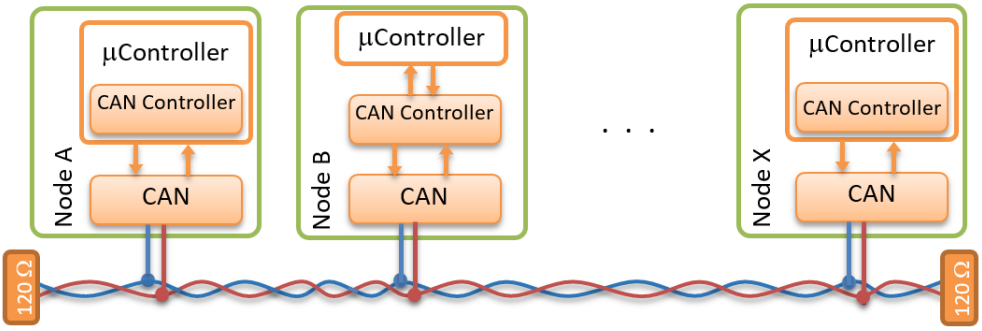

Figure 3.1: A CAN network.

The functions performed by the Physical and the Data-Link layers are listed in Table 3.2, and will be detailed in the following sections.

The Linear Bus as Physical Medium

Table 3.1: Relation between OSI model, ISO-11898 standard and implementation.

Data-Link	Logic Link Control		
	Medium Access Control	ISO 11898-1	CAN Controller
Physical	Physical Layer Signaling		
	Physical Medium Attachment	ISO 11898-2	CAN Transceiver
	Physical Medium Specification	ISO 11898-3	Bus (Physical Medium)
	Medium Dependent Interface		Connector
	OSI Model	*Standards*	*Implementation*

Table 3.2: Implemented function per layer

Data Link Layer	LLC Logic Link Control	Acceptance Filtering Overload Notification Recovery Management
	MAC Medium Access Control	Data Encapsulation/Decapsulation Frame Coding (Stuffing/Destuffing) Medium Access Management Error Detection/Signaling Acknowledgment Serialization/Deserialization
Physical Layer		Bit Encoding/Decoding Bit Timing/Synchronization Driver/Receiver Characteristics

The bus is defined in standard 11898-2 (High-speed Medium Access Unit) as a linear bus and uses a twisted pair with characteristic impedance equal to 120Ω, having a propagation speed of 4 to 5 ns/meter. As any electrical signal suffers reflections in the cable end, leading to signal distortions, the bus must be terminated by the cable characteristic impedance in a way to minimize these reflections. The termination resistor is placed in the bus ends and must be equal to 120Ω, as depicted in Figure 3.1

The accepted topology can present some variations, and can be a star, a twin star or a hybrid (mixed) topology, which are represented in Figure 3.2. These variations of the linear bus present additional limitations, namely on the maximum distance between nodes, the stub/dropline length or the allowed maximum bit rate.

3.1.2 Bus Signal Levels

CAN bus uses a differential signal to encode the logical levels that are termed Dominant and Recessive, corresponding to logical value 0 and 1, respectively. The CAN transceiver translates the logical level in two electric signals applied to the two bus wires, termed *CAN_Hi* and *CAN_Lo*. Applying a nominal voltage of 2.5V in both wires corresponds to the recessive level and the dominant level is obtained by applying a nominal voltage of 3.5V in CAN_Hi and 1.5V in CAN_Lo, that translates to a differential voltage equal to 2 V. These values are represented in Figure 3.3, including admissible limits for transmitter and receiver.

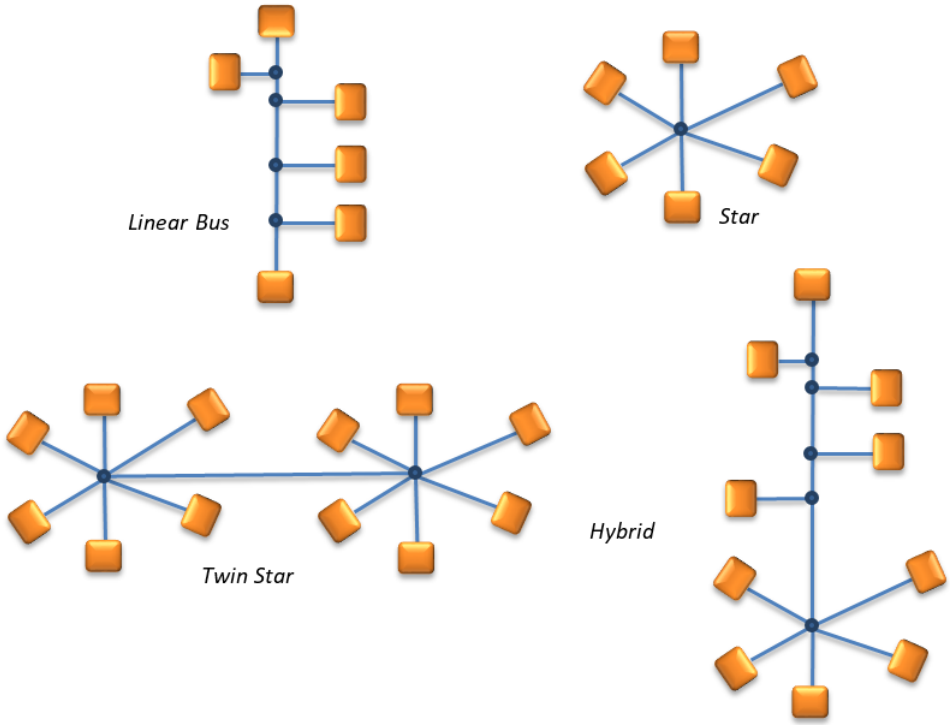

Figure 3.2: Possible physical topologies in CAN.

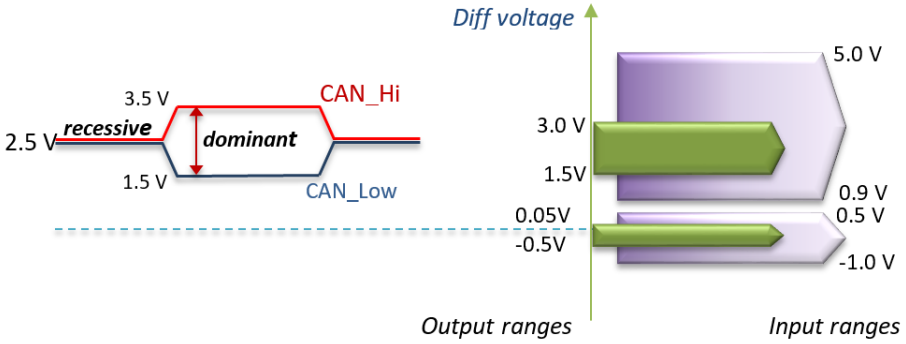

Figure 3.3: Voltage levels in CAN bus.

3.1.3 Bit time

The bit time is divided in four segments, as depicted in Figure 3.4. The temporal definition is measured in TQ (Time Quanta), being this value obtained dividing the controller clock period. The bit duration is then given in TQ. The segments are the following:

- *Synchronization Segment* - in this segment a transition is expected; duration is 1 TQ;
- *Propagation Time Segment* - this segment compensates the delays introduced by the signal propagation on the bus, delays in the input comparator and also in the output driver of the transceiver; the duration can be configured between 1 and 8 TQ;
- *Phase Segment 1* and *Phase Segment 2* compensates for phase errors, by adding TQ to the Phase Segment 1 or subtracting from the Phase Segment 2, to align the following transitions with the Synchronization Segment; the duration is between 1 and 8 TQ.

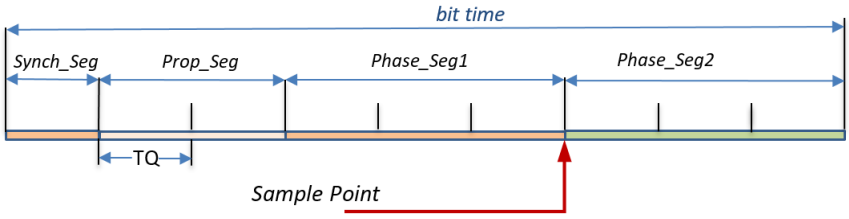

Figure 3.4: Bit time - division in segments.

As can be seen in Figure 3.4 the sample point is located between the last two segments. For instance, the microcontroller PIC18F2680 data-sheet [Inc07] refers that the optimum position of the sample point should be equal to 80% of the bit duration.

At the bit level, a hard synchronization can only occur once per frame and corresponds to the recessive to dominant edge transition of the SOF bit, where all the receivers will synchronize with the transmitter node (reset the bit time counter to the Synchronization Segment). Due to oscillator drift in the nodes, the bit clock varies and the nodes can lose the achieved synchronization. To maintain it, a resynchronization is performed in every recessive to dominant edge transition, adjusting the local bit time as necessary, based on the difference between the expected and the measured position of this transition edge (if this edge falls in the Synchronization segment, before or after).

3.1.3.1 Bit rate versus Bus length

When contending for bus access, each CAN controller must be able to read the bus level and compare it with the written value, to evaluate the outcome of current bit arbitration. This fact imposes a limitation on the maximum bus length for a defined bit rate (or nominal bit time).

The bus level is defined by the contributions of all nodes, so the sample point must be located inside the nominal bit time duration and must take into account all the delays

introduced by the hardware and transmission medium, as in Equation (3.1). Firstly we must consider the delay due to signal propagation on the bus t_{bus}, considering the greatest distance between any two nodes. Secondly t_{hw} accounts for delays introduced by the output and input stages of the controllers and also the transceiver delays in writing/reading the signal on the bus.

$$\tau_{delay} = t_{bus} + t_{hw} \tag{3.1}$$

In the bit time definition, the propagation segment duration must account the time necessary from the bit transmission start in one node to reach the most distant node plus the time necessary to the contribution of the second node to be sensed by the first one. So, this duration must be at the least twice the value of τ_{delay}, as in Equation (3.2).

$$t_{prop\_seg} \geq 2 \cdot \tau_{delay} \tag{3.2}$$

Due to this relation, for each bit rate value there is a maximum bus length that can be used, as illustrated in Table 3.3 [DNGG12]. We can observe that the bus can be several km long for very low bit rate (2.5 km for 20 kbps) but for 1 Mbps the bus length can only be 25 meters. The values presented in this table are somehow conservative, as for instance by choosing superior quality hardware it is possible to use a longer bus for a specified bit rate, e.g. 40 meters at 1 Mbps [Par07].

Table 3.3: Relation bit rate vs bus length

bit rate (kbps)	bit time (μs)	Maximum bus length (m)
20	50	2500
62.5	16	1000
125	8	500
250	4	250
500	2	100
1000	1	25

3.1.4 CAN Data-Link Layer

The CAN standard defines the following frames, according to their content and aim: Data, Remote, Error and Overload.

3.1.4.1 CAN Data Frame

The standard data frame (2.0A, 11 bit identifier), is represented in Figure 3.5, and is divided as follows.

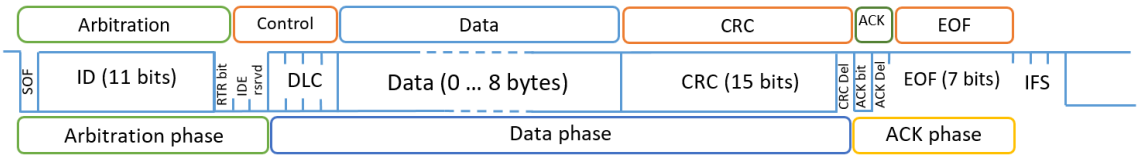

Figure 3.5: Standard data frame in CAN.

- *SOF* - any node that detects the medium free and has a pending message to transmit, try to send the frame by putting the bus in dominant level. If other nodes also possess messages to transmit, then this bit synchronizes the transmission process in all nodes;
- *CAN ID* or *ID Message Identifier* - it's an 11 bit number that identifies the frame, allowing a total of 2048 different identifiers. There cannot be two frames with the same ID. This value is utilized in the arbitration phase as a mean to control access to the bus. The node with the lowest ID frame will win the arbitration phase, proceeding with the frame transmission and others nodes switch to receiver mode (see section 3.1.5 for details).
- *RTR* - 1 bit set to dominant level;
- *IDE bit (Extended ID bit)* - bit set to dominant level for standard frame identifier (11 bits);
- *Reserved bit* - reserved for future use (see CAN FD section);
- *DLC (Data Length Code)* - 4 bits that represent the number of bytes in the Data Field;
- *Data Field* - this is the frame payload and can have between 0 and 8 bytes of data;
- *CRC Field* - this is a 15 bit field that contains the Cyclic Redundancy Check value that is calculated using the Arbitration Field, Control Field and Data Field. It is followed by a recessive bit termed CRC Delimiter;
- *ACK Field* - this is a 2 bit field; the first bit (ACK slot bit) is written with recessive level by the transmitter and any receiver that sees a correct frame should put the bus in dominant level and the second bit (ACK delimiter bit) must have a recessive level and is the ACK Field terminator;
- *EOF (End-Of-Frame)* - this field contains 7 bits, all with recessive level and ends the frame transmission.

After the EOF field and before the next frame transmission start there must be three recessive bits, named IFS (InterFrame Space) or Intermission, which must exist between any successful and successive frame transmissions.

CAN Data Frame 2.0B (29 bit identifier) The extended frame, represented in Figure 3.6, possesses the same base format as the standard one, being different in the following fields:

- *SRR (Substitute Remote Request)* - renames standard frame RTR bit; must have a recessive level;
- *IDE bit* - one bit with recessive value;
- *Extended ID* - 18 bits of length; with the first 11 bits of IDE (Base ID in the figure) constitutes the remaining bits of the 29 bits identifier;
- *R1, R0* - reserved for future use; must have dominant level.

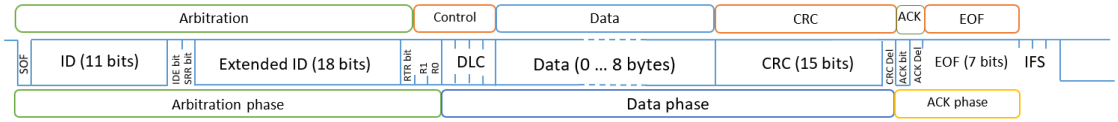

Figure 3.6: Extended data frame in CAN.

3.1.4.2 Remote Frame

This frame is used to request the transmission by other node of a specific data frame (with the same ID). The format is the same of the correspondent data frame, except for:

- The RTR bit must have a recessive level;
- There is no payload in the Data field.

In case of simultaneous transmission of the data frame and the remote frame there is no unresolvable arbitration conflict due to same ID number, because the RTR bit is also used in the arbitration mechanism and has different values for the two frames. Since the data frame has this bit with dominant level then this one is transmitted firstly.

3.1.4.3 Error Frame

When the transmitter node or any one of the receiving nodes detects an error they start, in the following bit, a transmission of an error frame, signaling and globalizing the detected transmission error. The error frame is composed by the Error Flag (6 dominant bits), and the Error Delimiter (8 recessive bits) that must be followed by the Interframe Space (3 recessive bits), as depicted in Figure 3.7. Other nodes that have not detected the original error will now detect the bit stuffing error imposed by the error flag and will also transmit their error flag, resulting in the superimposed error flag, that can last for another 6 bits, imposing an inaccessibility time of at most 23 bits.

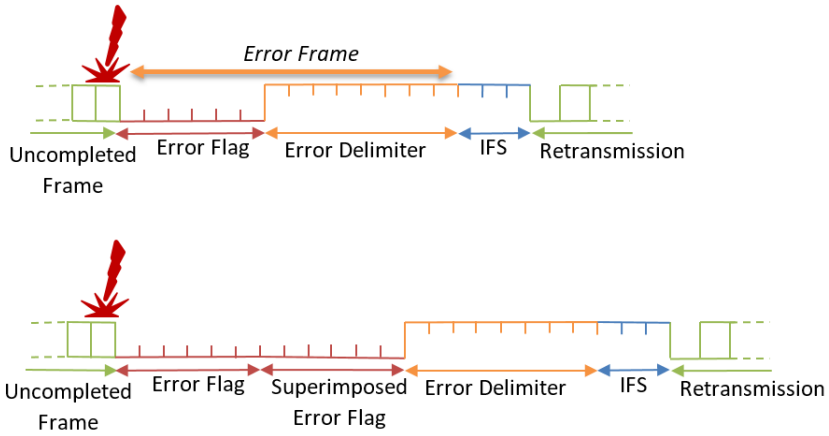

Figure 3.7: Active Error Frame, minimum size and with superposition of error flags.

3.1.4.4 Overload Frame

The overload frame has the same format as error frame, and the fields are termed overload flag (6 dominant bits) and overload delimiter (8 recessive bits). This frame is sent in the following situations:

- A dominant bit is detected during intermission period;
- Internal receiver conditions that needs extra time before processing next frame.

The overload frame is transmitted in the bit after the first situation is detected. When is the second situation that triggers the sending of the overload frame, the frame is transmitted in the first bit of the intermission time. After that, other nodes detect a dominant bit in the intermission time (overload condition) and also transmit their overload frame.

3.1.5 Non-destructive arbitration mechanism CSMA/CR

In CAN, the MAC (Medium Access Control) method uses a deterministic mechanism named CSMA/CR or CSMA/DCR (Carrier Sense Multiple Access with Collision Resolution or DCR- Deterministic Collision Resolution). In the arbitration phase, when two or more nodes are writing to the bus, the bus level is recessive only if all nodes have written this value and have a dominant level if at least one of them has written a dominant level. From the logical level point of view, we can consider the bus as an AND logical gate, where each node corresponds to an input, where the logical 0 corresponds to the dominant level and the logical 1 to the recessive level. The frame priority for bus access is defined by the CAN ID field, where a lower value stands for higher priority. We must remember that there are no two frames with the same ID, so each frame has a unique priority.

The arbitration phase starts when a node that has a pending message to transmit and encounters the bus idle, starts by issuing a dominant SOF bit. Any other node that has also a message to transmit synchronizes with it. Afterwards, each node writes its own bit and listens to the present bus bit level. If it is different from the one that he has written then he

knows that other message contending for the bus access has higher priority (dominant level) and then switches to receiver mode. This process continues until only one node considers himself the transmitter and afterwards transmits its message. An example of this process is depicted in Figure 3.8.

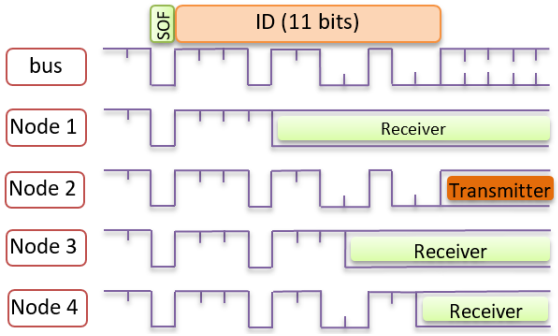

Figure 3.8: Example of arbitration process with 4 nodes.

In this example, after some node writes the SOF bit, all other nodes with pending transmissions will be synchronized. In the first ID bit all nodes write a recessive value, the bus has this value and they all read a recessive level and proceed to transmit the next bit. This happens again in ID bit 2 and 3. In the fourth ID bit, the nodes 2 to 4 write a dominant value and node 1 a recessive one, the bus level is dominant. Then, node 1 reads a dominant value, different from the one he has written and removes himself from the arbitration process and assumes from now on a listening mode. Nodes 2, 3 and 4 proceed by transmitting their fifth ID bit and by the same comparison process, on a bit-by-bit basis, a node that encounters a different level than the written one will withdraw from the arbitrating process (node 3 on the seventh ID bit and node 4 on the 10th ID bit). So, in the end, only one of the nodes wins the arbitrating process and will proceed by transmitting its frame.

3.1.6 Bit Stuffing mechanism

The line code used in CAN standard is NRZ (Non-Return to Zero), where the nominal bit time is simply the inverse of the bit rate. By using this code it is possible that a large number of bits with the same value will be transmitted on the bus and due to this fact the bit synchronism can be lost. To overcome this situation, the CAN standard mandates that whenever five consecutive bits with the same level exist in a sequence, a bit with different polarity should be placed in the following position, being this process named bit stuffing. In Figure 3.9 an example is presented, firstly for typical scenario where 2 stuff bits are inserted right after two sequences of 5 bits of the same polarity, marked in yellow. The third line shows the destuffing operation in the receiver, where the detection of 5 consecutive bits imply the removal of the following received bit. In (B) the worst case scenario is presented, where an initial inserted stuff bit imply additional insertion of stuff bits. This is the situation that gives rise to maximum bit stuffing in a CAN frame.

The frame bits subject to the bit stuffing mechanism are the ones presented in Figure

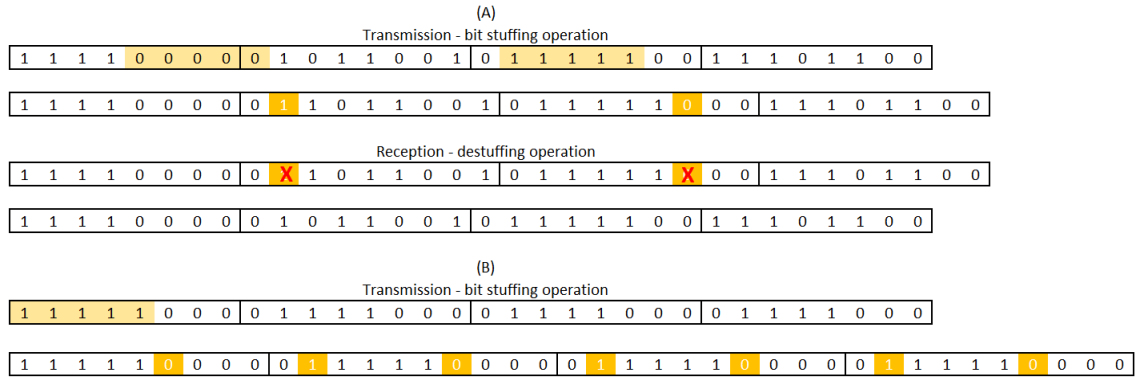

Figure 3.9: Bit stuffing mechanism

3.10, from the SOF bit until the last bit of the CRC. The number of transmitted bits in the bus depends on the bit pattern of each frame, bounded by a situation with zero stuff bits and the one with maximum stuff bits. The number of bits varies between 34 and 98 (34 control bits plus 0 to 8 bytes of data), so the maximum number of bits can be calculated using Equation (3.3), where DLC stands for the number of bytes in the Data field.

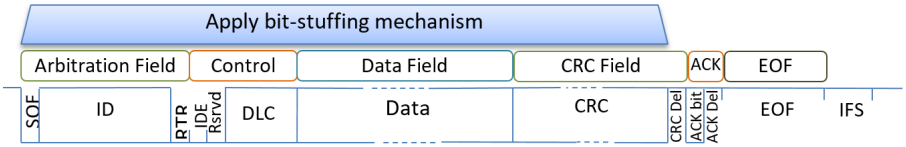

Figure 3.10: Bits subject to the bit stuffing mechanism (standard frame)

$$bits_{MAX} = g + 13 + 8 \cdot DLC + \left\lfloor \frac{g + 8 \cdot DLC - 1}{4} \right\rfloor \tag{3.3}$$

where g equals 34 for standard frames with 11 bits in ID field and 54 for the extended format (29 bits ID). Then, for the standard identifier, the total number of bits in a data frame with minimum stuff bits (meaning zero) and with maximum stuff bits are the ones presented in Table 3.4, where the values presented include the 3 bits of IFS [Nol03].

3.1.7 Error detection, signaling and recovery

CAN standard defines powerful mechanisms for error detection, used by the transmitter and receivers, as listed in Table 3.5. Error detection is followed by the transmission of an error flag that has a format that violates the bit stuffing rule (6 consecutive dominant bits), globalizing this way any error detected by at least one node.

The errors and scenarios that leads to it are described next:

Table 3.4: Number of bytes in standard CAN data frame

DLC	0 stuff bits	Maximum stuff bits
0	47	55
1	55	65
2	63	75
3	71	85
4	79	95
5	87	105
6	95	115
7	103	125
8	111	135

Table 3.5: Error types

Error detection mechanism	who ?
Bit monitoring	Sender
Acknowledgement	Sender
Cyclic Redundancy Check	Receiver
Bit Stuffing	Receiver
Form Error	Receiver

- *Bit Monitoring Error* - a transmitter always checks if each transmitted bit has the same value as the value read from the bus. If a difference is detected an error frame follows and the message is marked for retransmission. This mechanism must be disabled in the arbitration phase and also in the Acknowledgment slot;
- *Acknowledgment Error* - the sender writes a recessive bit in the Ack slot and expects that any node that receives correctly the message will re-write this bit with a dominant value. If the sender reads a recessive level an Acknowledgment error is detected, then it transmits an error frame and marks the message for retransmission;
- *Cyclic Redundancy Check Error* - the transmitter, using the bits from the frame beginning until the last bit of the data field, calculates a 15 bits CRC vector, which is transmitted right after the Data field. Each receiver, based on the same received bits also calculates the CRC vector using the same algorithm. The two CRC's are then compared and if any difference exists a CRC error is present. The nodes that detect this kind of error, transmit an error frame in the following bit;
- *Bit Stuffing Error* - the bit stuffing mechanism, previously described, guarantees that between the SOF and the CRC delimiter there are no more than five consecutive bits with the same level. So, whenever more than 5 bits with the same value are read, we say that a bit stuffing violation has occurred. Any node that detects this error, transmits an error frame;
- *Form Error* - a CAN frame has in certain positions a fixed level (recessive), which

are End-Of-Frame, Interframe Space, Acknowledge delimiter and CRC Delimiter. Any node detecting a dominant level in one of these bits has found a Form Error and will send immediately an Error Frame. A particular case occurs when the dominant level is detected in the last but one bit of the EOF field, which could lead to inconsistent error scenarios, which is presented in Section 3.1.10.

3.1.8 Fault confinement

Since the bus is a broadcast medium, a faulty node can interfere severely in the message exchange process and possibly disrupt the functioning of the whole system. The extreme case is known as the babbling idiot [BB03], where a node transmits frames continuously, making it impossible for other nodes to exchange any messages. The CAN standard defines a mechanism to prevent that faulty controllers degrade significantly the communications and does this by auto excluding nodes with error counters with values above specified thresholds. In what regards the error status, a node can be in one of three states: error-active, error-passive and bus-off. The controller state is determined by the values present in two error counters: TEC (Transmit Error Counter) and REC (Receive Error Counter), as depicted in Figure 3.11, which also details the transition thresholds between states.

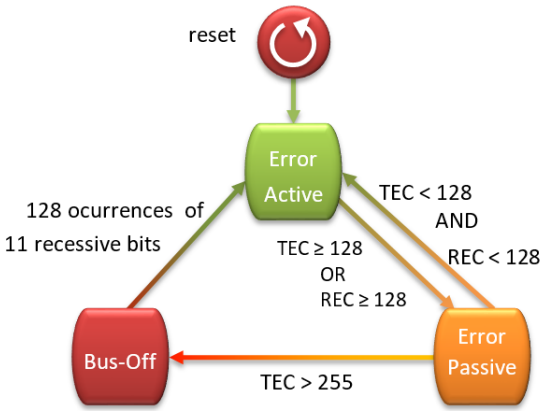

Figure 3.11: CAN controller - Error state machine.

In Error Active state, the controller is assumed to be fully functional, been allowed to transmit and participle actively in error detection and error signaling using the active error flag. This state is attained after a reset and whenever the REC and TEC counters both have a value less than 128. When suspect of faulty behavior the controller transits to the error-passive state, being still able to transmit frames but its error signaling capabilities are now restricted since it can only use passive error flags. The controller transits to this state if any of these counters have a value greater than 127 and less than 256. The controller continues to update the error counters and if both fall for a value less than 128 then it transits back to error active state. In the bus-off state the node is not allowed to transmit any type of frames, since considers itself already corrupted. It can only return to active error state after a reset or on receiving 128 consecutive correct frames. The rules used to increment/decrement the

counters update these counters in a way that their current value gives a good indication of the controller state-of-health. On successful frame transmission both error counters (TEC and REC) are decremented by 1 unit. An error detection and subsequent transmission of the error flag implies the increment by 8 in the sender. On the receiver side, the same situation implies that the REC counter is incremented by one. If the node that firstly detected the error sees a dominant level after the sixth bit of the error flag (due to secondary error flags by other nodes) then it acknowledges that by incrementing the REC counter by 8 units.

In error-passive status the transmission of passive error flag does not force other nodes to acknowledge that this node has detected an error, being this way a source of error detection inconsistency. Nevertheless, in the CAN controller a warning condition exists and a flag is set if any of the counters exceeds 96, which can be used to trigger measures at higher-levels.

3.1.9 Frame Filtering (Acceptance Filtering)

The CAN controller implements a mechanism that filters all the received frames, places the contents in the receiver buffers and signalize the reception to the application if the frame ID matches a specific ID or a set of IDs. This way, the microcontroller only has to process the messages of interest and not all messages sent in bus. This is especially important in many distributed systems that use small microcontrollers, which possess limited resources in terms of available memory and computing power.

3.1.10 Possible Inconsistent Scenarios

In normal operation, all nodes in the CAN bus, being it the transmitter or the receivers, share the same view of the message status - error free or with error. Nevertheless, in some particular scenarios this is not true.

From the transmitter point of view, a successful transmission happens if no error is detected until the last bit of EOF field. On the other hand, a receiver considers the frame correct and will accept it as valid if no error is detected until the last but one bit of EOF field. Figure 3.12, defines these bits. Because of this different validation rule used by transmitter and receiver there are some very particular scenarios that lead to inconsistent scenarios, being this flaw first presented by [RVA+98], being referred as Inconsistent Message Duplicate (IMD) and Inconsistent Message Omission (IMO).

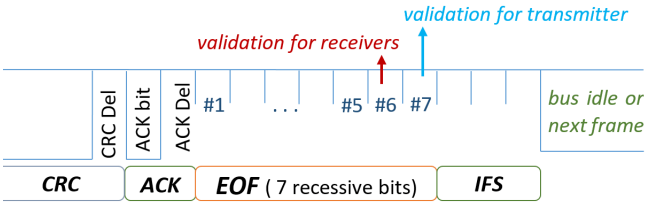

Figure 3.12: Last time consistency.

If an incorrect level, meaning a dominant one, is detected by a node in the EOF field (as this field is composed of 7 recessive bits), then an error frame is transmitted in the next bit

forcing all the nodes to reject the frame. This is achieved in a consistent way until the bit prior to the last but one bit. But if the last but one bit is perceived as a dominant level, e.g. induced by EMI, by one or more nodes then consistency on error detection and frame discarding is not guaranteed, being this a serious impairment to attain the desired reliability.

In Figure 3.13 the **A** set of receivers detect an incorrect level in the last but one bit of EOF, so they discard the frame and start error signalling in the last bit of EOF. The transmitter observing a dominant bit here (last bit) will mark the transmission as failed and will try to retransmit the message. The nodes pertaining to the **B** set already have accepted the frame (as they have not detected any incorrect bit value until last but one bit), interpreting a dominant bit in the last bit of EOF as an overload frame. After retransmission of the message, the nodes in set **B** will get two copies of the same message, being this described as Inconsistent Message Duplicate (IMD).

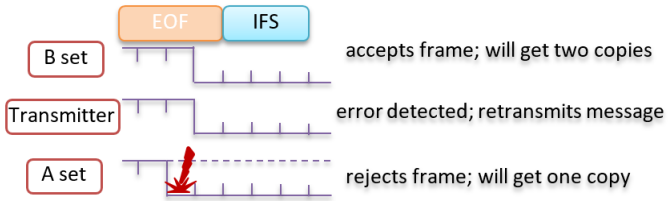

Figure 3.13: Scenario of Inconsistent Message Duplicate.

A different case occurs if before retransmission the transmitter crashes or suffers some malfunction that temporarily takes it out of work (that at least lasts beyond the message deadline). Then, as depicted in Figure 3.14, the nodes pertaining to the **A** set, that, as described in the previous situation, have already rejected the message, due to the transmitter not retransmitting the message, will not get it, being this an Inconsistent Message Omission (IMO).

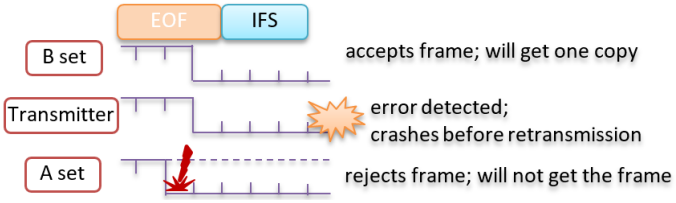

Figure 3.14: Scenario of Inconsistent Message Omission.

Using the same formulation as in [RVA+98], considering a failure rate for the microcontroller (including the CAN controller), λ_p equal to $10^{-5}/hour$ and the BER's for the various environments from [FAFF04] we obtain the values for IMD and IMO presented in Table 3.6.

In [RVA+98] the authors propose the fault-tolerant broadcasts protocols EDCAN (Eager Diffusion), RELCAN (Reliable Broadcast) and TOTCAN (Totally Ordered) to obtain consistency. All these protocols need extra-rounds of message exchanging to attain consistency, not being well suited to Time-Triggered settings.

Table 3.6: IMO's and IMD's in CAN, for the diverse ambients

Environment	BER	IMD/hour	IMO/hour
Benign	$3.0 \cdot 10^{-11}$	$8.60 \cdot 10^{-4}$	$1.19 \cdot 10^{-14}$
Normal	$3.1 \cdot 10^{-9}$	$8.89 \cdot 10^{-2}$	$1.23 \cdot 10^{-12}$
Agressive	$2.6 \cdot 10^{-7}$	$7.45 \cdot 10^{0}$	$1.04 \cdot 10^{-10}$

To resolve IMO, Kayser[KL99] proposes a solution based in hardware redundancy, described as the SHAdow REtransmitter or simply SHARE, where the dedicated hardware detects specific error patterns, the ones that leads to IMO, and if found it retransmits the frame. This solution is tailored to Event-Triggered networks.

Proenza [PMJ00], propose the MajorCAN protocol, which would resolve the consistency problem on a frame basis. However this solution does not comply with the standard protocol and its use would imply building specific controllers, which is not cost effective.

The IMD can be circumvented by using a message counter, so whenever the receiver gets a message with the same counter value it can safely discard it, as the pertinent information that the message carries is the same.

3.1.11 CAN with Flexible Data Rate - CAN FD

As bandwidth requirements in the automotive industry keep on increasing, Bosch in close cooperation with car makers and CAN specialists developed an improved CAN Data-Link protocol [BG12]. The major enhancements are an increased bit rate in the data phase and bigger payload, with a maximum of 64 bytes instead of only 8 of standard CAN.

In figure, taken from [Lin12], a complete CAN-FD frame transmission, with 4 nodes competing for bus access, that shows clearly the speedup used in the data phase.

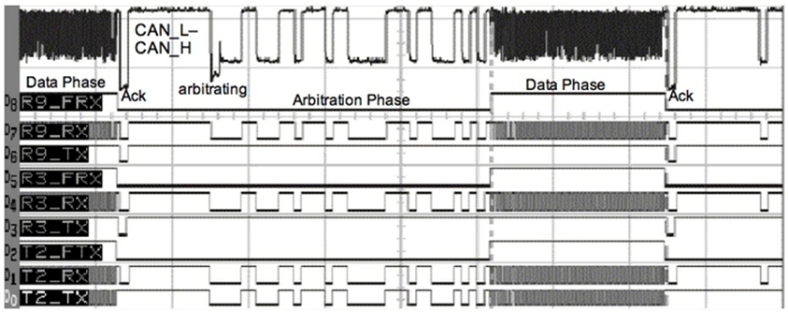

Figure 3.15: Frame transmission, including arbitration between 4 nodes [Lin12].

3.1.11.1 CAN FD Data Frame

The CAN FD frame has some bits redefined and with some new fixed levels.

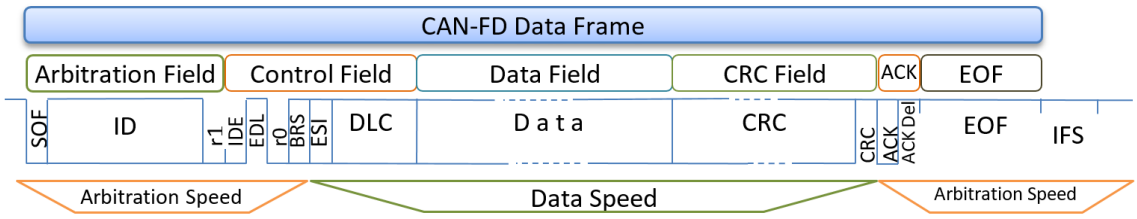

Figure 3.16: CAN-FD Data frame.

Looking at Figure 3.16 we can identify,

- *reserved bit, r1* - 1 bit, dominant level;
- *Identifier Extension Flag, IDE* - 1 bit, dominant level;
- *Extended Data Length, EDL* - identifies this as CAN FD frame, recessive level (in CAN2.0A this was the r0 bit, dominant level);
- *r0* (*reserved*) - dominant level;
- *Bit Rate Switch,* BRS - the value defines if a different bit rate should be used or not in data field transmission. With a recessive level the bit rate is higher, else it uses the same bit rate as in the arbitration phase;
- *Error State Indicator, ESI* - has dominant level if the controller is in Error Active state and recessive for Error Passive state.
- *Data Length Code, DLC* - represents the number of bytes in the data field. For the first eight combinations is the same as CAN 2.0 and the remaining 8 the coding is presented in Table 3.7, being the maximum value 0xF that corresponds to 64 bytes of payload;
- *CRC* - to guarantee the same Hamming distance as in CAN 2.0, the CRC polynomial is different depending on the data field length. CRC vector uses 17 bits (DLC $\leq$ 16) or 21 bits (all other cases). The CRC vector calculation now includes the stuff bits, unlike the CAN2.0 that does not use them. In this field the stuff bit insertion rule is different, being now static and one stuff bit is inserted each 4 bits, which translates to a total of 21 bits and 25 bits effectively transmitted in the bus, respectively.

The CAN data frame now includes the ESI bit (Error State Indicator) that informs the other nodes of the current error state of the transmitter node. Also, remote frames are not supported anymore, being the RTR bit renamed r1 and has always a dominant level. Considering the minimum and maximum bit stuff insertion, a CAN-FD transmission time can be calculated using equations (3.4) and (3.5), where τ_{arb} and τ_{data} corresponds to the

Table 3.7: Possible value of DLC field and payload size in CAN-FD

DLC	Data bytes
0000 ... 1000	0 ... 8
1001	12
1010	16
1011	20
1100	24
1101	32
1110	48
1111	64

bit time in arbitration and data phase, repectively, and DLC is the number of payload bytes [BS14].

$$C_{i(min)} = 29 \cdot \tau_{arb} + \left(27 + 5 \cdot \left\lceil \frac{DLC - 16}{64} \right\rceil + 8 \cdot DLC\right) \cdot \tau_{data} \tag{3.4}$$

$$C_{i(MAX)} = 32 \cdot \tau_{arb} + \left(28 + 5 \cdot \left\lceil \frac{DLC - 16}{64} \right\rceil + 10 \cdot DLC\right) \tau_{data} \tag{3.5}$$

Figure 3.17 presents an interesting comparison, showing that we can send a 64 bytes CAN FD frame, with a speedup factor equal to 8, in less time than an 8 byte data-frame that uses CAN 2.0 standard.

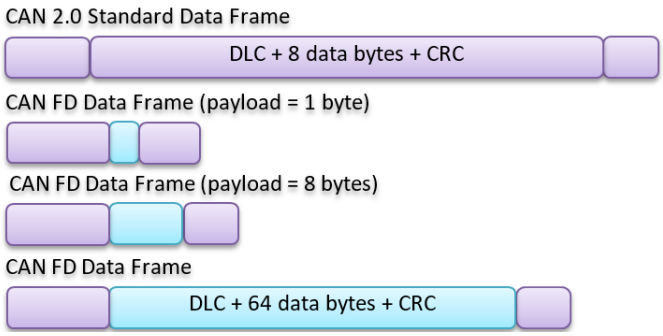

Figure 3.17: Comparison between transmission time of standard CAN Data frame and CAN FD with speedup factor of 8.

3.2 Other Communication Protocols

A description is made of several available communication networks and protocols, detailing the ones that we are somehow related to this thesis.

Table 3.8: Transmission time of CAN-FD Frame, with speedup factor equal to 8 and bit rate 1 Mbps

DLC (bytes)	0 stuff bits (μs)	Maximum stuff bits (μs)
0	32.375	35.500
1	33.375	36.750
2	34.375	38.000
3	35.375	39.250
4	36.375	40.500
5	37.375	41.750
6	38.375	43.000
7	39.375	44.250
8	40.375	45.500
12	44.375	50.500
16	48.375	55.500
20	53.000	61.125
24	57.000	66.125
32	65.000	76.125
48	81.000	96.125
64	97.000	116.125

3.2.1 TTP/C

TTP/C is a communication protocol [KB03] specifically designed for safety-related automotive applications, being its development led by Prof. Hermann Kopetz of the Technical University of Vienna, spanning for a period that lasts more than two decades.

From the architectural point of view it has dual passive bus, as represented in Figure 3.18. To obtain fault tolerance the nodes can be replicated and grouped to form a Fault-Tolerant Units (FTU), being diverse configurations possible, depending on the type and severity of faults to tolerate. In this figure there are 3 FTUs, where two of them have 2 node replicas, that perform the same functions, and one of them is non-replicated (has cardinality equal to one).

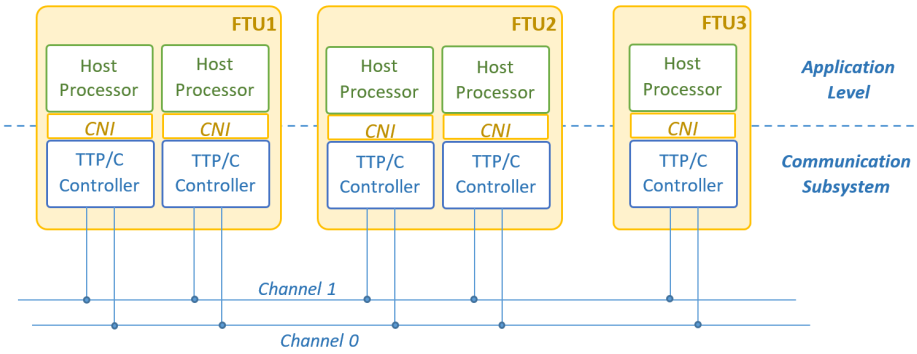

Figure 3.18: Example of TTP/C cluster.

In TTP/C terminology a node in a FTU is also called Smallest Replaceable Unit (SRU), and the internal structure is presented in Figure 3.19.

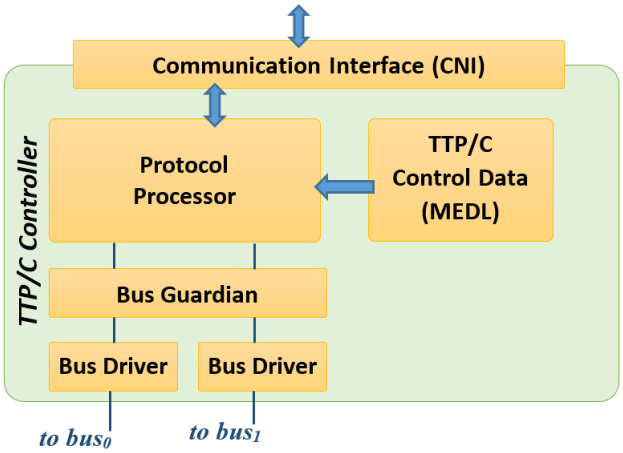

Figure 3.19: Node internal structure

A bus guardian is a module that regulates the access to the bus, restricting the access to determinate time intervals. In TTP/C, the bus guardians ensure transmission only during the correct timeslot, in all cases, guaranteeing a fail silent behaviour. The guardian can be local, as in the case of bus topology, or central when a star is used.

The protocol permits also star topologies, being possible to have single star with central bus guardians, replicated star and mixed bus/star topologies.

The bus access scheme uses TDMA, with a fixed assignment of slots to nodes and every node must send a message periodically, so there are guaranteed delivery times, with known jitter. Each TDMA round is divided in slots, that can have different sizes. Due to fault tolerant considerations, the messages in both buses follow the same sequence in each TDMA round. In each TDMA round, each node has a slot allocated, meaning that the number of slots is equal to the number of nodes that pertains to the network.

Figure 3.20 represents the cluster cycle for the network represented in Figure 3.18, where message with number 1 refers to SRU1 and 2 for the SRU2. In this figure we can observe that a slot is a window allocated to a node to transmit a message, a TDMA round is a sequence of slots, where each station transmits exactly one per round and a cluster cycle is a sequence of different TDMA rounds.

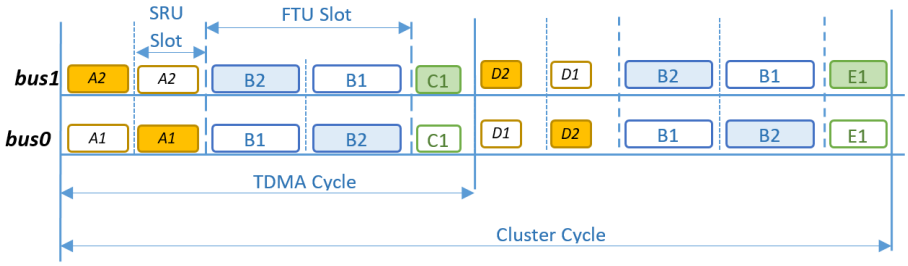

Figure 3.20: TTP/C round.

In each slot a frame must be transmitted, even if there is no new data to transfer. There are three types of frames: Normal, Initialization and Extended. The normal frames (N-frames) carry user data, with a maximum payload equal of 16 bytes including also protocol related field (header) and 3 bytes of CRC [MBSP02]. Initialization frames (I-frames) are protocol specific that carries specific state information, allowing nodes to integrate an operational cluster, being composed by header, the C-State with maximum of 6 bytes and 2 bytes CRC. The C-State is the Communication Controller state that includes information on current mode, the time field that denotes current global time and the membership field that contains activity information on all nodes of the cluster. Both protocol and user data can be transmitted in extended frames (X-frames).

For correct TDMA access is of utmost importance that each node has the same notion of global time. This is provided by a distributed clock synchronization service, that implements a fault-tolerant average algorithm, with and achievable precision of 1 μs.

The possible bus data rate are 500 kbps, 1 Mbps, 2 Mbps, 5 Mbps or 25 Mbps.

Every TTP/C controller possesses a data structure, the Message Descriptor List (MEDL), that describes the complete communication pattern, corresponding to a static schedule. So each node, has knowledge of all the messages (sent and received) and their encoding and also the complete message dispatching table. It is possible to have mode changes, that are also described in the MEDL.

Also, the protocol includes a Membership Service, being every node's membership status made available in each TDMA round, explicitly in I-frames or embedded in the message CRC field of N-frames. By analyzing this information a node can determine the correct functioning status of each node in the cluster.

Event channels can be defined by a priori reservation of a specified number of bits in a message, which are reserved for that particular node and for this reason cannot be shared among nodes. So, in what concerns asynchronous traffic, the bandwidth efficiency is low.

3.2.2 Ethernet

Ethernet is defined by standard IEEE802.3, was proposed by Metcalfe in 1976 [MB76], being nowadays one of the most used networks in the world, in practically every area. Referring to shared Ethernet, it uses as transmission media a bus, being the access control via CSMA/CD. A station with pending messages to transmit that finds the bus free, starts transmitting and simultaneous listen to the channel signals. If it senses a collision (that can happen if two or more nodes starts transmitting simultaneously), its stops transmitting its data and sends a jamming signal. After it delays the message retransmission, being the delay given by random exponential backoff algorithm, which is different for each node. Due to the algorithm used, the bus access is non-deterministic, being this a reason to be considered, as is, unsuitable for real-time network, despite the large bandwidth available.

Ethernet as in IEEE802.3 standard, defines the two lower layers (Data Link and Physical) of the seven-layer OSI networking reference model. Shared Ethernet was initially available, circa 1983, with date rates of 10 Mbps over coaxial, followed by twisted unshielded/shield cable using star-wired cabling topology with a central hub. Other speeds were standardized, 100 Mbps referred as Fast Ethernet and 1000 Mbps as Gigabit Ethernet, along with multiple variants of each.

The bus constitutes a single collision domain, limiting the communication between network nodes to half-duplex.

A Ethernet frame is divided in the fields described next, being represented in Figure 3.21.

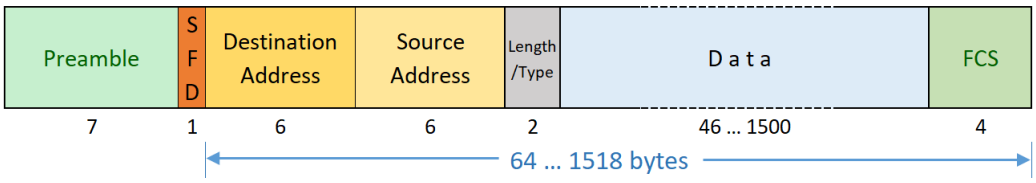

Figure 3.21: Ethernet Frame

- *preamble* - 56 alternating 0 and 1 bits to synchronize receiver clocks, 7 bytes;
- *Start of Frame Delimiter, SFD* - marks the end of preamble, with binary sequence 10101011, 1 byte;
- *MAC Destination Address* - 6 bytes;

- *MAC Source Address* - 6 bytes;
- *Length/Type* - defines the payload size, and with values greater than 1536 signalizes EtherType, 2 bytes;
- *Data* - user data, 46 to 1500 bytes;
- *Frame Check Sequence, FCS* - frame CRC, 4 bytes.

After each frame transmission an idle time between packets must exist, referred as inter-packet gap, which comprises the time need to send 12 bytes (idle line state).

3.2.3 Proposals for Real-Time/Industrial Ethernet

In Switched Ethernet the hub is substituted by a switch, granting this way full-duplex communication between nodes. The topology used is an active star, where the center point is a switch that connects each node with two links - uplink and downlink, as represented in Figure 3.22.

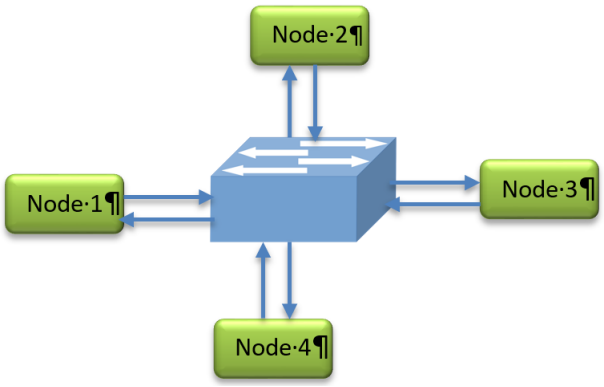

Figure 3.22: Switched Ethernet

The switch type can be classified as *Store and Forward* and *Cut Trough*, where the first stores the incoming frame and checks it for validity before forwarding it to its destination ports. The later, starts forwarding the frame as soon as the destination MAC is decoded, presenting lower latency but may forward frames that contain errors, contrary to the first type.

3.2.3.1 Avionics Full-DupleX Switched Ethernet (AFDX)

Predictability is a mandatory requirement in the aviation industry, as it is necessary to obtain system certification. Also due to demands in bandwidth due to the deploying of fly-by-wire systems, the use of Integrated Modular Avionics (IMA) modules and the need to replace point-to-point connections, along with demand for lower costs and use proven and widespread technology, lead to the introduction of Ethernet in this demanding field.

The Avionics Full-DupleX Switched Ethernet (AFDX) is based on switched Ethernet with a bit rate of 100 Mbps [AFD05]. The point-to-point communication links are substituted in

the AFDX network by the uses of Virtual Links. These are defined as 1-to-many distribution list, so the switch receiving the frame, inspects it and delivers the frame to the n end nodes that possess the VLID (Virtual Link ID). An AFDX frame is presented in Figure 3.23 and its relation to standard Ethernet frame.

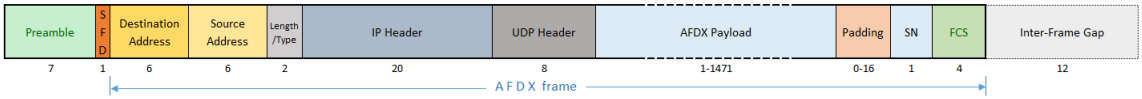

Figure 3.23: AFDX mapping on Ethernet frame

48 bits	
Constant field (32 bits)	**Virtual Link Identifier (16 bits)**
xxxxxx11 xxxxxxxx xxxxxxxx xxxxxxxx	vvvvvvvv vvvvvvvv

Ethernet MAC Controller Identification (48 bits)			
Constant field (24 bits)	**User _Defined_ID (16 bits)**	**Interface_ID (3 bits)**	**Constant field (5bits)**
00000010 00000000 00000000	nnnnnnnn nnnnnnnn	mmm	00000

Figure 3.24: Addressing in AFDX

To guarantee predictability a rate-constrained strategy is used. Since the message traffic share the same physical links, the use of virtual links prevents the traffic of different virtual links to interfere. To enforce this traffic isolation is necessary to limit the rate and size of the Ethernet frames per virtual link. Then, each virtual link has the following parameters:

- *BAG - Bandwidth Allocation Gap* - is the minimum time between sending 2 consecutive frames. This value must be between 1 and 128ms, being a power of 2;
- *Lmax* - the largest Ethernet frame that can be transmitted in the virtual link (in bytes);
- *Jitter* - an upper bound on frame transmit latency, measured from the *BAG* start instant (in μs).

The flow in the node, is regulated by the Virtual Link Scheduler, that multiplexes the frames from each virtual link queue, based in individual *BAG* and *Lmax*, transmitting it at the predefined time instants for the specified Virtual Link. An example is presented in Figure 3.25.

The virtual links are defined statically and their parameters, *BAG* and *Lmax*, should be optimized to minimize bandwidth use, guaranteeing the timeliness of all messages. One example of this process is presented in [SBCH13].

To guarantee fault tolerance to permanent faults, AFDX can be used with replicated switches, nodes and links, where the two networks transmits exactly the same data. In the end system a Redundancy Management module is responsible for discarding the redundant frames.

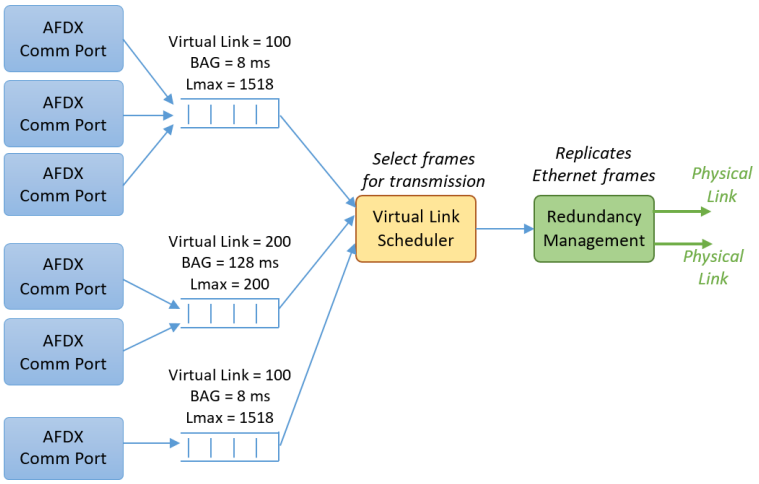

Figure 3.25: AFDX Virtual Link Scheduling.

3.2.3.2 TTEthernet

The TTEthernet protocol was developed by TTTech to enable time-triggered communication over Ethernet, being inspired by the TTP/C protocol, targeting the auto industry, avionics systems and industrial automation [KAGS05].

This protocol is constructed on top of switched Ethernet, adding an adaptation layer termed Time-Triggered Extension, on top of the 802.3 standard. The standard and the safety-critical architecture, where the last one uses replicated TTEthernet switches and links, central bus guardians and enhanced TTEthernet controller, are presented in Figure 3.26.

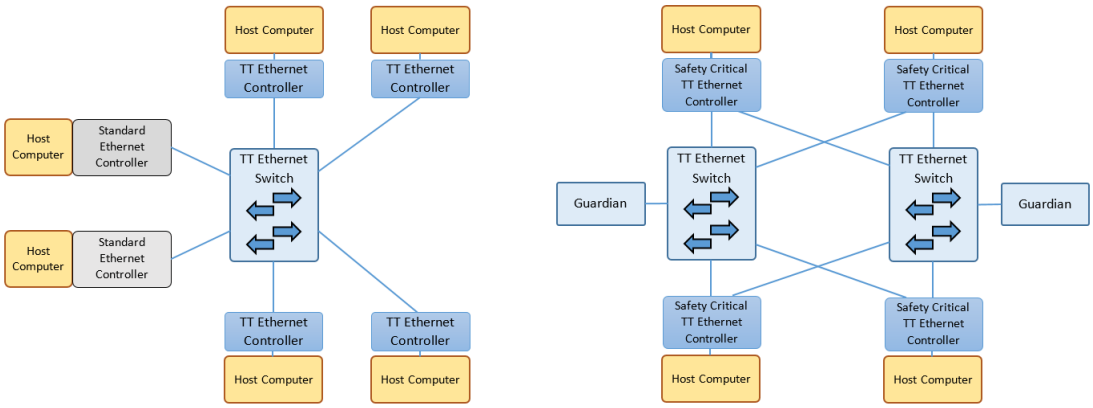

Figure 3.26: TTEthernet architecture: Standard vs Safety-Critical (adapted from [KAGS05]).

Time-Triggered (TT) messages are sent in precise time instants, according to a predefined communication schedule, obtained off-line and static. Virtual links (VLs) are obtained in the scheduling process, defining the logical connection between sender and one or more receivers and is identified by the critical traffic identier (CTID). The switch is loaded with a statically defined forwarding table that associates the VLs obtained in the scheduling process with the corresponding output ports. As the network configuration is static, the messages always follow the same VLs, they present a predictable behaviour, guaranteeing the timeliness of the

TT messages.

In TTEthernet three traffic classes are defined: Time-Triggered, Rate-Constrained and Best-Effort, trying to give adequate QoS to messages used in processes with diverse requirements. Rate-Constrained (RC) messages always have minimum time interval between frames pertaining to the same stream, so there is a maximum bandwidth assigned, with delays and jitter with known limits. Best-Effort (BE) traffic is intended for classical Ethernet frames, without any temporal guarantees and, in the limit, no guarantee of message delivery.

An example of a TTEthernet communication cycle is presented in Figure 3.27.

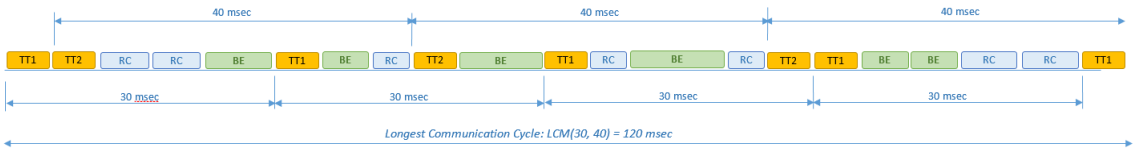

Figure 3.27: Defining a TTEthernet cycle.

The TTEthernet frame is presented and compared with the standard frame in Figure 3.28. The destination address is substituted by the CT Marker (4 bytes) which is a static identifier used to distinguish time-triggered frames from other traffic and CTID (2 bytes) that is used by the switches to route time-triggered frames through the network.

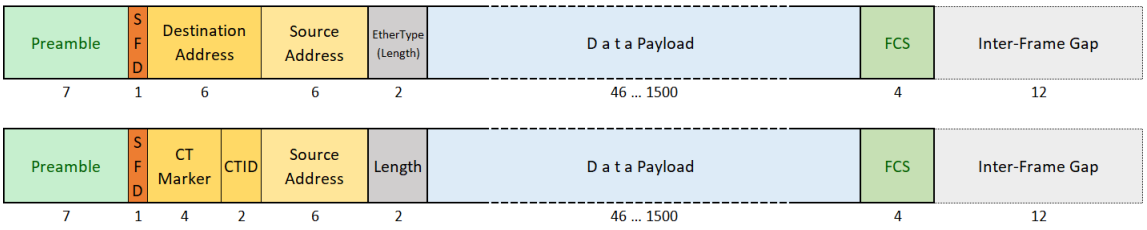

Figure 3.28: Ethernet and TTEthernet frames.

TTEthernet defines a fault-tolerant clock synchronization mechanism (SAE AS6802) that is used to synchronize network components, necessary to obtain timeliness and low jitter of TT messages.

3.2.3.3 FTT-Ethernet

The FTT-Ethernet is an instantiation of the FTT paradigm over shared Ethernet, using COTS Ethernet controllers, that are enhanced with a software layer so they conform to the TM message requests, which are sent by the Master node [PAG02].

The time is divided in Elementary cycles and each EC in the Synchronous Window is followed by the Asynchronous one, being an example presented in Figure 3.29.

The Master has a database with the characteristics of real-time messages, synchronous and asynchronous and also non real-time messages. The scheduling of synchronous messages is performed online in the Master node and can use any scheduling algorithm.

The TM message is represented in Figure 3.30, having the broadcast as destination address, to reach all nodes. The frame Ethernet data field carries the FTT frame that includes

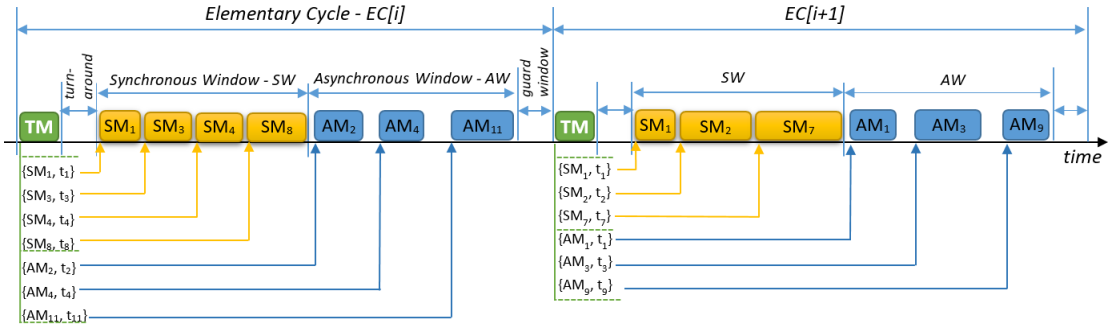

Figure 3.29: Elementary Cycles in FTT-Ethernet (adapted from [Ped03]).

the message type (MST_ID for TM message), the number of data messages that should be transmitted in the current EC (#Msgs). For each message, the identifier and transmission duration (Msg ID & Len) follows.

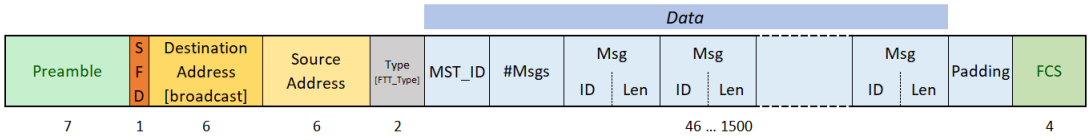

Figure 3.30: FTT-Ethernet Trigger message (adapted from [Ped03]).

At the slave side, there is a modified DLL with a transmission control layer put on top of the Ethernet layer, both for real-time and non real-time traffic. This is necessary to avoid collisions in the synchronous window, in order to maintain the timeliness of messages, and confine each traffic type to the respective window. A synchronous data message is an Ethernet frame where the Data field is composed by Mesg_ID, Counters & Flags and the real-time data and is represented in Figure 3.31.

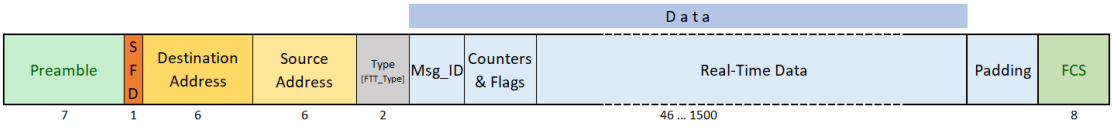

Figure 3.31: FTT-Ethernet Data Message (adapted from [Ped03]).

3.2.3.4 FTT-SE and HaRTES

Moving from shared to switched Ethernet brings important efficiency gains as the protocol can take advantage of features like the absence of collisions and the existence of parallel transmission paths. Due to to the inherent absence of collisions for each port leads to a major simplification of the protocol implementation in the slave nodes, as they do not need anymore to enforce a collision-free medium access. As the message serialization is performed by the switch, the slaves can transmit the messages immediately after decoding the TM, instead of waiting for a specific moment to transmit (that they must calculate and time precisely),

as was the case with shared Ethernet. This way, the contents of the TM are simplified, needing only to convey the ID of the messages to transmit. Moreover, it is possible to take full advantage of multiple transmission paths by abandoning the pure broadcast architecture of FTT-Ethernet. This can be achieved by building optimized schedules that exploit this parallelism, increasing the aggregated throughput.

The new communication system architecture, where the FTT Master is attached to one switch port and sends the TM to other stations is presented in Figure 3.32 [Mar09].

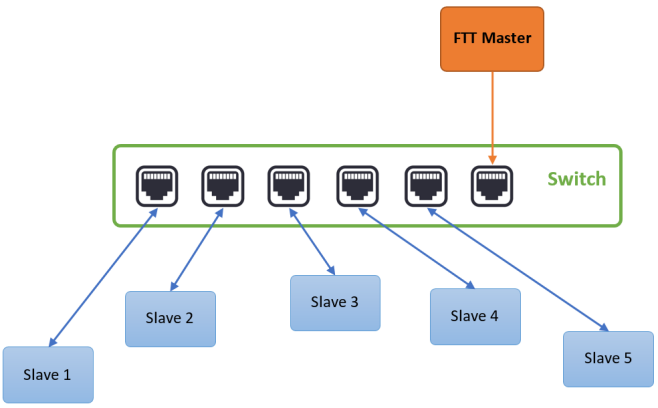

Figure 3.32: FTT-SE Architecture (adapted from [Mar09]).

The HaRTES - Hard Real-Time Ethernet Switching [San11] - represents the evolution of FTT-SE, and it now uses a modified Ethernet switch that is based on the FTT paradigm.

The use of the FTT-enabled Ethernet switch overcomes structural limitations in the FTT-SE protocol, as for instance the ones that require that all participating nodes are FTT-compliant, which implies the deployment of specific device drivers in the nodes that might not be readily available. The participation of legacy nodes not conforming to FTT rules may completely jeopardize timeliness of real-time messages

By adding traffic confinement capabilities to the switch the above limitations can be overcome, resulting then in the FTT-Enabled Ethernet switch, represented in Figure 3.33.

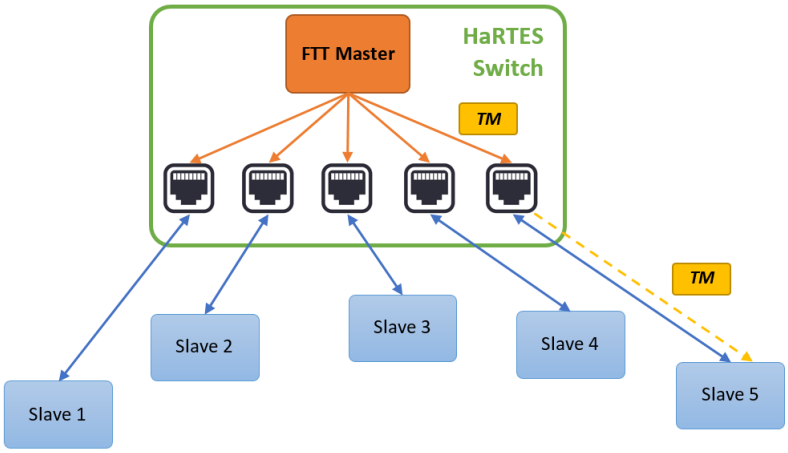

Figure 3.33: HaRTES Architecture (adapted from [San11]).

The new architecture allows gains in key aspects, as for instance an increase in system integrity as unauthorized transmissions can be promptly blocked, seamless integration of non FTT-compliant traffic without compromising real-time timeliness of synchronous messages, transmission of TM with higher precision or a simplification in handling asynchronous traffic.

3.2.3.5 Fault Tolerance for FTT Architecture

The FT4FTT intends to be an architecture that conforms to the FTT paradigm for a distributed embedded system which supports applications that are real-time, highly-reliable and adaptive [GPBB19]. It uses a communication subsystem, called Flexible Time-Triggered Replicated Star for Ethernet (FTTRS), which is based on Hard Real-Time Ethernet Switching (HaRTES). The FTTRS communication subsystem is presented in Figure 3.34.

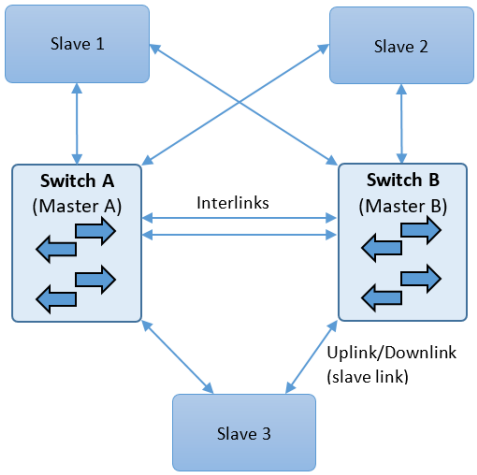

Figure 3.34: FTTRS architecture (adapted from [BDBP06]).

The FTTRS intends to increase the reliability by overcoming vulnerabilities of HaRTES. This is achieved by eliminating the single points of failure of HaRTES with critical components replication. It ensures the replica determinism of these replicated components, it restricts the failure semantics of the components, and it makes message exchanges capable of tolerating transient and permanent channel faults.

3.2.4 Other Real-Time protocols and Industrial Internet

Other Ethernet proposals exists, but with very specific characteristics, as they modify standard Ethernet to attain real-time. The more popular industrial Ethernet protocols are Profinet, EtherNet/IP, EtherCAT, SERCOS III, and PowerLink [WI11].

3.3 Time-Triggered Protocols for Operational Flexibility

This section presents in detail three protocols: TTCAN, FTT-CAN and FlexRay.

3.3.1 TTCAN

The TTCAN introduces a time-triggered framework on top of CAN protocol, is described in 11898-4 ISO standard [STA04] and corresponds to a Layer 5 (Session Layer) of the OSI model.

In TTCAN the communications are organized in a system matrix, as depicted in Figure 3.35, being this divided in basic cycles, with a maximum number of 64 and always a power of 2. In this matrix the row represents a basic cycle (BC) and the column, termed Transmission Column (TC), refers to the window duration. Each basic cycle is composed by time slots or windows of fixed duration, being possible to transmit one message in each window. The cycle always start with a Reference message. This message contains the cycle counter (and possibly other data), so each node prepares the messages to send, if any, that will be sent in the specified time window. There are three different types of windows:

- Exclusive/Reserved - in this window only the message with the specified ID can be transmitted, typically used for periodic messages;
- Arbitration - in this window any node can transmit, being the native CAN arbitration mechanism used for medium access, but anyway the message cannot end after the defined window boundary;
- Free - window not available for message transmission, being reserved for future use.

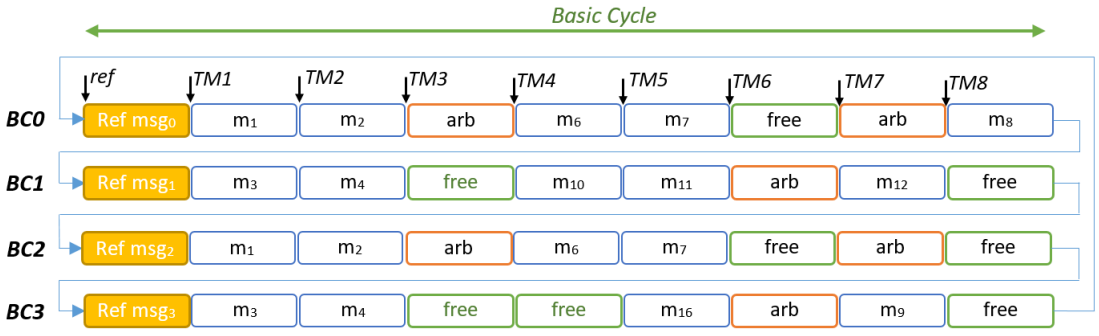

Figure 3.35: TTCAN System Matrix.

Each matrix column must have the same size in every basic cycle, implying that if different messages are sent in different basic cycles then the column size is defined by the longest message, implying a waste in used bandwidth. Since the columns have a fixed size, the retransmission mechanism in case of error is disabled in a way to not compromise the defined time schedule.

At the start of each window (except for the Reference message window) a time interval is defined, where the message scheduled for this TC/window must start transmission, as long as it finds the medium free. This requirement ensures that the assigned message for this TC will not overflow to the next time window, guaranteeing timeliness of messages in the matrix cycle. This time interval is Transmission Enable Window (TEW) and in the case of

arbitrating windows it will start at the beginning of the first TC and ends at the end of the TEW of the last merged window.

3.3.1.1 Timing Synchronization and Fault Tolerance

The reference message payload is depicted in Figure 3.36. For level 1, the Control Information byte uses 6 bits to define de Cycle Count (0 ... 63), 1 bit is reserved and the eighth bit (MSB) indicates if the next BC starts immediately after the current one or if it is triggered by some event. The remaining bytes could be used to transmit user data. Using Level 1 timing the nodes use a clock implemented by a 16 bit counter that is reset with the reception of the SOF bit of each reference message. The counter is incremented each Network Time Unit (NTU), obtained from the CAN protocol engine, being one NTU equal to the nominal bit time (e.g., $1\mu s$ for 1 Mbps bit rate).

Figure 3.36: Reference message payload

In level 2 timing the nodes uses at least a 19 bit wide counter as the node clock (at least 8 times faster than the nominal bit rate) and the reference message now contains the cycle count and a three byte vector with the global time that is held by the current time master node. The remaining 4 bytes can be used to transfer user data. Using the received global time and comparing it with the local node time, an offset and rate are obtained that is afterwards used to adjust the local clock to the master clock. Additional details on the clock synchronization algorithm can be found in [HMFH02]. The attainable clock precision using Level 2 timing is equal or less than 1 μs. The normal functioning of a TTCAN network implies the regular transmission of reference messages, which are sent by a time master. The absence of this message stops all communications, as is this messages that triggers all other message transmissions. So, to guarantee that the reference message is transmitted it is possible to define up to eight potential time masters. Each master is configured with a reference message ID and a time-out value, defining a hierarchy of time masters, where the highest priority is for lower message ID and time-out. If the current time master fails to send the reference message, then the next potential time master will send it. If this one also fails, then the next in the hierarchy will try and so on. On detection of reference message in the bus, any potential time master will abort their pending transmission, guaranteeing that only a reference message is transmitted per basic cycle.

3.3.1.2 Scheduling Algorithms

The matrix cycle must be defined before system start and loaded in the microcontroller so the system can conform to it.

This is obtained by executing a scheduling process, that takes into account the characteristics of the system message set and the protocol constraints. For instance, all transmission columns must have same duration, despite messages with different transmissions times can be allocated to it, in different basic cycles.

The scheduling process is then an highly complex optimization process that can be tackled recurring to heuristic methods, as for instance the ones described by Schmitd [SS07], or even to stochastic algorithms, as in [CBF01]. The final objective is, of course, to obtain a schedulable system, but also to leave as much as possible free bandwidth/windows for eventual future upgrades or having low jitter.

3.3.1.3 Inconsistency Scenarios in TTCAN networks

TTCAN is built on top of the CAN protocol, inheriting the generic characteristics of it. The inconsistency scenarios presented for CAN in Section 3.1.10 are still present with a major difference, all the IMD scenarios are now transformed in IMO scenarios. This happens since the sender when detects an error is prevented from retransmitting the message, due to the use of single shot transmission. Then, this corresponds to the IMO scenario previously referred.

So, with a similar characterization of the system as the one used to derive Table 3.6, and noting that now there is not possible to have IMD scenarios, for the same environments the number of IMO scenarios per hour for a TTCAN network are presented in Table 3.9.

Table 3.9: IMO scenarios in TTCAN, for the diverse environments

Environment	BER	IMO/hour
Benign	$3.0 \cdot 10^{-11}$	$8.60 \cdot 10^{-4}$
Normal	$3.1 \cdot 10^{-9}$	$8.89 \cdot 10^{-2}$
Agressive	$2.6 \cdot 10^{-7}$	7.45

This constitutes a serious impairment to use this protocol in high-reliability and safety-critical applications, as the number of IMO scenarios or transmissions failures per hour is far to great, even in benign scenarios.

The TTCAN protocol did not get general acceptance by the industry, despite the good characteristics and the initial enthusiasm about this protocol extension. In fact, the number of hardware chips that implement this protocol is very low, with the exception of a couple of

microcontrollers from Microchip, Texas Instruments and Infineon , which is scarce compared with the generalized offering of microcontrollers with CAN 2.0 controllers, from multiple vendors.

3.3.2 FTT-CAN protocol

According to the FTT paradigm [APF02], the network time is divided in a succession of ECs, with a preconfigured fixed duration, which constitutes the temporal resolution of the traffic, as represented in Figure 3.37.

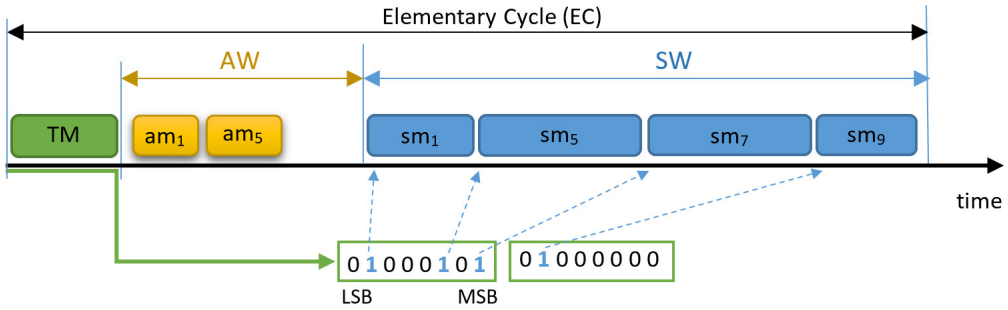

Figure 3.37: Elementary Cycle (EC) and Trigger Message (TM) encoding in Flexible Time-Triggered Controller Area Network (FTT-CAN).

FTT-CAN that corresponds to an instantiation of the FTT paradigm over CAN, is a Master-Slave protocol and the master node schedules the TT traffic online, for each EC, communicating the schedule to the slave nodes using a Trigger Message (TM) transmitted at the beginning of each EC. FTT-CAN also supports ET messages, which are triggered autonomously by each node. The EC is composed of two windows, designated Asynchronous Window (AW) and Synchronous Window (SW), which carry the ET and TT traffic (respectively). The duration of each SW depends on the TT traffic scheduled for that EC and is communicated to the nodes in the respective TM.

The FTT paradigm was already implemented and demonstrated using several underlying technologies, such as CAN [APF02], shared Ethernet [PAG02], and switched Ethernet [Mar09], [San11].

The FTT-CAN protocol implementation uses a simplex bus and the TM encodes in its payload the messages to be transmitted in that EC using one bit per message request (see Figure 3.37). Each slave decodes the TM, and, at the beginning of the SW, triggers the transmission of scheduled messages, for which it is the producer. FTT-CAN only controls which messages are transmitted within the SW, not defining a particular order, which tends to follow the native CAN arbitration scheme, with possible priority inversions due to practical technological issues, like the nodes latency.

The Master uses an online scheduler that can implement any scheduling policy, e.g., FP (Fixed Priority), RM (Rate Monotonic), or EDF (Earliest Deadline First), being independent of the arbitration process of the underlying network technology. This node possesses a

database, the System Requirements Database (SRDB), with the attributes of the messages and other system operational parameters, e.g., EC duration. The message set can be updated online using special control messages, e.g., to add and remove messages or modify their attributes. All such requests are directed to the Master node and subject to an admission control mechanism, being accepted only if they result in feasible system configurations. Event messages are triggered autonomously by the end-nodes, relying on the native CAN arbitration mechanism to prioritize and serialize concurrent transmissions. End-nodes are responsible for confining the event traffic in the AW. To do so, they use the information contained on the TM to determine the AW duration and suspend transmission at the appropriate times.

3.3.2.1 Schedulability tests

Communication services are delivered to the application by two subsytems, the Synchronous Messaging System (SMS) and the Asynchronous Messaging System (AMS). As there are requirements on real-time functioning in both types of messages, there is a need to evaluate them.

3.3.2.1.1 Synchronous Messages tests The Synchronous Requirement Table (SRT) is defined in Equation (3.6), where DLC_i is the number of bytes in the message payload, C_i the transmission time (with maximum bit stuffing), Ph_i the relative phase, T_i the period, D_i the relative deadline and Pr_i the message priority.

$$SRT \equiv \{SM_i(DLC_i, C_i, Ph_i, T_i, D_i, Pr_i), i = 1 \cdots N_S\} \tag{3.6}$$

In FTT-CAN, when scheduling the synchronous messages for the next EC, some idle time may appear at the end of the SW, as presented in Figure 3.38.

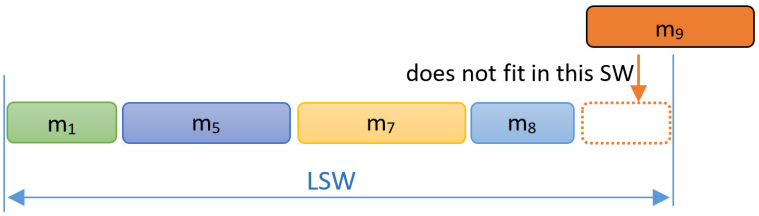

Figure 3.38: Inserted Idle Time in FTT-CAN.

This Inserted Idle Time (IIT) is essential to allow for the transmission of the TM without any blocking. The actual amount of IIT added in each EC depends on the traffic scheduled for that SW and it can be upper bounded by the length of the longest ready message whose transmission must be postponed to a future EC to avoid a TM overrun, as would be the case of message m_9 in Figure 3.38. Using as upper bound of the IIT the greatest of all transmission times, we obtain X as in Equation (3.7).

$$X = \max_{j=1..N_S} (C_j) \tag{3.7}$$

The scheduling model with IIT that is used is the Non-Preemptive Blocking-Free scheduling model [AF01]. Traffic schedulability in this model can be assessed as if the scheduling was fully preemptive, as long as the message transmission times are inflated as in Equation (3.8), where LEC represents the EC duration and LSW the maximum SW duration. This value can then be used in the known schedulability bounds, to assess the schedulability of the message set. Then inflating all transmission times, we obtain the inflated message set. Then, Liu and Layland bound for Rate Monotonic [LL73] scheduling policy can be applied, as in Equation (3.10). The synchronous real-time messages are then schedulable with RM under any phasing if Equation (3.10) is verified [APF02].

$$C_i^E = C_i \cdot \frac{LEC}{LSW - X} \tag{3.8}$$

$$U(LSW - X) = U_{LL} \cdot \frac{LSW - X}{LEC} \tag{3.9}$$

$$U = \sum_{i=1}^{N_S} \left(\frac{C_i}{T_i} \right) < N_S \cdot \left(2^{1/N_S} - 1 \right) \cdot \frac{LSW - X}{LEC} \tag{3.10}$$

The sufficient schedulability condition for EDF policy, under any phasing, is given by Equation (3.11).

$$U = \sum_{i=1}^{N_S} \left(\frac{C_i}{T_i} \right) < \frac{LSW - X}{LEC} \tag{3.11}$$

Response Time Analysis of Synchronous Messages in FTT-CAN

The message set described by the SRT can also be checked using common Response Time Analysis (RTA) [ABR+93], as long the transmission times are inflated as previously referred.

Therefore, when considering an FTT-CAN system with a message set M where each message is described by the *n-tuple* $m_i = \{T_i, O_i, D_i, C_i\}$, schedulability can be guaranteed if an upper bound to message i response time (R_i) when considering the inflated transmission time (C_i^E), as in Equation (3.12), is lower than or equal to the respective deadline (D_i) for all n messages. As usual, Equation (3.12) can be solved with a common fixed-point iteration method and $hpe(i)$ stands for the set of messages having higher or equal priority than message m_i.

$$R_i = C_i^E + \sum_{k \in hpe(i)} \left\lceil \frac{R_i}{T_k} \right\rceil \cdot C_K^E \tag{3.12}$$

This equation and test is for error-free scenarios.

3.3.2.2 Asynchronous Messaging System Scheduling Analysis

3.3.2.2.1 Asynchronous traffic scheduling The Asysnchronous requirement table is defined in Equation (3.13)

$$ART \equiv \left\{ AM_i(DLC_i, C_i, mit_i, D_i, Pr_i), i = 1 \cdots N_A^{RT} \right\} \tag{3.13}$$

where DLC_i, C_i, D_i and Pr_i are defined the same way as in the SRT and *mit* stands for minimum interarrival time, being N_A^{RT} the number of real-time asynchronous messages in the system. There could be other asynchronous messages, without real-time requirements, generically referred as AM^{NRT}.

The analysis of schedulability is detailed in [APF02].

3.3.2.3 Additional information on FTT-CAN

3.3.2.3.1 The Trigger Message TM internal coding is presented in Figure 3.39 [Sil10], where the 4 most significant bits of the CAN ID field are fixed with the bit combination 0001, guaranteeing that TM has higher priority than all other messages in the system. The Master ID sub-field identifies the active Master node, disseminating to the slaves the ID of the current Master (3 bits - 0 .. 7 identifiers). The 3 LSB in the CAN ID are used has a sequence number, from 0 to 7 and repeating afterwards, which allows the slave nodes to infer if they are receiving all TM's and by the correct order.

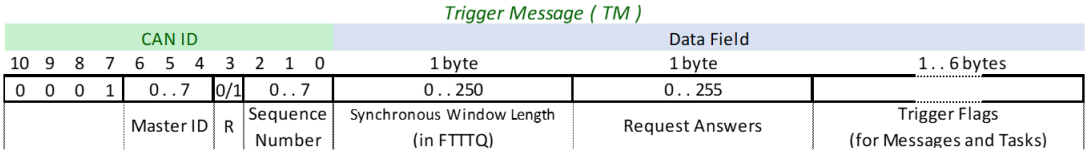

Figure 3.39: TM internal structure.

The TM payload is divide in three parts:

- 1st byte - Length of Synchronous Window, *LSW* - as the LEC is a global system parameter, the slaves use the *LSW* value to synchronize the release of periodic messages at the start of the synchronous window. It also allows to define the time instant of sending and aborting the transmission of asynchronous messages, imposing this way temporal isolation between traffic types. The *LSW* is coded in FTT Time Quantum's, where 1 FTTTQ represents the the number of bits in one EC divided by 250;

- 2nd byte - Master response to slave requests, that would imply changing operational parameter (e.g. add new periodic message to the system);

- 3rd to 8th byte - Trigger Flags - instruct the slaves on synchronous messages to send or tasks to activate in the current EC, where a 1 in a particular bit position defines that the message coded in that bit should be sent (or task activation).

The maximum number of bytes in the payload of a CAN message limits the maximum number of distinct periodic messages that can be triggered to 48, since two of the eight bytes are reserved, as described previously.

3.3.2.3.2 Slave Messages The coding of the messages sent by slaves are depicted in Figure 3.40, presenting the same format, where the four most significant bits define the message type: Synchronous or Asynchronous (Real-Time or Non-Real Time). The sub-IDs assigned give greater priority to Synchronous messages, followed by Asynchronous Real-time messages and lastly by Asynchronous Non Real-time messages, following the standard CAN ID priorities, where lower ID's means bigger priority.

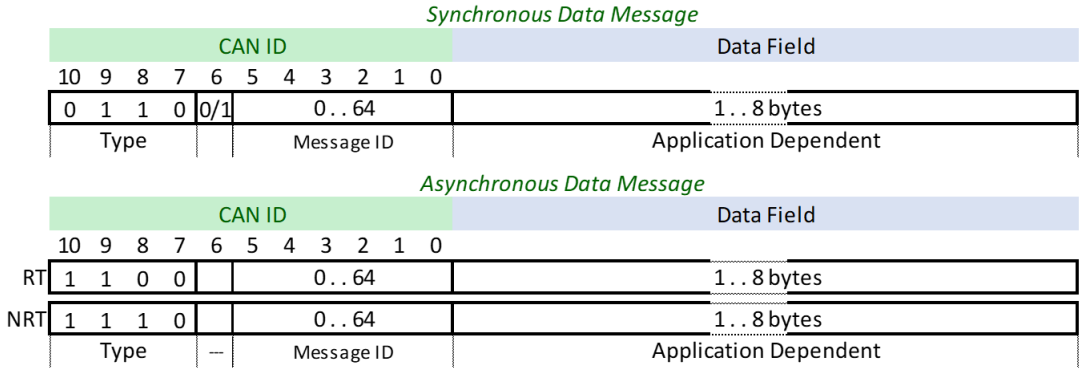

Figure 3.40: Slave messages internals.

3.3.2.3.3 Inside the Master The master internal structure is presented in Figure 3.41 [Fer05]. All periodic message characteristics (period, deadline, offset and DLC) and tasks definitions are recorded in the Synchronous Requirements Database (SRDB). The scheduler implements the scheduling algorithm/policy - Rate Monotonic (RM), Deadline Monotonic (DM), Earliest Deadline First (EDF) or other - and runs on a EC-by-EC basis.

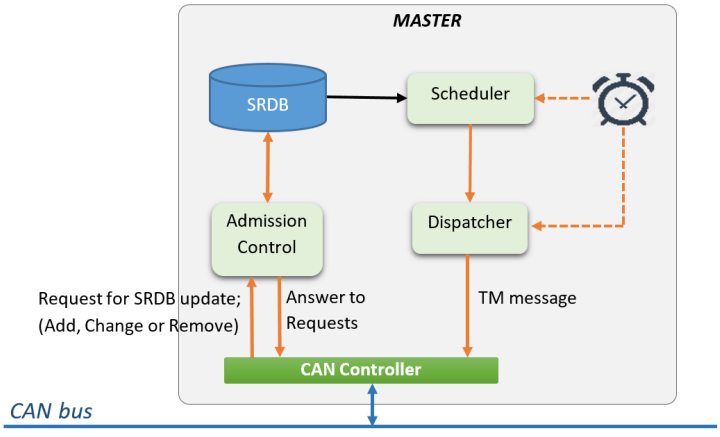

Figure 3.41: Master node internal structure.

The message priority used in the scheduling process is independent of the CAN ID, being this ID relevant only inside the window where it is transmitted (defines transmission order inside EC).

3.3.2.3.4 Operational Flexibility The online scheduler allows a high degree of operational flexibility, as it is possible to add, remove or change parameters of periodic messages, without any interruption in the system service. The requested changes are subject to schedulability verification by an admission control module, to guarantee that any change introduced will not jeopardize the schedulability guarantees given, namely the ones of real-time traffic.

The intended changes are requested by the slave nodes using specific asynchronous messages and do not disrupt the normal functioning of the system. The format of these messages are depicted in Figure 3.42.

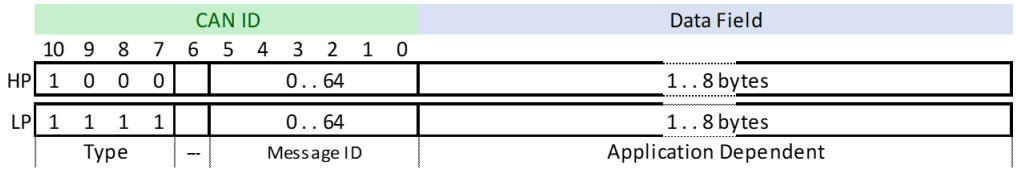

Figure 3.42: Format for request from slaves.

These messages, sent by the slaves with requests to the Master node, have been assigned lower priority than all other message types, being also defined a hierarchy: High and Low priority. This functionality allows, if needed, that the system starts with a minimum configuration (with no periodic messages in the SRDB), being the slaves, after start up, that request the insertion of periodic messages, using the messages just described.

This is a very important characteristic of FTT-CAN, that distinguishes from other TT protocols. So, this protocol presents highly operational flexibility, which contrasts for instance with protocols like TTCAN or TTP/C, where all the synchronous messages are statically defined.

3.3.2.3.5 Master Replication The master node constitutes a single point of failure and any malfunction may render the system non-operational. To make it fault tolerant they propose to replicate the node, with the number of physical replicas depending on the intended system reliability, with a maximum of seven [FPAF02]. The master substitution process is depicted in Figure 3.43. The backup masters must use the single shot transmission mechanism. In the available implementations of FTT-CAN [FTT18], the CAN controllers does not support this functionality, so adaptations were implemented. By placing the TM message in the CAN controller buffer and issuing an abort instruction, the message transmission is effectively aborted if it has not started yet, being also removed from the transmission buffer (that means that the active Master was transmitting its TM message). On the other way, if message already started transmission, then the abort request has no effect and the TM backup message transmission proceeds (corresponding to a situation where the TM message

of the active Master is absent from the bus).

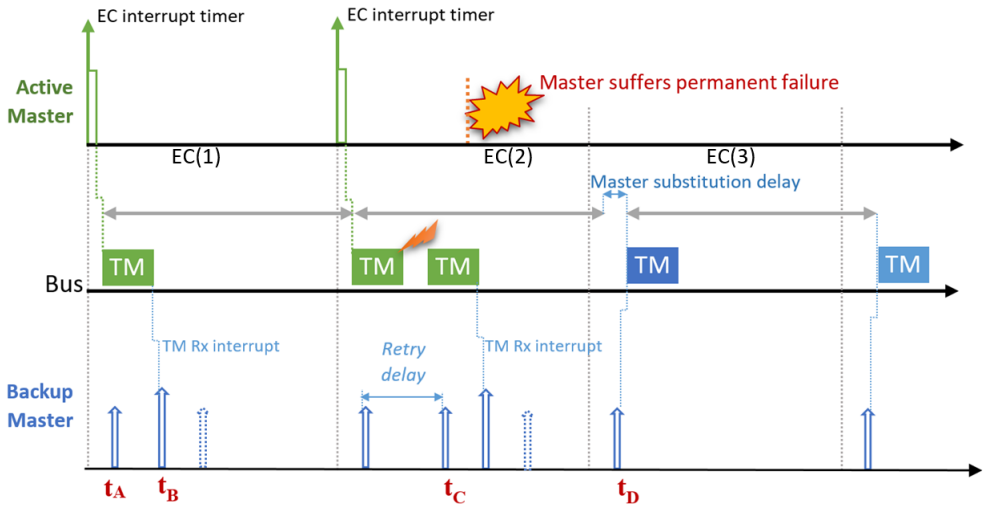

Figure 3.43: Timing for Master replacement due to permanent hardware fault.

In fact, Figure 3.43 presents three scenarios for the functioning of active and backup Masters, that are further described:

- EC(1) - The active master sends the TM and the backup Master tries to send its own TM with a slight delay (at time instant t_A). As the backup Master encounters the bus occupied by the TM message transmission, the abort request is successful and this node starts a timer, with value equal to Retry Delay, to make another transmission try in case the ongoing TM message transmission fails. In the meanwhile (at time instant t_B), the TM is received by the backup node that clears the retry timer and initiates a new timer to trigger the TM message transmission in the next cycle (again with a small delay relative to the active TM).

- EC(2) - at first, the backup TM aborts its TM transmission and the timer retry delay remains active, as the backup master does not receive the active TM, due to TM error transmission. When this timer expires (at time instant t_C), the bus is occupied by the active TM transmission and the sequence described in EC(1) repeats; thus this node maintains its backup status;

- EC(3) - the active master suffer a permanent failure and does not transmit the TM. Then the backup Master finds the bus idle (at time instant t_D) and succeeds in sending its TM, with a small delay, assuming this instant forward the role of active master.

Master replication raises several question concerning consistency and synchronization of their databases, being this problem description and solutions presented in [FAF+06], [Fer05], [MAF+06].

3.3.2.3.6 Multibus Solution Replication The bus also constitutes a single point of failure that can be eliminated by using multiple buses. In the FTT-CAN scope, a solution

was proposed that allows the use of N buses ([SF06], [SFF06]), using the architecture depicted in Figure 3.44, where each node can be connected to one or more buses.

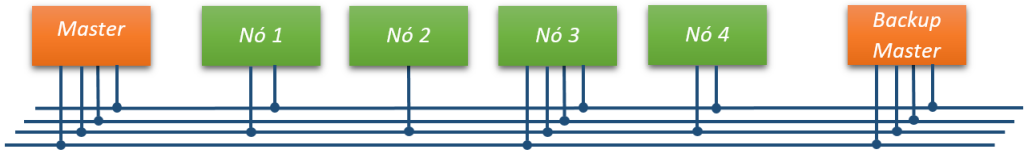

Figure 3.44: Bus redundancy.

The proposed scheme allows sending the same message in several buses, increasing this way the transmission reliability. On the other hand, if different messages are transmitted in the buses, then the available bandwidth increases with the number of buses, in comparison with the single bus configuration. Of course, intermediate situations can be found where bandwidth and reliability can be increased simultaneously. When a bus permanent failure is detected, the master can use one of the other buses to send critical messages, possibly dropping some non-critical messages and working in degraded mode. Using more than one bus also raises other questions, different from the ones found for single bus. Firstly the Master must be updated to schedule messages in more than one bus [SFF07a], perform bus failure detection and master node substitution [SFF07c], and also the management of the operational buses [SFF07b].

3.3.2.3.7 Slotted FTT-CAN Ataide et al. [APL+06] have proposed a static TDMA scheme to transmit messages in the SW. Each slot is defined for the longest message, including also a guard window to guarantee that messages sent in one slot will not overlap to the next slot and compromise timeliness of other messages.

This scheme resolves eventual priority inversion problems and minimizes jitter in message reception. The penalty is in bandwidth use, as the slots are defined for the largest message (8 bytes) plus guard window, implying a wasted time per slot when messages with lower payload are used.

3.3.3 FlexRay Protocol

3.3.3.1 Basic description

New challenges in the automotive industry are imposed by the introduction of new functionalities in automobiles, namely X-by-wire systems (also known as Drive-by-wire) or active safety systems, as for instance ADAS (Advanced Driver-Assistance Systems), that impose higher bandwidth, a deterministic and fault-tolerant functioning, which was not considered at the time attainable by the current network protocols, namely the omnipresent CAN protocol. So, a consortium composed of several car manufacturers (initially BMW, Mercedes, Volkswagen and Audi, later joined by others) and semiconductor companies (Motorola/Freescale, Siemens/Infineon, Philips/NXP), developed the FlexRay specification. The first version was

publicly presented in a conference in April 2002 (Munich) and the first FlexRay Product Day happened in September 2004 [Par07].

The main characteristics are, according to the proponents:

- scalable flexibility;
- possibility of various physical topologies;
- maximum raw data rate equal to 10 Mbps, per channel;
- double channel, for increased fault-tolerance.

3.3.3.2 Physical Topology

The possible physical topologies include point-to-point, passive bus, active and passive star or any combination of the previous ones, exemplified in Figure 3.45.

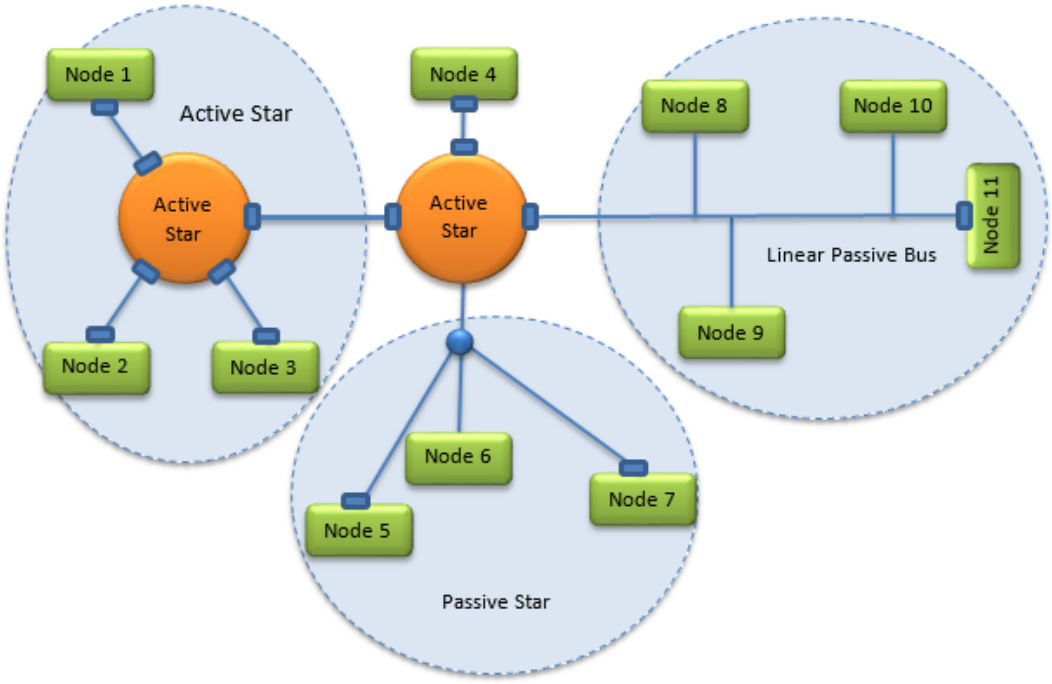

Figure 3.45: Possible physical topologies in FlexRay network - Hybrid Network with passive bus, passive star and active star.

FlexRay Communications System Electrical Physical Layer Specification (Version 3.0.1) document refers several limits that are imposed to the various topologies [Fle10]. In a point-to-point connection, the maximum distance between nodes is 24 meters. Using a bus topology, the number of nodes is limited to 22. Active stars can only be connected by a point-to-point link and the number of active stars in the path between two nodes is limited to two for bit rates of 2.5 and 5 Mbps and to only one if 10 Mbps are used.

A FlexRay node is presented in Figure 3.46, including the Host processor, a communication controller, that implements the protocol stack, bus guardians (not mandatory) and

bus drivers. The bus guardians can be local to the node, in case of bus topology, or centrally placed if an active star is used. These guardians restrict the controller access to the transmission medium in predefined time intervals, preventing this way that a babbling idiot behaviour compromises communications.

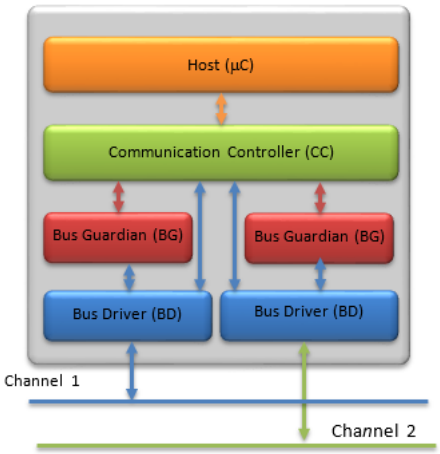

Figure 3.46: Inside a FlexRay node.

Communication bit rate

The 3.0 standard defines three possible bit rate values, 2.5, 5 and 10 Mbps, corresponding to nominal bit duration *gdBit* of 0.4, 0.2 and 0.1 μs. The maximum clock deviation (*gClockDeviationMax*) is defined in FlexRay specification as 1500 ppm, corresponding to the maximum and minimum *gdBit* values presented in Table 3.10 calculated by Equations (3.14) and (3.15), respectively.

$$adBitMax = \frac{gdBit}{1 - gClockDeviatioMAX} \tag{3.14}$$

$$adBitMin = \frac{gdBit}{1 + gClockDeviatioMAX} \tag{3.15}$$

Table 3.10: Bit rate and bit duration in FlexRay

bit rate (Mbps)	2.5	5	10
$gdBit(\mu s)$	0.4	0.2	0.1
$adBitMax(\mu s)$	0.3994	0.1997	0.09985
$adBitMin(\mu s)$	0.4006	0.2003	0.10015

Physical Level Signaling

The bus signal is differential, being the two lines referred as BP (*Bus Plus*) and BM (*Bus Minus*), and the difference between the two is *uBus*. There are four possible values in the

Bus: *Idle_LP*, *Idle*, *Data_0* and *Data_1*, as presented in Figure 3.47. The level present in the bus is dependent on the differential signal between the lines BP and BM. If the differential signal is zero the level is *Idle*, if positive (above a defined threshold) the level is *High* and the other way a *Low* level is present. If a time greater than 11 *gdBit* (*cChannelIdleDelimiter*) in *High* state is detected on the bus, then the bus is considered to be in *Idle* state. Also, the *Idle* level minimizes the used power by the FlexRay nodes (or Bus Drivers).

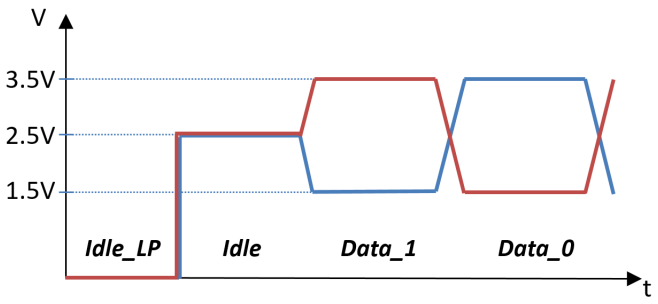

Figure 3.47: Voltage levels in communication lines.

3.3.3.3 Communications Organization

The timing hierarchy in FlexRay defines the *microtick*, *macrotick*, segment and cycle, being these represented in Figure 3.48.

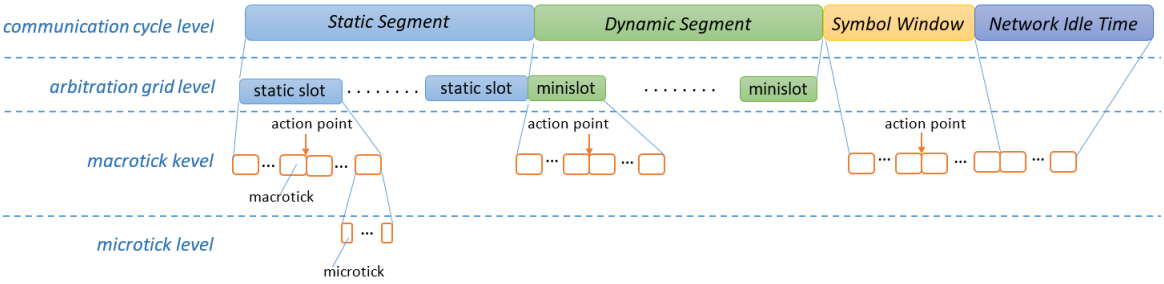

Figure 3.48: Timing Hierarchy in FlexRay (adapted from [Fle10]).

At the lowest level, the *microtick* is directly obtained from the node oscillator clock and can be different in each node, being thus a local value. Since the medium access is TDMA, every communication action must start at a precise moment in time, being each instance defined in *macroticks* (MT). So, the *macroctick* represents the lower level of common time, being its duration equal for all nodes in the cluster. The duration of slots, *minislots*, segments and cycles are also defined in *macroticks*.

At the communication cycle level the cycle is divided in four segments: *Static Segment* (SS), *Dynamic Segment* (DS), *Symbol Window* (SW) and *Network Idle Time* (NIT), being mandatory to have at least two segments, the SS and NIT, being the other two optional. This is depicted in Figure 3.49.

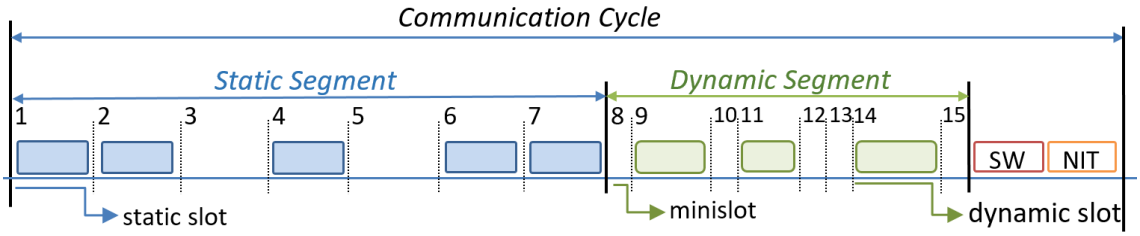

Figure 3.49: Communication cycle in FlexRay.

The SS is composed of static slots and the access method is TDMA, with a static schedule, defined off-line before the system starts. The slots are all of the same size and each one is associated to a particular node on the network. The minimum number of slots in this segment is equal to 2. It is mandatory that the node sends a message in his allocated slot, no matter if it possesses or not new data to send. The macrotick alignment in this segment is presented in Figure 3.50.

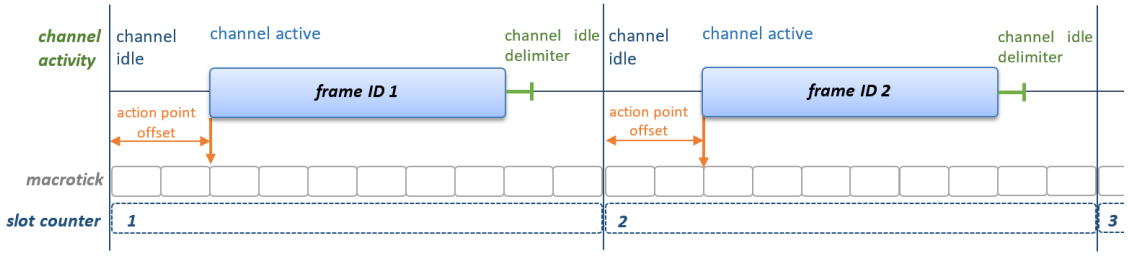

Figure 3.50: Message transmission with macrotick alignment in the Static Segment.

The medium access control in the DS is done by a mini-slotting scheme, also referred as Flexible TDMA, being this segment composed by minislots, all with the same (small) duration. Each one of the minislots is owned by a node that has the chance to send a message starting in his allocated minislot or let the time elapses if it does not have a message to send. If the node transmits a message then it is called a Dynamic Slot, being the minislot enlarged and its duration dependent on the message payload. If the node has a pending message to send that does not fit in the current segment, then it must wait for the next cycle to try to send the message. Details on timing on the DS segment can be observed in Figure 3.51.

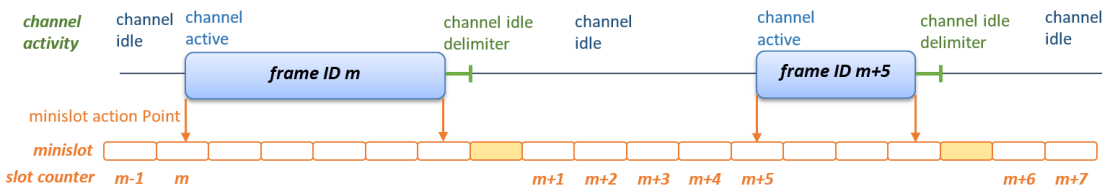

Figure 3.51: Message transmission and minislots in the Dynamic Segment.

The Symbol Window is used for signaling when the network is started and also for network maintenance. In the he Network Idle Time segment there is no communication activities. This

last segment is used by the cluster nodes to adjust the cycle duration, making adjustments in each node clock offset, by adding or removing a particular number of *macroticks*, in a way to achieve a global clock synchronization. This continuous adjustment is necessary due to the fact that is virtually impossible to have exactly the same *macrotick* duration in all nodes and also a drift as time evolves. Due to fabrication tolerances, variations in temperature or aging, for instance, the node crystal frequency can change as times passes, and the clock synchronization can be lost.

3.3.3.3.1 Communication Cycle - Static Segment, Dynamic Segment and others

Communications are performed based on cycles that repeat after the hyperperiod. Defined as minimum of 2 and maximum of 64, and must be o power of 2 number. Access method in the Dynamic Segment is FTDMA, a technique based on the access method used in Bytefligth.

At the frame level, the smallest time particle is the macrotick, being defined and based on the local node clock that each *macrotick* is composed of *n microticks*. The definition of these values is mandatory to achieve good performance.

A **FlexRay frame**, represented in Figure 3.52 is composed of Header, Payload and Trailer, as described.

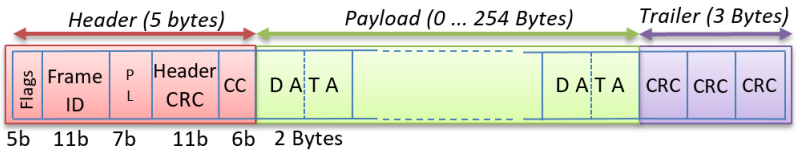

Figure 3.52: FlexRay data frame.

The Header segment is 5 bytes long and its composition is the following:

- Flags (5 bit flag)
  - *Reserved bit* - must be a logical 0 and is reserved for future use;
  - *Payload Preamble Indicator* - a logical 1 indicates the presence of a special vector in the beginning of the Payload. For a frame in the static segment it means that a management vector is present there and if the frame pertains to the dynamic segment then it is the message ID, which is a 16 bit identifier. This message ID can be used by the receiver to decide to store the frame content, after applying a filtering process on the message ID. A logical 0 in this indicator means that there isn't a special vector placed there.
  - *Null Frame Indicator* - a logical 0 implies that the payload possesses meaningful data. A logical 1 indicates that there is no valid data in the Payload, which must be ignored by the receivers.
  - *Sync Frame Indicator* - if this bit is set (logical 1) then the current frame is a Synch Frame, which should be used by the cluster nodes in the clock synchronization mechanism, else this bit must be 0;

  - *Startup Frame Indicator* - with a logical 1 value it indicates that it is a startup frame, being this a special frame only used in the startup process of the cluster by the so called coldstart nodes. In this case the Synch Frame Indicator must be also logical 1.

- *Frame ID* (11 bits) - defines the frame identifier, and with 11 bits this identifier can a have a value between 1 and 2047, being the ID 0 invalid. Each frame must have a unique ID value.

- *PL - Payload Length* (7 bits) - defines the Payload size in number of words, being a word equal to two bytes. The minimum value is 0 and the maximum is 127 (254 bytes). This value is fixed and identical for all the frames in the Static Segment. In the Dynamic Segment, these value can be different for messages and can also present distinct values in different cycles.

- *Header CRC* (11 bits) - is the 11 bit-field that contains the Cyclic Redundancy Check applied to the all other bits of the Header section. On reception, this CRC must be recalculated by the receivers to check the Header correctness.

- *CC - Cycle Count* (6 bits) - defines the current cycle number (value between 0 and 63).

The Payload segment, with a size between 0 and 254 bytes, follows the Header segment and is composed of the data to be transmitted. Depending on the *Payload Preamble bit* value, the first bytes of this segment can be used as a network management vector (0 to 12 bytes, only in the Static Segment) or the message ID (2 bytes, only in the Dynamic Segment), used by the receivers to filter messages and give it specific treatment (e.g. discard it based on ID). The last one is the Trailer Segment, which is 3 bytes long, containing a 24 bit Cyclic Redundancy Code, being this value computed over the two previous segments.

FlexRay uses Non-Return to Zero (NRZ) **coding** to transmit bits in the communication channel. Before transmission, the frame content is decomposed in bytes and afterwards converted in a bit stream, which is transmitted in the serial link. The frame transmission starts by sending the TSS (*Transmission Start Sequence*), imposing a low bit value for *gdTSSTransmiter*, immediately followed by a high bit termed FSS (*Frame Start Sequence*). Afterwards each byte of the Header, Payload and Trailer segments are sent preceded by the BSS (*Byte Start Sequence*), which is made of a Low plus a High bit, used for bit synchronization between transmitter and receivers. This way each byte is coded in 10 bits. Finally the frame transmission end is marked with the FES (*Frame End Sequence*), which is composed of one low and one high bit. This coding is represented in Figure 3.53. and 3.54

A frame transmitted in a static slot has the coding explained before and one transmitted in the Dynamic Segment follows exactly the same coding until the FES. After the FES a DTS (*Dynamic Trail Sequence*) follows, being composed of specific number of Low bits and terminated by a High bit, as seen in Figure 3.54. The number of bits in *DTS* must be chosen in order to the frame ending coincides with the mini-slot action point. In the receiver the inverse operation is applied by removing the inserted bits and the frame content is restored.

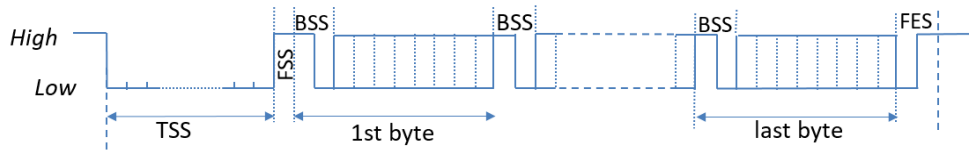

Figure 3.53: Frame transmission in the Static Segment.

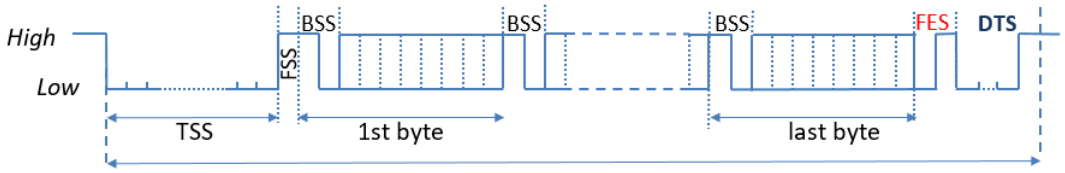

Figure 3.54: Frame transmission in the Dynamic Segment.

The **frame size**, meaning the number of bits in each frame, in the Static Segment, is then given by Equation (3.16) and a frame in the Dynamic Segment is increased by the number of bits in the DTS field, as in Equation 3.16.

$$\begin{aligned} frame\_Size &= n_{TSS} + n_{FSS} + Payload \cdot (n_{BSS} + 8) + n_{FES} \\ &= n_{TSS} + 1 + Payload \cdot 10 + 2 \\ &= n_{TSS} + Payload \cdot 10 + 3 \end{aligned} \tag{3.16}$$

$$\begin{aligned} frame\_Size &= n_{TSS} + n_{FSS} + Payload \cdot (n_{BSS} + 8) + n_{FES} + n_{DTS} \\ &= n_{TSS} + Payload \cdot 10 + 3 + n_{DTS} \end{aligned} \tag{3.17}$$

The transmission time of a frame obtained simply by multiplying the total number of bits by the bit duration, $gdBit$, that corresponds to the transmission time of one bit, e.g., 0.1 μs for 10 Mbit/s bit rate.

$$transmit\_time_{frame} = gdBit \cdot frame\_Size \tag{3.18}$$

To define the **slot size** - $gdStaticSlot$ - we must also take into account the clock imprecision and propagation delay times. So, each frame transmission starts at the action point inside the slot (see Figure 3.50), being this value defined using the clock precision and the estimated maximum delay in signal propagation. After the FES bits, an idle time must be present, being this time equal or bigger than the action point time [MGL+12].

Regarding **Clock Synchronization**, as the MAC mechanism uses a TDMA, is of utmost importance to guarantee that all communication controllers share a common global notion of time, with adequate precision. Each node has a local view of the global time, which must be continuously corrected to maintain the global clock synchronization. It uses the Fault Tolerant Midpoint Algorithm [WL88] that is used to calculate the offset correction value, and adjust the node clocks.

3.3.3.3.2 Fault Tolerance and Dual Bus configuration For critical messages, a dual bus configuration can be used to increment fault-tolerance, by sending the same message in both channels in the same slot. If the messages sent in the channels differ, then the second channel provides more bandwidth, allowing in the limit to duplicate it. Any intermediate solution is also possible.

3.4 Summary

This chapter presented the details on CAN, Ethernet and derivatives, TTCAN, FTT-CAN and Flexray protocols, that are essential in following chapters.

Chapter 4

Error Recovery in TT Systems

The approaches used to obtain fault tolerance in TT systems are various, being several ones described next. Fault tolerance can be obtained with time redundancy, spatial redundancy or both, and each one with different arrangements. Depending on the type of redundancy use, it is possible to cope with temporary and permanent faults, and obtain the intended reliability level.

It is assumed that the underlying communication network uses a shared medium, namely one or more buses.

4.1 Fault-tolerance with Hardware Redundancy

Fault-tolerance can be achieved using spatial redundancy, with hardware replication - nodes, buses or both.

4.1.1 Slave Replication with Single Bus

The transmission of messages can be done with replicated nodes, where each node tries to transmit one message, being the nodes referenced as primary and backup and messages termed primary and replica. The backup node tries to transmit the same message, with a small delay to guarantee that this node fails entering the arbitration process with the primary node. As depicted in Figure 4.1 in case of successful transmission of the primary message, the backup node aborts its transmission, which is the most common scenario. If an error occurred while trying to transmit the primary message, after error signalling, the backup node has a pending frame in the output buffer of the CAN controller, so it tries to transmits the replica message. To be able to use this mechanism in FTT-CAN, a slotted version [AP12] must be used. It is also assumed that single-shot mode is used by the CAN controllers.

There is a need to extend each slot duration to allow transmission of the replica message. As the error can happen at the end of the data frame, each slot must be extended by ($C_{MAX} + C_{error}$). If this mechanism is applied to all messages, then the necessary bandwidth would be more than doubled. Nevertheless, this mechanism can deal also with permanent node faults,

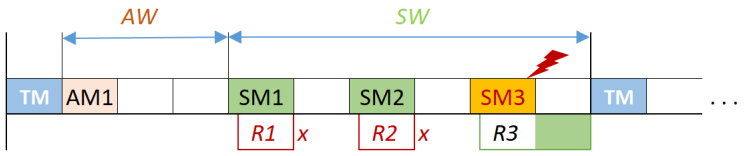

Figure 4.1: Using slave node redundancy, two scenarios: with replica transmission not necessary (aborted for messages SM_1 and SM_2) and replica message success (for SM_3).

which is not the case with other alternatives, since one node remains active in case of failure of the other one.

This approach would rise costs significantly, as each node has to be replicated. An undesired increased software complexity (to guarantee replica determinism) is also necessary and there is a significantly increase in bus bandwidth use.

A node with two (or more) embedded CAN controllers would alleviate cost concerns and software complexity, but the system would not be tolerant to permanent failures anymore. The two variants of slave replication (total and partial) are depicted in Figure 4.2, left side and right side, respectively.

Figure 4.2: Full and partial node replication

4.1.2 Bus Redundancy

Another alternative is to use replicated buses, with messages (primary and replicas) always transmitted in each bus. In the FTT-CAN scope this can be achieved using the proposal in [SF06], as briefly described in Section 3.3.2.3.6 and represented again in Figure 4.3, with two buses. The messages with transmission reliability requirements are transmitted in both buses, being a successful transmission obtained if at least one of the messages replicas is delivered to the recipient node. This method is represented in Figure 4.4, where the simultaneous transmissions of messages in both buses allows correct reception of message M_2, despite the error occurred in bus_1 (in this picture only the messages in the SW are shown). This scheme can cope with bus permanent failures, which is not the case of the previous approach.

4.1.3 Bus and Node Redundancy

Finally, to cope with simultaneous permanent bus and node failures, the previous replication schemes can be made even more robust if a combination of node and bus redundancy is

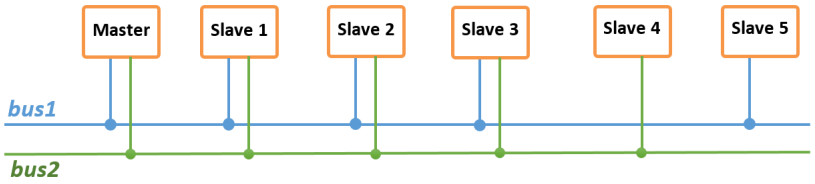

Figure 4.3: Bus redundancy in FTT-CAN (with two buses).

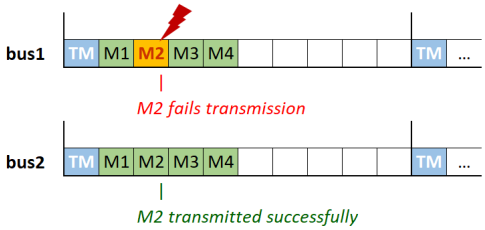

Figure 4.4: Using bus redundancy, transmission success in bus_2 for message M_2.

used, as for instance is presented for FlexCAN architecture [PF04], which share some common design characteristics with FTT-CAN.

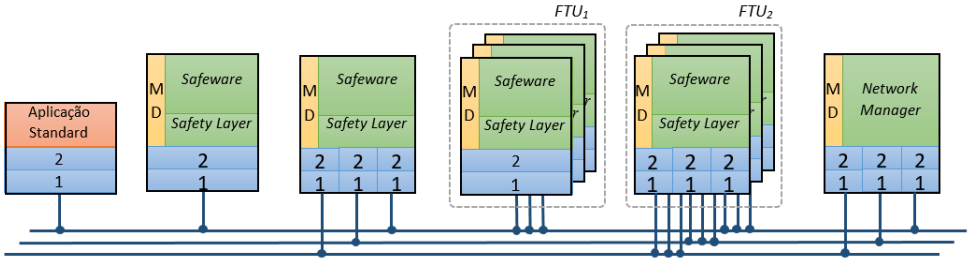

Figure 4.5: FlexCAN architecture - possible node configuration.

4.2 Fault-tolerance with Temporal Redundancy

4.2.1 Real-Time Event Channels in CAN

The work in [KCM05] presents the COSMIC middleware applied to a Time-Triggered CAN network, implementing a TDMA access scheme. It shows a method to recover errors in real-time messages. These messages are sent in dedicated offline-scheduled slots that are enlarged to allow for retransmissions using the CAN native error recovery method. This is represented in Figure 4.6. The slot enlargement is then dependent on the number of faults foreseen by the fault model. For instance, if the fault model previews two possible errors when a message is transmitted, then all messages protected by retransmissions should have a reserved time slot equal to the duration of 3 messages. This corresponds to tripling the necessary bandwidth when compared with no error recovery.

This method is still inflexible, since the slots are scheduled at pre-runtime. Nevertheless, at runtime, the bandwidth assigned to retransmissions but not effectively used can be re-

claimed to carry sporadic and non-real-time traffic, to allow for a more efficient bandwidth utilization. This is done simply by assigning higher CAN IDs (lower priority) to event-triggered or sporadic messages.

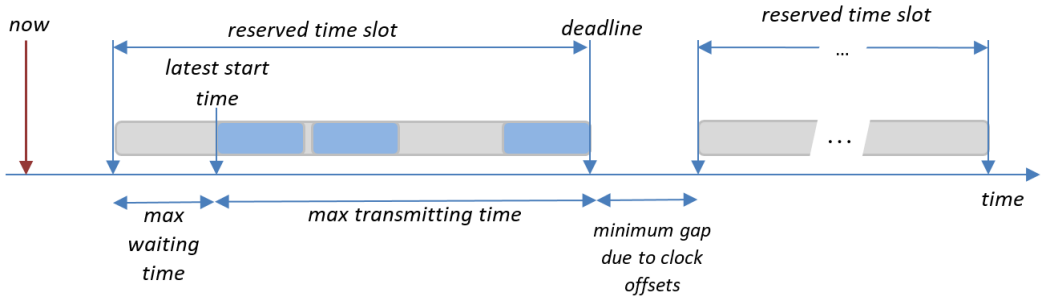

Figure 4.6: Error recovery in TT window (adapted from [KCM05]).

4.2.2 Message Retransmission and Acknowledgement in FlexRay

The work presented by Li et al [LNZ+09] was the first one that targeted schedulability in FlexRay networks, having fault tolerance as a goal. The proposal tackles fault tolerance aspects by introducing an acknowledgment and retransmission scheme, defined in the Static Segment and that works at the application level. A previously defined static schedule, built without fault-tolerance in mind, acts as starting point. Then, unused static slots and also available time in assigned slots are configured to be used to transmit acknowledgments and to retransmit frames found with errors or simply omitted.

The fault recovery rate is defined as the percentage of faulty messages guaranteed to be retransmitted before their deadlines.

Using Mixed Integer Linear Programming optimization to minimize the extra bandwidth, the experiments reported there show a case in which about 50% of the messages can be recovered. The results also showed that, for the same system, as used bandwidth increases, thus with less available empty slots, the efficiency of the mechanism drops and the authors conjecture that better results can be obtained if the proposed mechanism is optimized with the initial scheduling process.

The limitation of this proposal is on the achievable reliability levels and its specific application to already defined schedules.

4.2.3 Message Replication in the Static Segment of FlexRay

In FlexRay networks the TT traffic is allocated statically to slots in the Static Segment (SS). One way to guarantee the reliability in message transmission is to send several copies of the same message proactively, per message period. Tanasa et al. [TBEP10] propose a method to recover errors in the SS that basically defines the number of copies of each message that must be sent to obtain a global success probability (GP) that must be greater than the intended system reliability, for a given mission time. The proposed method can be described by Figure 4.7.

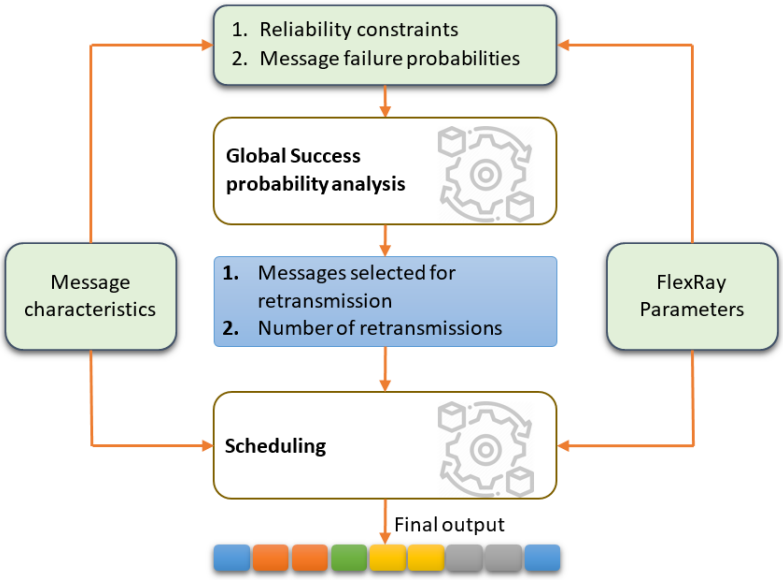

Figure 4.7: Proposed method (adapted from [TBEP10]).

First the reliability constraints are defined. The reliability goal ρ, is defined by Equation (4.1), that is constrained by γ, which corresponds to the maximum probability of a system failure in the mission time.

$$\rho = 1 - \gamma \tag{4.1}$$

With a known BER, the probability of failure of message i is given by equation (4.2), where W_i is the number of transmitted bits of message m_i in the Static Segment.

$$p_i = 1 - (1 - BER)^{W_i} \tag{4.2}$$

If more than one copy per period is sent, then a transmission failure can only occur if the original message and all its k_i copies fail. So the probability of transmission failure of message i is given by Equation (4.3).

$$PF_i(k_i) = p_i^{k_i+1} \tag{4.3}$$

Conversely, the probability of at least one instance of the message is successfully transmitted is given by the complementary of the previous equation, as in Equation (4.4).

$$PF_{i\_success}(k_i) = 1 - p_i^{k_i+1} \tag{4.4}$$

Considering a mission time MT, the message m_i occurs $\frac{MT}{T_i}$ times, so Equation (4.4) should be iterated for all instances, leading to the success probability of message m_i, as in Equation (4.5).

$$PS_i(k_i) = \left(1 - p_i^{k_i+1}\right)^{\frac{MT}{T_i}} \tag{4.5}$$

For the message set that is composed of N elements, all messages have to be considered, along the mission time (all instances). The global success probability can be obtained, being this termed GP, as in Equation (4.6).

$$GP = \prod_{i=1}^{N} PS_i(k_i) = \prod_{i=1}^{N} \left(1 - p_i^{k_i+1}\right)^{\frac{MT}{T_i}} \tag{4.6}$$

4.2.3.1 CLP Formulation

The previous formula to obtain GP assumes that the number of retransmissions for each message, k_i, is known. So the question that must be firstly answered is what should these values be? The authors propose that the k_i's values can be obtained by formulating an optimization problem in Constraint Logic Programming (CLP), that will find an optimal solution.

The constraints are the following:

The Reliability Constraint in Equation (4.7) that basically states that the goal to attain should be greater than the intended reliability.

$$GP = \prod_{i=1}^{N} \left(1 - p_i^{k_i+1}\right)^{\frac{MT}{T_i}} > \rho \tag{4.7}$$

The minimum number of retransmissions constraint in Equation (4.8), where k_i^L is the smallest value that satisfies Equation (4.9). Since $PS_i(k_i)$ is a sub-unitary value, when calculating the GP, which is the product of all PS_i's, to obtain at least a reliability greater than ρ, then all individual values must be greater ρ.

$$k_i \geq k_i^L \tag{4.8}$$

$$PS_i(k_i) \geq \rho \tag{4.9}$$

The protocol also places a limit on available resources, namely the maximum number of slots, as in Equation (4.10) and the Scheduling Constraints in Equation (4.12).

$$\sum_{i=1}^{N} (k_i + 1) \leq NS \tag{4.10}$$

The slot function takes a message as parameter and returns the slot assigned to it. The domain for this function, represented by D, is thus, the set of messages including the n_i instances in the hyperperiod H, as defined in Equation (4.11), and their retransmissions. It follows that the constraints for the domain D are then given by Equations (4.12) and (4.13).

$$H = lcm\{T_1, T_2, \ldots, T_n, FC\} \tag{4.11}$$

$$\begin{cases} D_i = \{M_i^{j,1}, M_i^{j,2}, \ldots, M_i^{j,k_i+1}\}, \forall_{1 \leq j \leq n_i} \\ D = D_1 \cup D_1 \cup \ldots \cup D_N \end{cases} \tag{4.12}$$

and

$$\begin{cases} slot : D \longrightarrow \{1, 2, \ldots, NC\} \times \{1, 2, \ldots, NS\} \\ slot(M_i^{j,l}) = (c_i^{j,l}, s_i^{j,l}) \end{cases} \tag{4.13}$$

Other constraints are defined regarding deadlines, instant of message production and finish and also slot occupations closes the list of scheduling constraints.

4.2.3.2 Optimization objective

The objective of the CLP optimization is to minimize the total number of used slots in the FlexRay cycle, FC, or equivalently, minimize the bandwidth use to guarantee the reliability level.

$$minimize : \sum_{i=1}^{N} (k_i + 1) \tag{4.14}$$

4.2.4 Heuristic Approach

The author's state that the CLP method is computing intensive and does not scale, so they propose in this same article to use heuristics to overcome these limitations. The heuristic approach is depicted in Figure 4.8 and can be described as follows:

- Stage I: compute k_i values, using the reliability constraints;
- Stage II: generate the schedule, using the constraints defined by CLP;
- Stage III, is necessary if stage II does not find a feasible schedule.

The main results of this article are presented to show that the CLP method is correct and can generate synthesized schedules that meet the reliability goal. Also, the alternative heuristic methods are also effective in generating a feasible schedule, being more computationally efficient, but might be sub-optimal.

Little information was given on the k_i's values for the tested message sets. Nevertheless, in one of the sets they show that using messages with k_i equal to 1 and 2, that means that the necessary bandwidth to achieve a transmission reliability equal to $(1-10^{-5})$, in a mission time of one hour, was more than doubled, using a BER equal to 10^{-7}.

The authors have proposed the fault-tolerant approach tailored to the Static Segment of FlexRay networks, but this approach can be applied to any network that uses static scheduling and TDMA access, as is for instance TTP/C and TTCAN. To achieve this, it is necessary to update namely the schedule generating module, as the constraints are different for each protocol.

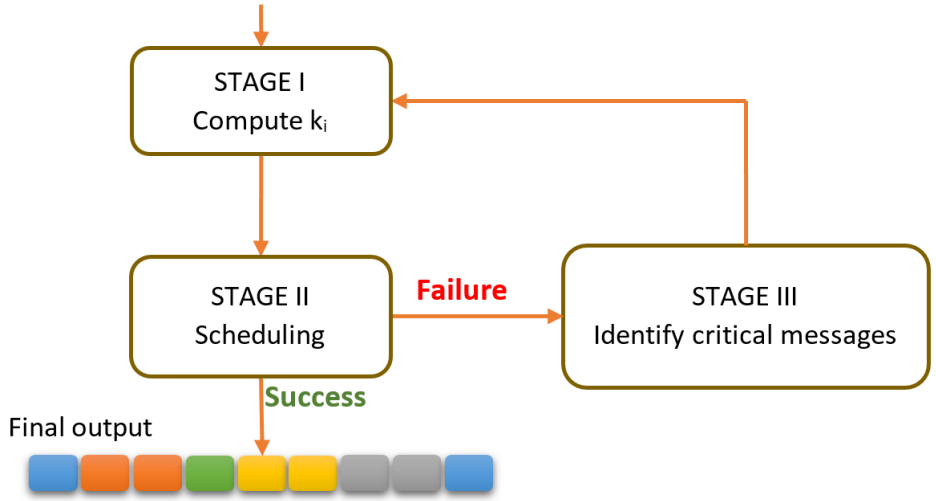

Figure 4.8: Proposed heuristic (adapted from [TBEP10]).

4.2.5 Windowed Transmission in TDMA CAN

The idea behind the concept of windowed transmission enlargement/reduction is depicted in Figure 4.9. In the top timeline the classic temporal replication scheme with two slots for transmitting 2 copies of a message is presented, in this case both are successful. In the middle timeline two error occur and due to slotted arrangement and single shot transmission, the message suffers a transmission failure. In the bottom timeline the slots are merged with retransmissions allowed and considering the same faults as in center timeline a success is attained. The authors suppose then that is possible to use smaller merged slots and attain the same reliability goal.

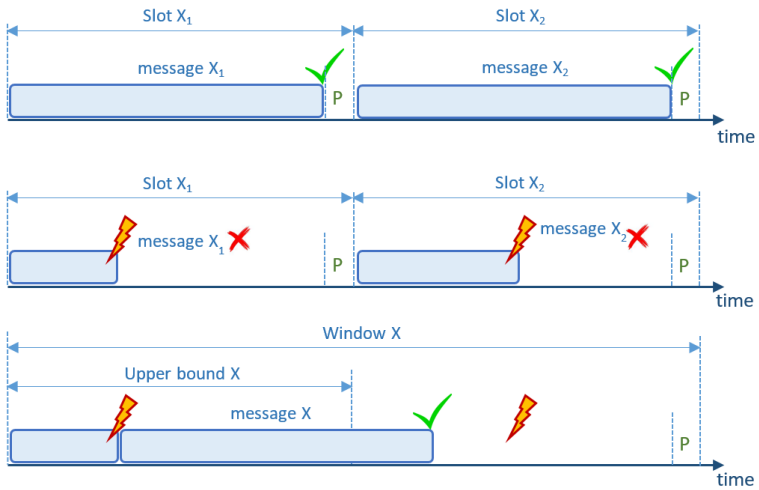

Figure 4.9: Short windowed transmission concept (adapted from [SS10]).

The recovery mechanism is presented by Short et al in [SS10], to guarantee the message transmission in TDMA based CAN networks. Messages are transmitted in specific windows or slots, which are enlarged to make room for CAN native automatic message retransmissions

upon errors, but only if the retransmissions fit in the defined window. The paper presents a method to calculate the window size together with a simulation study that points to a bandwidth utilization reduction between 3% and 30%, on average, depending on the environment type (from Benign/Normal to Aggressive/Hostile, respectively), when compared with the case where a predefined number of message copies is sent to attain the same reliability level. The article includes a test case where the mechanism is applied to a critical message with 8 bytes payload, a period of 100 ms and an intended transmission reliability of 10^{-9} errors per hour and using an Aggressive environment with BER equal to $2.6 \cdot 10^{-7}$, showing that using the windowed method the intended reliability level is obtained with a window 26.4% shorter than using multiple copies, which in this case, implies the transmission of four copies.

The implementation of this mechanism implies building specific hardware (FPGA based) for the nodes and is still inflexible in what concerns modifying the message set dynamically, as the schedule is built offline. The use of special hardware, makes it not compatible with COTS CAN controllers, which is a negative point of this proposal.

4.2.6 Temporal Replicas in FTT-CAN - Locally Controlled

In FTT-CAN networks to cope with transient message transmission failures, time redundancy can be used. Different schemes can be devised, depending on where inside the EC (in what window), who controls retransmission (central or local) and also if done on-demand (on error detection).

4.2.6.1 Retransmission in the Asynchronous Window by the Sender

In FTT-CAN the existence of the Asynchronous Window, opens the opportunity for message retransmission placing. So, a node detecting an error in one of its synchronous messages can perform message retransmission in the EC following the one where the error occurred, inside the AW. There is a guarantee that the message is effectively retransmitted in the AW, being the first one transmitted, as the rule for CAN IDs choosing for messages, in FTT systems, always assigns higher priorities (lower values) to the synchronous messages. This method is graphically depicted in Figure 4.10, where message SM_3 suffers an error and is retransmitted in the AW of the next EC, before any asynchronous message.

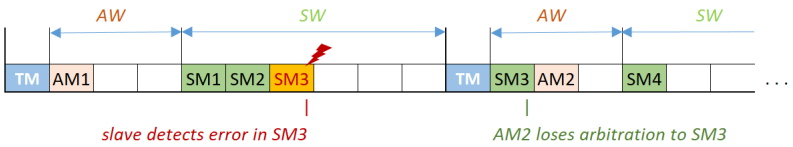

Figure 4.10: Error recovery process in the Asynchronous Window.

In any case, there will be interference on the asynchronous traffic, being this limited by the maximum number of errors occurring per EC.

This approach, despite interesting in its working principle, breaks the timing isolation between synchronous and asynchronous traffic, removing also from the Master sphere of

control the triggering of the synchronous messages that need to be retransmitted.

4.2.7 FTT-CAN Static Error Recovery

Using the native error recovery of CAN (automatic retransmission in case of error), slack time is reserved in all SW to allow the retransmission of recovery messages [Fer05], including also a maximum error frame (equal to 23 bits) per retransmitted message. This corresponds to an automatic retransmission by the Sender in the SW.

This approach gives very fast recovery time, as the error is recovered immediately, in the same cycle where it occurred, contrary to previous approaches, where the recovery only happens in the following cycle.

It is also more bandwidth efficient than for instance the one presented in Section 4.2.1, since the extra-time allocated in the synchronous window is shared by all the TT messages sent in the current EC. The amount of slack time is dependent on the maximum number of errors considered in each EC and is always wasted when there are no errors, thus limiting the efficiency of the approach.

This mechanism will not succeed in case of temporary failure in the node (controller or transceiver) that prevents it from transmitting the message (message is absent in the bus). As the Master has no intervention in the recovery process, he is unaware of this failure, and does not take any further measures to recover the missing message, even if there is still time available before the deadline, leading to an effective message transmission failure.

The reserved slack also puts a limit on available time for the SW, the *LSWmax*. Then considering m_errors the maximum number of errors previewed by the fault model, Equation (4.15) gives the slack time that must be reserved in each SW. This time is completely wasted when there are no errors, and cannot be reclaimed by other messages.

$$slack\_SW = \big(C_{MAX} + C_{error}\big) \cdot m\_errors = (C_{MAX} + 23) \cdot m\_errors \tag{4.15}$$

Table 4.1 gives the largest SW possible, for various possible values of *LEC* and considering *LTM* with duration equal the longest CAN frame (payload equal to 8 bytes).

Table 4.1: Available time for SW (bit rate = 1 Mbps), as a % of LEC

	LEC		
m_{errors}	2.5	5	10
0	94.6%	97.3%	98.7%
1	88.3%	94.1%	97.1%
2	82.0%	91.0%	95.5%
3	75.6%	87.8%	93.9%
4	69.3%	84.7%	92.3%

As can be observed the penalty on the available time for the SW is very high for small values of LEC, specially with more expected errors, reducing as LEC grows.

4.3 Summary

In this chapter several methods were described for error recovery in the time domain, for TT networks. Bandwidth and latency were discussed and compared, along with bandwidth efficiency.

Chapter 5

Error Recovery in FTT-CAN - Dynamic Approach

This chapter details a proposal to obtain adequate reliability levels of the message transmission subsystem, in the scope of FTT-CAN networks. It starts by describing the rational behind the proposal, that uses time-domain redundancy to obtain fault-tolerance with adequate reliability level, using online scheduling and a retransmission server, attaining higher bandwidth efficiency when compared with state-of-the-art approaches.

The synchronous message transmissions, in the FTT-CAN protocol, are centrally controlled by a Master node, on a EC-by-EC basis, using an online scheduler with any scheduling policy. By inserting a new module that detects message transmission errors it is possible to include the message retransmissions requests in the scheduling process. The message retransmission will be managed by a Server (a new module), being the server type, capacity and period defined according the fault model used and the desired reliability target.

Firstly it is introduced the single replica version of the error recovery method. This approach assumes that the fault model is restricted to a maximum of one error per EC and also that the server permits a single replica, allowing this way to maintain full compatibility with FTT software in the slave nodes. With these two restrictions, an analytic model for obtaining worst case response time of synchronous messages is presented, including the identification of limit error and recovery scenarios. Other aspect analyzed are the different politics implemented by the server and possible implications in terms of interference and message timeliness.

After identifying limitations on the achievable reliability target posed by the previous approach, due to single replica retransmission, using the Poisson fault model (with the one error per EC restriction removed), new error and recovery scenarios are identified that permit the definition of the minimum number of replicas to send per error detected, to attain a predefined reliability target. Then, including multiple replica transmission, a new formulation for response time calculation is obtained, together with new server characteristics (namely the updated server capacity). The implementation of this new recovery method implies that the FTT software in the slave nodes must be modified.

5.1 ReScheduling by the Master

In the previous chapter several methods to achieve fault-tolerance were described, which use diverse forms of redundancy.

When using hardware or spatial redundancy - nodes, buses or both - the software complexity (e.g. to guarantee replica determinism), the energy consumption and specially the cost rise. Nevertheless, this kind of redundancy is mandatory if we must tolerate permanent errors, either in the nodes or in the buses. Due to the prevalence of transient errors versus permanent ones, with a ratio greater than 100 [PKOS04], our aim is at first to recover errors of transient nature. Anyway, if permanent errors must also be tolerated, our proposal can be complemented with spatial redundancy techniques, as the ones presented in the previous chapter.

The other possibility is to use time redundancy, by transmitting more messages in the time domain than the strictly necessary if there were no errors. In a TT system, this kind of redundancy can be applied in proactive or in reactive mode. The proactive mode implies sending several replicas of each message instance, without the factual knowledge of error occurrence. This is typically the approach used in classic TDMA systems, where the schedule is static and obtained off-line. This implies that the used bandwidth will grow significantly (e.g., with two replicas per message the necessary BW doubles) and that due to slot reservation, the allocated BW to message replica transmission is wasted when there are no errors, as no other messages can use these static slots.

A better use of the available bandwidth is possible if the slots/windows used for replica transmission are shared by a set or all messages. In this case, the slot/window duration is proportional to the expected number of errors in the communication cycle, being statically allocated per cycle. The retransmission is only triggered on message error detection, so the protocol must possess a mechanism for error signalling. This approach is the one used in FTT-CAN by reserving slack time in the Synchronous Window [Fer05], but can also be used in TTCAN by assigning an arbitration window to message retransmissions. The BW waste is smaller than in classic TT systems, but again is completely lost in no-error scenarios.

An even better bandwidth efficiency can be attained if there is no static allocation of slots/windows and all the available bandwidth is managed altogether for all messages - normal and retransmissions. This can be achieved using an online scheduler that has as input a list of failed message transmissions. Then the scheduler will integrate the message retransmission requests, managed by a server, with normal messages in the scheduling process. The scheduler must also guarantee the message timely delivery with an adequate reliability level. This can be done with FTT-CAN, including new modules in the master node, as described in the following paragraphs.

In the FTT-CAN protocol it is the Master node that controls the EC where each synchronous message is transmitted. This node implements a message scheduler and transmits the Trigger Message (TM), which instructs the slave nodes to send the corresponding messages. By listening to all the transmitted messages in the SW, in each cycle, and recording

which messages were successfully transmitted, it can, by comparing to the scheduled messages for each EC with the ones correctly received, determine the messages that have failed transmission. Using this information, these messages can re-enter the scheduler and will be scheduled together with the remaining messages, being recovered in a latter cycle.

The central control by the Master node has other advantages, as for instance allowing different recovery levels for distinct messages, where some messages are subject to the recovery process and others are simply ignored (no recovery for non-critical messages, for instance) or limit the number of recovery attempts to limit the bandwidth used by the recovery process, inducing this way less interference in the remaining traffic (or the use of less resources).

This corresponds to a dynamic approach, which is in contrast to the passive approach of simply reserving slack time in the SW to allow for message retransmissions, as proposed in [Fer05], using the CAN native error recovery mechanism.

This method corresponds to the one proposed in this thesis, that will be further detailed in the following sections.

5.2 Error Recovery in the Time Domain - Single Replica Version

As already stated, in FTT-CAN networks the master node is responsible for all the scheduling decisions concerning the synchronous traffic, which are then disseminated to the slave nodes by the Trigger Message, in each cycle. Since CAN uses a broadcast bus, configuring the CAN controller filter in the master node to accept all messages sent by other nodes, makes it a listener of all messages. So, by listening to all bus traffic, the master may build a list of all successfully transmitted messages. Then, at the end of the synchronous window, a Bus Error Detector block compares the list of scheduled messages with the messages actually received in that EC, thus identifying eventual errors and omissions. The IDs of such messages are then put in the Error Server Queue. Afterwards, the scheduling for the next EC is performed, considering both the active synchronous messages (described in the SRDB) and the messages affected by errors, contained in the Error Server Queue. This process is shown in Figure 5.1, where it can be seen the introduction of the two additional blocks in the new version of the master.

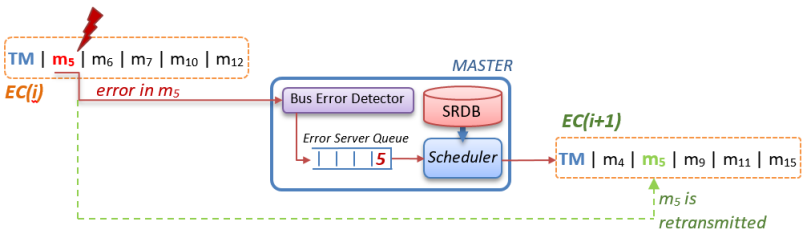

Figure 5.1: Rescheduling by the Master.

The failed messages in the error queue are managed by a server, that is defined by its

policy, capacity and period. Of course, the server policy must be compatible with the synchronous traffic scheduler.

For compatibility reasons, to use the slaves without changing the protocol stack, it is mandatory that the coding of the TM remains the same (as described in Section 3.3.2.3.1), since the slaves are instructed by the TM on what messages they should send in the current EC. This limits the number of copies of a particular message to be sent per cycle to one, since the TM encodes in only one bit the identifier of each message that has to be transmitted (see Figure 3.37). This was an important restriction, as we want to introduce the fault tolerant mechanism with minimal (or no) updates to the FTT stack software in the slave nodes.

Depending on implementation details, namely where, inside the EC, the scheduler is run (due for instance to hardware limited processing power, as is the case of 8 bit microcontrollers), the recovery latency can vary, with minimum value equal to one EC. In Figure 5.1 we have considered the fastest implementation approach, that presents a latency equal to one EC. This topic will be further discussed in Section 6.5.2.

5.2.1 Motivational Example

In this example we assume that the arrival of faults can be modelled by a Poisson process with arrival rate equal to λ and restricting the number of faults per EC to one, due to small values of elementary cycle length and even smaller fault probabilities.

The system has the message set described in Table 5.1, where the messages with lower identifier have higher priority, C_i and T_i are the transmission time and period of message i, in time units, being U the message utilization value. Messages are recovered by a Deferrable Server with maximum priority assigned, capacity equal to 4 maximum size messages and period equal to 4 ECs. The issues related with the server type, capacity and period will be discussed later on.

Table 5.1: Message set used in following examples

i	C_i	T_i	$T_i(EC)$	U
1	1	8	2	12.500%
2	1	8	2	12.500%
3	1	12	3	8.333%
4..13	1	24	6	10*4.167%
				75.0%

Figure 5.2 depicts several scenarios, without and with errors, in the message set hyperperiod, where the first timeline (top), shows the sequence of message transmissions per EC, without errors and assuming a rate-monotonic policy scheduler. This corresponds to a base scenario, that will be used in comparisons with other scenarios that present errors. In this figure the numbers on top indicate when the messages become ready and need to be transmitted, being sent in the following ECs subject to the available capacity and according to

their priority.

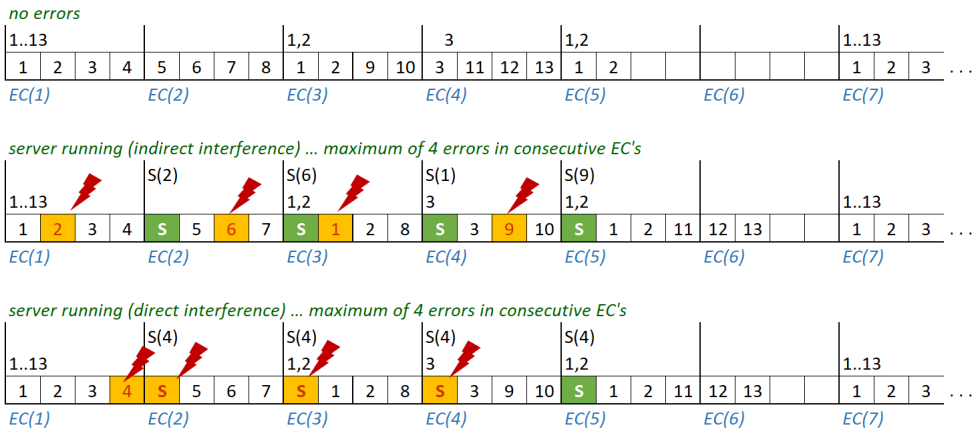

Figure 5.2: No Error, Indirect and Direct example interference scenarios.

The second timeline illustrates the behaviour of the error recovering process, considering that we are interested in obtaining the response time of message m_{12}, that corresponds to one message that does not suffer any error. The lightning bolt symbol represents a fault in the bus and the **S** letter a server activation on behalf of the message with its number indicated between parentheses. At last, a yellow background means that the message suffers an error. Comparing with the no-error scenario it is easily observable the effect of server execution, on behalf of different messages, that pushes message m_{12} transmission to EC(6), delaying its transmission 2 ECs, increasing its response time to 6 ECs against 4 ECs in the no error scenario. Other messages, also not hit by errors, suffer different interference, depending on its priority. For instance, message m_8 increases its response time from 2 to 3 ECs. On the other hand, messages m_1, m_2 and m_3, when not hit by errors, present the same response time, as the server execution does not push none of them to latter ECs, as they still fit in the EC where they were scheduled.

When determining the response time of a particular message and considering that no error hits it, we term this the additional scheduling latency due to the server execution as **Indirect Interference**.

A different interference is perceived by the messages when hit by errors. For instance consider message m_2 in the second timeline. We observe that this message (or m_6 for instance) is scheduled in the same EC as in the no-error scenario and we also observe that it is successfully recovered in the following cycle (remember that server uses maximum priority). This does not correspond to the worst case scenario, as with errors in consecutive ECs the message recovery can be further delayed if the retransmission is also hit by additional errors. This scenario is depicted in the bottom timeline of Figure 5.2, where it is clear that message m_4 and the retransmissions, represented by S(4), are sucessively delayed to the next cycle, resulting in a response time equal to 5 ECs, instead of only one in the no-error scenario.

When computing the response time of a particular message and considering that this message is hit by an an initial error and all recovery tries were also hit by additional errors,

then we call this a **Direct Interference** scenario.

5.2.2 The Recovery Server

A message transmission error occurrence is an event that can be regarded by the scheduling process as an aperiodic request to retransmit that message. This request must be somehow answered in conjunction with all other periodic messages requests.

So, as message retransmissions require bus time, it surely has an impact on the response time of the remaining messages. As the assumed fault arrival model is random by nature, being the instants of errors occurrence unpredictable, it is necessary to bound the interference of retransmitted messages, at least when message timeliness guarantees are a system requirement.

The question that arises then is how to deal with such events and still guarantee the timeliness of all messages, including the one that has to be retransmitted. We decided to use a server to manage the retransmissions because, in addition to the functionality to bound the interference of retransmitted messages, a server:

- is resource-efficient, since it consumes bandwidth only when activated, i.e., in the presence of errors;
- allows controlling the reactivity to errors via its associated priority and budget/period;
- is predictable and analyzable.

5.2.2.1 Server Type

The system scheduling policy uses fixed priorities, being the aperiodic server types and characteristics already described in Section 2.5. Due to their simple implementation an low execution overhead the two obvious candidates were the Polling Server and the Deferrable Server. The reactivity of the Polling Server is closely tied with its period, since its executions are scheduled independently of the occurrence of errors. As such, in the aperiodic case, the error response time may approach two server periods, even when the server has full capacity. As the assumed fault model implies that an error can occur anywhere, to guarantee a prompt recovery the server must be activated as soon as possible (in the next cycle). This is essential to messages with short deadlines, or else these messages will loose its deadline while waiting in the error server queue, even if there is slack time in the schedule. Considering, e.g., a system with a SW that can transmit 10 maximum messages, the BW use of the Polling Server, that must be configured to have T_S equal to 1 EC to ensure a response time of one EC, would be equivalent to one message per EC (remember that we are considering a maximum of one error per EC) or 10% of the available BW. Note that the server period is independent of the average error rate or even the considered bound on maximum number of errors per server period.

On the other hand, the Deferrable Server can be activated anywhere in time, as long as there is available capacity. This way, the server capacity can be configured based on the

maximum error assumption that may happen in the server period. This characteristic allows allocating much lower bandwidth to the server. As an example, considering a maximum of 10 errors per 1000 ECs, the Deferrable Server can then be configured to recover 10 errors in this period, which corresponds to 0.1% of the available BW. This values compare favourably against the 10% that a Polling Server would require in the same circumstances. The simulation results presented in Section 6.2.2 confirm this observation.

Since the implementation cost is basically the same of the Polling Server, from now on, when referring to the recovery server, the Deferrable type is the one considered. There is, however, a potential penalty in terms of schedulability, due to back-to-back execution.

5.2.2.2 Obtaining Server Capacity and Period

After choosing the server type, as described previously, two other server key parameters must be defined: capacity C_S and period T_S. These two parameters are intertwined and together define the server allocated bandwidth, that should be as small as possible.

The recovery mechanism implies that when an error is detected by the Master node, a retransmission can only occur if there is sufficient remaining capacity in the server. If not, the retransmission will be put on hold, until the replenishment at the start of next server period. In this case, delaying the recovery try increases the probability of missing deadlines, which can became too high, specially in the case of messages with short deadlines.

So, to be effective, the capacity assigned to the server must be at least equal to the maximum expected number of errors that can happen in each server period. This way, when an error is detected a prompt recovery try can be performed.

As a Poisson process is used to model the error arrival, we can use Equation (5.1) to compute the probability of having n or more faults in a time interval equal to τ. For the server we are interested in a bound for the expected number of errors in its period, with adequate probability assurance. In fact, it is preferable to use as system design metric the probability of non-recovery, in the server period, designated $p_{\epsilon s}$, which can be computed with Equation (5.2). The choosing of the $p_{\epsilon s}$ is closely related with the message transmission global reliability target that we intend to achieve. Resuming, we are saying that the probability of exhausting the server capacity, in one server period, must be lower than the defined threshold. In Figure 5.3 the probability $P_\lambda(\geq n_{errors}; T_S)$ is plotted for several values of T_S, with $T_S = \alpha \cdot (1/\lambda)$ and $\alpha = 0.25, 0.5, 1$ and 2, always for an Aggressive environment (Table 2.1). The α parameter is a value that permits to adjust the period in multiples/fractions of the $1/\lambda$ reference value.

$$P_\lambda(\geq n; \tau) = \sum_{k=n}^{\infty} e^{-\lambda\tau} \frac{(\lambda\tau)^k}{k!} \tag{5.1}$$

Based on this equation, it is then possible to determine the number of errors that must be accommodated by the error-handling mechanism for a particular period T_S, and from this, compute the minimum server capacity for its specified period T_S (as presented in [MVPA13]). The server capacity is then equal to the minimum n_{errors} value that verifies Equation (5.2),

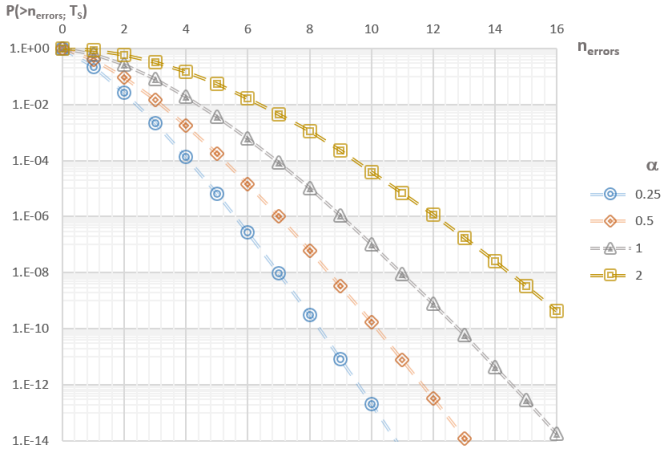

Figure 5.3: Probability of finding k or more errors, function of server period.

when single retransmission replica is considered.

$$P_\lambda(\geq n_{errors}; T_S) \leq p_{\epsilon s} \tag{5.2}$$

For example, looking at the graph in Figure 5.3, considering $\alpha = 1$, ($T_S = 1/\lambda$), and a target $p_{\epsilon s} = 10^{-7}$, the minimum value of n_{errors} that satisfies Equation (5.2) is 10, so the server is configured with a capacity equal to 10 maximum messages. This represents a server bandwidth of only 0.035% of the total system bandwidth (considering $C_{MAX} = 135$ bits, $T_S = 1/\lambda = 3.85$ seconds and a 1000 kbps bit rate), in the worst case, which represents a negligible fraction of the available bandwidth. A quick check for the $T_S = 0.25 \cdot (1/\lambda)$ curve, shows that for the same $p_{\epsilon s}$, the server must be configured with C_S equal to 7, that represents a bandwidth of 0.098% of the total available BW. If the server period is further reduced we observe that the server allocated BW grows, so there is no advantage in attributing smaller values to T_S, as the reactivity to errors is granted by the server type chosen (Deferrable type).

So, choosing a server period equal to $1/\lambda$, where λ is the average error arrival rate, giving an average of one error recovery per T_S, seems an appropriate choice.

Note that the server allocated BW for T_S's of the order presented in Figure 5.3 always represents a negligible fraction of the available bandwidth. Nevertheless, as we will see in the following section, the interference caused by the server execution on the scheduling of the remaining messages is non-negligible, and must be accounted for in the schedulability analysis.

5.2.3 Message Response Time

In this section we will derive equations to calculate the response time of messages, considering Indirect and Direct Interference.

5.2.3.1 Message Response Time with Indirect Interference

Figure 5.4 shows four scenarios with increasing number of errors. The first timeline is the scenario without errors, to establish the base case. The used message set is the one described in Table 5.1.

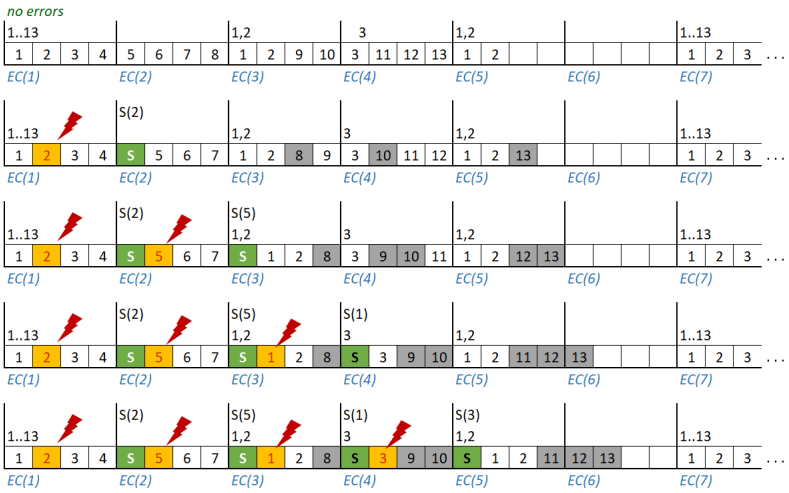

Figure 5.4: Example scenarios for indirect interference, with 1 to 4 errors.

The timeline with one error shows that messages m_8, m_{10} and m_{13} (in gray background) suffer a delay of one EC due to server execution on behalf of message m_2. All other messages are not delayed, in spite of server execution. As more errors are considered, in the following timelines, more messages are delayed, being observable that in the fourth timeline (with 3 errors) there is one message (m_{13}) that is already delayed 2 ECs, meaning a worst case response time increased by this value. In the bottom timeline (4 errors in consecutive ECs), message m_8, m_9, m_{10} and m_{11} have response times increased by one EC and m_{12} and m_{13} by two ECs. We also note that there are messages that, despite the interference imposed by the server execution, do not get an increase in their response time, e.g. m_1, m_2, m_3. So, the interference caused by the server execution increases with the number of errors (and recoveries), being the response time increase also function of message priority.

To obtain the response time we consider a message set M, extended with a Deferrable Server, with capacity equal to N_S maximum size messages and period equal to $1/\lambda$. We assume that its capacity is never exhausted in each server period, being configured as explained in section 5.2.2.2. The number of errors is limited by max_{errors} within the server period and at most one error per EC. Then, by applying the Non-Preemptive Blocking-Free scheduling model [AF01], all the execution times are inflated, as in Equation (5.3), represented by the E superscript.

$$C_i^E = \frac{LEC}{LSW - C_{MAX}} \cdot C_i \tag{5.3}$$

This allows the use of classic Response Time Analysis of preemptive systems [ABR+93]. The interference of one message per EC, resulting from server execution to recover the error

in previous EC and also error signalling, results in Equation (5.4),

$$R_i^{n+1} = C_i^E + \sum_{j=1}^{EC_{number}(R_i^n)} \left(C_{MAX}^E + C_{error}^E \right) + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^n}{T_k} \right\rceil \cdot C_k^E \tag{5.4}$$

where the $EC_{number}(R_i^n)$ is limited by the maximum number of errors considered by the fault model (max_{errors}), and is given by Equation (5.5).

$$EC_{number}(R_i^n) = \left\lceil \frac{R_i^n}{LEC} \right\rceil \tag{5.5}$$

The algorithm to perform the response time analysis with Indirect Interference in a message set M is *Algorithm IndirectInterf*.

ALGORITHM IndirectInterf
Message Response Time with Indirect Interference Calculation

Input: $M, S, LEC, LSW, max_{errors}$
Output: R^{EC} vector, Schedulable

1. $Schedulable$ = TRUE
2. Compute C_{MAX} in message set M
3. Compute S^E and M^E (inflate all transmission times using Eq. 5.3)
4. **for** $i = 1$ **to** N **do**
4.1 Compute R_i considering M^E and server interference $C_{MAX}^E + C_{error}^E$ added only once each EC, at most max_{errors} times (use Eq. 5.4)
4.2 **if** ($R_i > D_i$) **then**
mark message i as unschedulable
$Schedulable$ = FALSE
break
end_if
4.3 $R_i^{EC} = \lceil \frac{R_i}{LEC} \rceil$
end_for
5. **return** vector R^{EC}, $Schedulable$

The algorithm starts by assuming that the message set is schedulable, by assigning TRUE to the *Scedulable* flag. In Line 2, the longest message in the set is obtained, being then used in Line 3 to compute the inflated message set M^E and S^E according to Equation (5.3). In Line 4, the response time of each message is calculated using Equation (5.4), one at time, considering a total of max_{errors}, one per EC, being the number of messages N. A test for message schedulability is done inside this cycle, marking the message that violates its deadline (Line 4.2) and terminating the cycle. In Line 4.3 the response time is converted to an integer number of ECs, which is the system granularity. Finally in Line 5 the response time of all messages in the R^{EC} vector and a flag that signals schedulability of the message set is returned.

5.2.3.2 Message Response Time with Direct Interference

Figure 5.5 illustrates more clearly the direct interference scenarios, again with increasing number of errors (from one to four). We are only interested in the message that suffers transmission errors (message m_4 in this example). When more than one error is considered in the following cycles then the worst case scenario (for the message under analysis) is the one when retransmissions are consecutively affected by errors. Just by observing the various scenarios, it is obvious from all the timelines that each error induces a delay of one complete EC on the response time, so the successful transmission is delayed by max_{errors} ECs.

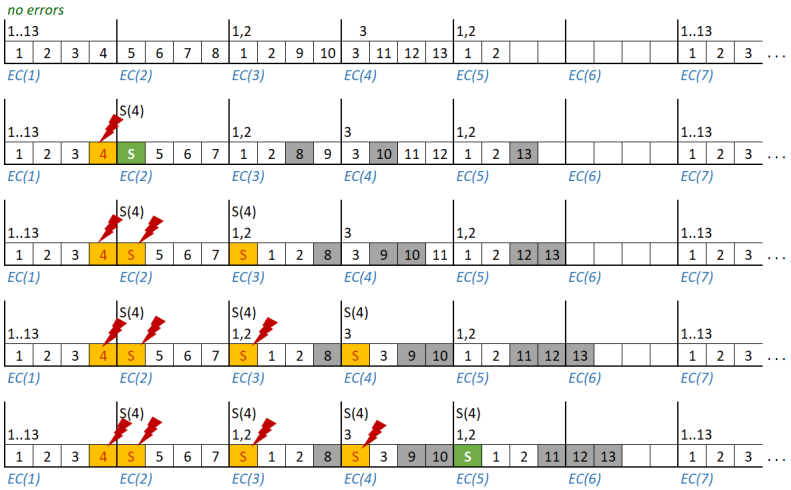

Figure 5.5: Example scenarios for response time calculation with Direct Interference, with increasing number of errors, from 1 to 4.

So, looking carefully at this example, an intuition can be gained that allows us to state that the worst case for messages hit by one or more errors is obtained by first calculating the response time without considering errors, and afterwards add one EC delay per error, since when a error occurs the recovery is delayed by one full EC. By representing the response time in number of ECs, Equation (5.6) allows this calculation, where R_i^{EC} stands for the response time of message i in number of ECs and $R_i^{EC}(no\_error)$ is the response time of message i when a scenario with no errors is considered.

$$R_i^{EC} = R_i^{EC}(no\_error) + max_{errors} \tag{5.6}$$

The algorithm *DirectInterf* obtains the response time with Direct Interference, that considers error in message and afterwards in each recovery try. It checks the schedulability of message set M, with synchronous window of length LSW and max_{errors} in consecutive ECs (one per EC).

ALGORITHM *DirectInterf*
Obtain Message Response Time with Direct Interference

Input: $M, S, LEC, LSW, max_{errors}$
Output: R^{EC} vector, $Schedulable$

```
1.   Compute C_MAX in message set M
2.   Compute M^E (inflate all transmission times using Eq. 5.3)
3.   Schedulable = TRUE
4.   for i = 1 to N do
4.1      Compute R_i considering M^E
4.2      R_i^EC = ⌈R_i / LEC⌉
4.3      R_i^EC = R_i + max_errors
4.4      if ( R_i^EC > D_i ) then
             mark message i as unschedulable
             Schedulable = FALSE
             break
         end_if
     end_for
5.   return vector R^EC, Schedulable
```

Lines 1, 2 and 3 are the same ones as in the preceding algorithm. In Line 4 we have a cycle to compute response time per message, composed of the two following steps: Line 4.1 starts by calculating the response time of the inflated message set, without considering errors (uses Equation (5.4) , without the middle term); in Line 4.2 the obtained values are converted to ECs, permitting to calculate in Line 4.3 the response time considering the errors by adding one additional EC per error. A schedulability test is performed in Line 4.4, similar to previous algorithm. Finally in Line 5 the response time, accounted in number of ECs, and the flag $Schedulable$ that signals the set schedulability are returned.

5.2.3.3 Obtaining Response Time with Both Type of Interference's

The impact on schedulability of a specific message is more penalizing in case of Direct Interference, as the response time is always increased by one EC per suffered error, while Indirect Interference causes only the interference of one message with maximum transmission time equal to C_{MAX} per cycle (per error), not necessarily leading to an additional EC delay. This should be most noticeable when more than one error in consecutive cycles is considered.

This is the reason why with this error model and recovery mechanism the correct response time is obtained considering the scenario for Direct Interference, with maximum number of errors in successive ECs. Finally, remember that these calculations are performed for a Deferrable server with highest priority and enough capacity per period to recover all errors.

5.2.4 Priority Assignment and Scheduling Policies for the Server

The system under consideration uses fixed priorities to schedule the messages. So, the scheduling policy and priority used by the Server have an impact on response time of all messages, both regular and under recovery. For instance, if the lowest priority is assigned to the server, then the recovery is delayed by any message ready for transmission, inducing long recovery times and large jitter, and possibly delaying the recovery beyond message deadline (specially for messages with small periods). This choice would promote the recovery service to background level, as the recovery process runs only when there are no regular messages ready [But11]. On the other hand, if the highest priority is used, then the lowest possible latency in recovery would be achieved, but with greater interference on the remaining traffic. To discuss these aspects with more detail, we will consider the following hypotheses on server policy and priority assignment:

- *MaxPriority* - maximum priority, higher than any message, is assigned to the Server;
- *SamePriority* - the server inherits the message priority;
- *SameDMP* - Same Priority with Deadline Miss Protection, meaning that this scheme is almost equal to the previous one, except that when the message deadline is close, the server priority is raised to the highest value;
- *ServerEDF* - the server priority is deadline aware, like Earliest Deadline First policy.

As each scheme implies a different algorithm, there is an impact on software complexity and run-time overhead of each one of the options.

5.2.4.1 Server with Maximum Priority

The first hypothesis considered was to assign the server with the highest priority, above any regular message. The rationale behind this option was scheduling the server immediately, to allow the retransmissions to occur with the smallest latency possible. This maximizes the probability of retransmission success before the message deadline.

This corresponds to Algorithm *MaxPriority*, that has as inputs the message set, M, the Ready Queue (RQ) of all messages ready for dispatching ordered by priorities and the Server attributtes. These consist on the Server Error Queue ($Server.EQ$), that includes information on messages that have failed transmission and are waiting, and on the server remaining capacity, C_S, that is replenished with period equal to T_S.

ALGORITHM *MaxPriority*
Server with Maximum Priority assigned

Input: *M*, *RQ* (Ready Queue) , *Server*
Output: *SchNextEC* (schedule for Next EC), *RQ*(updated), *Server*(updated)

```
1.    for i = 1 to Server.waiting do
          mSize = message_size(Server.EQ[i])
          if ( Server.Cs ≥ mSize ) then
1.1           Move message Server.EQ[i] to the head of RQ
1.2           Server.Cs = Server.Cs - mSize
          end_if
      end_for
2.    Build the schedule for next EC (SchNextEC) and Update Server
3.    return SchNextEC, RQ, Server
```

Firstly all the messages in the *Server.EQ* are moved to the *RQ*, provided that the Server capacity is not exhausted (Line 1.1), being the server remaining capacity updated accordingly. Under this scheme, retransmissions are added at the head of the *RQ*, giving them higher priority than all other messages, as intended. In Line 2 the scheduling is performed and since these messages are on the head of the *RQ* they are the first ones to be scheduled and dispatched, minimizing their recovery time. In Line 3 the EC schedule is returned, along with the updated *RQ* and *Server*.

5.2.4.2 Server with Same Priority

Despite being intuitive, assigning the highest priority to the server is not the only option as already stated, and in some scenarios it may impose more interference than strictly necessary. For example, consider that a message instance with a far deadline is affected by an error. Assigning the highest priority to the server would cause the retransmission to occur in the following EC, possibly delaying unnecessarily the transmission of messages with shorter deadlines.

In order to gain some insight into this problem, we show some example scenarios, with and without errors, to observe the behavior of the recovery mechanism and the different types of interference produced by different server scheduling policies and priorities. The message set used in the following scenarios is detailed in Table 5.2, where message deadlines are equal to periods and priorities are assigned using the Rate Monotonic policy. The EC size is 5 time units and the transmission time of all messages is one time unit, resulting an utilization of 55%.

Figure 5.6 shows the timelines corresponding to the maximum priority policy, without and with errors (top and middle, respectively). This example illustrates the scenario described above, in which an error in a message with a longer period (message m_6) causes a delay of one EC on a message with a shorter period (message m_5). This kind of interference can

Table 5.2: First message set to illustrate server scheduling policy and priority assignment

n	C_i	T_i	$T_i(EC)$	U
1	1	10	2	10.0%
2	1	10	2	10.0%
3	1	10	2	10.0%
4	1	10	2	10.0%
5	1	10	2	10.0%
6	1	20	4	5.0%
				55.0%

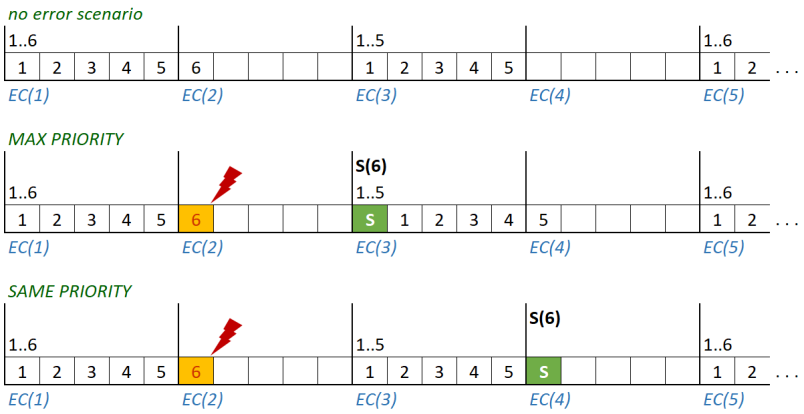

Figure 5.6: *MaxPriority* vs *SamePriority* in server policy, for message set of Table 5.2.

be avoided since message message m_6 could be retransmitted in the following EC, without violating its deadline.

So, an alternative policy, which prevents the priority inversion scenario just described, consists in preserving the messages priority during retransmissions, by assigning the server with the priority of the message to be retransmitted. In terms of implementation, such policy is easily accomplished, since it consists in handling retransmissions in the same way as regular messages.

The corresponding algorithm is presented next, where the changes introduced were inside the cycle of Line 1, to insert the messages present in the error server queue in the correct position of the Ready Queue. Also an additional check has to be performed to verify if the message was included in the schedule for the next cycle. If the response is negative, the messages not dispatched must go back to the error server queue, but only if the deadlines can still be met in the following cycle, else they are discarded. This corresponds to Line 3. With these changes the new algorithm was termed *SamePriority*.

ALGORITHM *SamePriority*
Server Inherits Message Priority

Input: *M*, *RQ* (Ready Queue) , *Server*
Output: *SchNextEC* (schedule for Next EC), *RQ* (updated), *Server* (updated)

```
1.   for i = 1 to Server.waiting do
         mSize = message_size(Server.EQ[i])
         if ( Server.Cs ≥ mSize ) then
             Move message Server.EQ[i] from Server.EQ to the the proper
             position in the RQ, according to its priority
             server.Cs = Server.Cs - mSize
         end_if
     end_for
2.   Build the EC Schedule (TM) and update Server
3.   Put all retransmissions not dispatched (with feasible deadlines)
     back on Server.EQ & adjust server capacity accordingly
4.   return SchNextEC, RQ, Server
```

Since, in this case, the messages in the *Server.EQ* are scheduled together with the remaining messages, following a system-wide fixed-priority scheduling policy, message retransmissions do not interfere with higher priority messages, as desired. This can be observed in Figure 5.6, bottom timeline.

With this priority assignment to the server, the error recovery latency is not minimized anymore, being dependent on the priority of the message affected by the error and on the instantaneous system load. Consequently, this scheduling policy has a behavior that is dual of the *MaxPriority* scheme.

On the other hand, there could be some undesirable side-effects, as inheriting the message

Table 5.3: Second Message set to illustrate server scheduling policy

n	C_i	T_i	$T_i(EC)$	U
1	1	10	2	10.0%
2	1	10	2	10.0%
3	1	10	2	10.0%
4	1	10	2	10.0%
5	1	10	2	10.0%
6	1	15	3	6.7%
				56.7%

priority could lead to unnecessary delays and possibly to deadline misses. The message set in Table 5.3, that is based on the one in Table 5.2, except in what concerns the period of message m_6, which is reduced from 4 ECs to 3 ECs, increasing only slightly the utilization factor, can be used to illustrate this point. In the scenario illustrated in Figure 5.7, bottom timeline, the message retransmission was delayed, leading to a deadline miss. Conversely, if the *MaxPriority* scheme was used, no deadline would be missed, as pointed out before (Figure 5.7, second timeline). Note that in this case the deadline miss is solely due to the chosen server policy and not to additional errors, being this a negative point attributed to server configuration only.

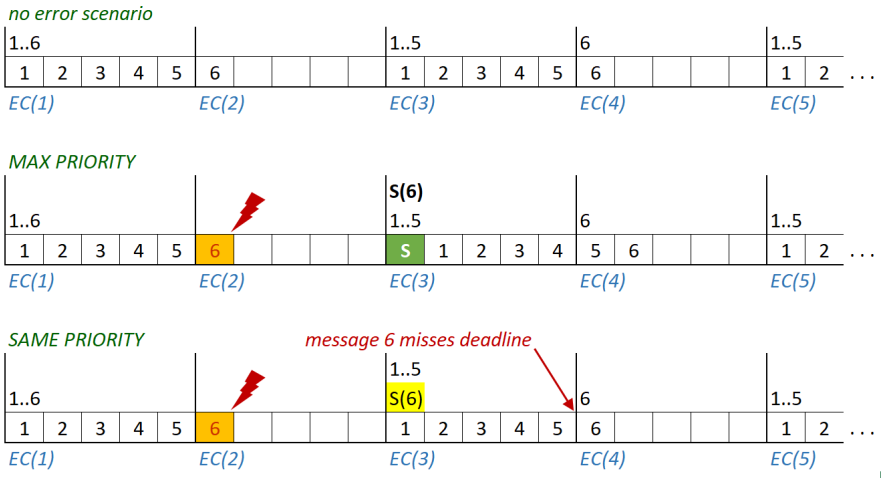

Figure 5.7: *MaxPriority* vs *SamePriority* server policy, for message set of Table 5.3.

5.2.4.3 Server with Same Priority and Deadline Miss Protection

Reasoning about the scenarios shown above, it can be concluded that both scheduling schemes have a common problem: the scheduling decisions regarding message retransmissions

do not take into account the dynamic system state and thus are suboptimal. This observation suggests that a possible way of improving the system performance may be achieved by using the message slack as a scheduling decision parameter. One simple way of obtaining this behavior consists in using the *SamePriority* scheme by default, in order to prevent priority inversions as much as possible, and assigning the maximum priority to the server whenever the message at the head of its error queue has a slack of one EC, in order to reduce the number of deadline misses affecting retransmissions of low-priority messages. This scheme, designated *SamePriorityDMP*, is described in Algorithm *SamePriorityDMP*. Lines 2, 3 and 4 are the same ones as in Algorithm *SamePriority*.

ALGORITHM *SamePriorityDMP*
Server Scheduling with Same Priority and Deadline Miss Protection

Input: M, Ready Queue (RQ), Server
Output: **Output**: *SchNextEC* (schedule for Next EC), *RQ*(updated), *Server*(updated)

```
1.    for i = 1 to Server.waiting do
          mSize = message_size(Server.EQ[i])
          if ( Server.Cs ≥ mSize ) then
1.1           if ( absolute_deadline(Server.EQ[i]) > Next EC ) then
                  Insert message in the RQ, with original priority
              else
                  Insert message at the head of the RQ
              end_if
              server.Cs = Server.Cs - mSize
          end_if
      end_for
2.    Build the EC Schedule (TM) and update Server.EQ
3.    Put all retransmissions not dispatched back on Server.EQ
      & adjust server capacity accordingly (Server.Cs)
4.    return SchNextEC, RQ, Server
```

This algorithm adds then and additional test, in Line 1.1, that forces the insertion of the message present in the *Server.EQ* in to the head of the *RQ*, promoting its priority to the maximum value. This is only done when the next EC is the last one that permits its retransmission without violating the message deadline, else it inherits the message priority. This modification guarantees that at least one recovery try is performed, overcoming this way the identified undesirable side-effect described previously. Though simple and easy to implement, as it requires only one additional test with respect to the *SamePriority* scheme, the *SamePriorityDMP* policy is still suboptimal as it neglects the messages slack until a border condition is met, namely the last EC before the deadline.

5.2.4.4 Server with EDF policy

The Earliest-Deadline First policy firstly presented in [LL73] is a scheduling policy that takes into account the dynamic state of the system, giving priority to the tasks with more urgent deadlines. This is the feature that makes EDF optimal with respect to meeting deadlines, while allowing higher utilization factors than fixed-priority schemes.

Errors happen in a random way, may affect both messages with short or long deadlines, so it is not possible to forecast which kind of requests will be submitted to the server, thus optimum decisions cannot be fixed nor taken off-line. Taking also into account the previous comments about the *SamePriority* and *SamePriorityDMP* policies, this feature seems particularly well suited for the error server scheduling, as it allows promoting dynamically the recovery priority.

If this policy is applied to the server, then retransmissions will have a minimum interference on other messages when their deadlines are farther away, but gain importance gradually, i.e., increase their priority, as their deadlines approach, as desired.

Also, related work that proposes hierarchical scheduling using different scheduling policies at different levels in the scheduling process [HP03], seems to point out that this could be an interesting strategy for the server scheduling policy (for the messages in the server only).

The operation of the EDF policy for the Server is described in Algorithm *ServerEDF*. The inputs and outputs are the same ones as in previous algorithms, being significantly different in the way that messages waiting in the *Server.EQ* are included in the schedule.

ALGORITHM *ServerEDF*
Server Scheduling with EDF Priority

Input: *M*, *RQ* (Ready Queue), *Server*
Output: *SchNextEC* (schedule for Next EC), *RQ*(updated), *Server*(updated)

```
1.  Sort RQ by deadlines
2.  Sort Server.EQ by deadlines
3.  for i = 1 to Server.waiting do
        mSize = message_size(Server.EQ[i])
        if ( Server.Cs ≥ mSize ) then
            Insert message in the RQ sorted by deadline
            server.Cs = Server.Cs - mSize
        end_if
    end_for
4.  Build the SchNextEC (Schedule for the next cycle) and update Server.EQ
5.  Put all retransmissions not dispatched back on Server
    & adjust server capacity accordingly (Server.Cs)
6.  return SchNextEC, RQ, Server
```

The algorithm starts by sorting the *RQ* and *Server.EQ* by deadlines (lines 1 and 2). In Line 3, each message waiting in the *Server.EQ* is deadline compared with the regular messages

being included in the schedule if their deadline is shorter than the one of any other regular messages. Line 4 builds the schedule for the next EC, using the *RQ* built in the previous step (with deadline ordering). As the schedule is limited by the maximum available time in the synchronous window, in case the messages in the server queue do not fit, then they go back to the server queue, being this done in Line 5. Line 6 returns the schedule for the next EC and the updated versions of *RQ* and *Server*.

The EDF scheduling policy has the potential to deliver the best results in terms of interference, outperforming the other policies in many cases. However, it incurs in a relatively high overhead, since it requires that both the server error queue and the Ready Queue are sorted by deadline and insertions on the Ready Queue must also be made according to the deadline. Considering that CAN is often used in embedded systems based on resource-constrained hardware (e.g., in 8 bit microcontrollers), such overhead can be problematic. Therefore a fine-grained cost-benefit analysis must be performed, considering namely the use of optimized solutions for the algorithm implementation.

Figure 5.8 presents the different behavior in terms of recovery latency and interference of the four proposed server scheduling policies considering the message set defined in Table 5.4. The timelines show a scenario in which only the *EDFServer* scheme can recover a message without causing any deadline miss.

Table 5.4: Example message set for comparison of the four policies.

i	C_i	T_i	$T_i(EC)$	U
1	1	10	2	10.0%
2	1	10	2	10.0%
3	1	10	2	10.0%
4	1	10	2	10.0%
5	1	10	2	10.0%
6	1	15	3	6.7%
7	1	15	3	6.7%
8	1	25	5	4.0%
9	1	25	5	4.0%
10	1	25	5	4.0%
11	1	25	5	4.0%
12	1	25	5	4.0%
13	1	25	5	4.0%
				87.3%

Figure 5.8: The four rescheduling politics compared for a particular system.

Some comments follows:

- Second timeline (*MaxPriority*): it has the fastest recovery latency, delaying messages with higher priority; the prompt recovery delays message m_{13}, making it loose its deadline;
- Third timeline (*SamePriorityDMP*): the interference with higher priority is avoided, delaying the recovery one more EC, when compared to previous policy; message m_{13} still misses deadline. Both *SamePriority* and *SamePriorityDMP* show an equivalent behaviour;
- Bottom timeline (*ServerEDF*): in EC(3) all the ready messages have lower deadlines than message in *Server.EQ*, so the server execution is delayed; in EC(4) the message in *Server.EQ* has the lowest deadline, so it is scheduled in this cycle; as the scheduling decisions are made by deadline, it is message m_7 that is postponed (it has the farthest deadline), giving space to message m_{13} to be scheduled EC(4) and not loose its deadline, contrary to the previous two timelines.

5.2.5 Limits on the recovery success

The use of single replica retransmission puts a limit on the guarantees given in message recovery, in case of errors in consecutive ECs. The scenario, depicted in Figure 5.9, that uses the message set of Table 5.5, with LEC equal to 10 time units, shows that a message can fail its deadline if this situation happens. For the messages present in a particular set, the ones that have smaller deadlines are more prone to this situation, since with the proposed recovery method a success in transmission can only happen if there is slack time to the message deadline, as the recovery message is delayed by at least one EC.

Table 5.5: Message set used in following examples

i	C_i	T_i	$T_i(EC)$
1	1	20	2
2	1	20	2

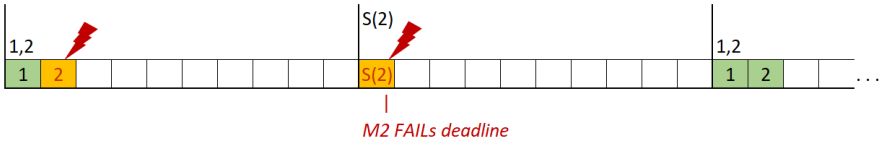

Figure 5.9: Two errors in consecutive ECs leads to message failing deadline (LEC = 10 time units).

The obvious solution to overcome this limitation is for the slave to send two copies in the same EC, possibly using both windows. Due to the single error assumption, a successfull retransmission would be guaranteed. In case of no error, the slave will get two copies of the same message and must discard one of them. This approach, when using the asysnchronous window to transmit the message copy, would maintain all the scheduling control in the Master node, but includes a penalty in traffic scheduling for the asynchronous traffic, equal to the interference of the longest synchronous message. The slave code would have to be changed to identify correctly this situation.

Another possibility to overcome this limiation is to adjust further the EC length, using smaller than half the value of the fastest messages, so these messages have several opportunities of recovery before reaching their deadlines. The example depicted in Figure 5.10, that uses the message set in Table 5.5 with LEC reduced to 5 time units (LEC is half the value of the previous example), shows that even with higher fault rate assumption it still guarantees success in message delivery.

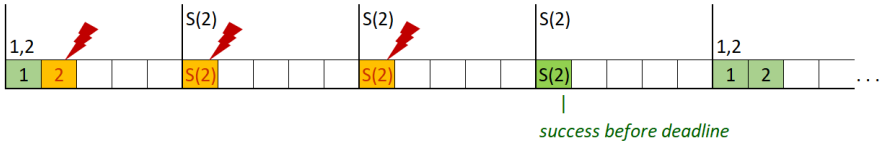

Figure 5.10: Three errors in consecutive ECs and message still meets its deadline (LEC = 5 time units).

This method implies a schedulability penalty because it increases the number of required TM messages and inserted idle time.

A third way consists in sending an adequate number of retransmission replicas, each time an error is detected. This will be further detailed in the next section.

As a concluding remark, we can state that this simple method can effectively increase the network transmission reliability, but only to a certain degree, as the recovery is compromised

by errors in consecutive cycles, namely for the messages with short deadlines. Knowing its limitations, it can be used if the application can deal with the attainable reliability level, that can be complemented using robust control algorithms in the application, that should be tolerant to a certain degree of faults (k out of n transmissions errors). Nevertheless, it allowed a first approach to a new dynamic recovery method (in the time domain), in the FTT-CAN scope, and also permitted to clearly identify its limitations.

5.3 Error Recovery in the Time Domain - Multiple Replica Version

5.3.1 Limitations and Motivating for an Improved Recovery Method

The limitations of the basic recovery method were already identified in Section 5.2.5, considering the simplified fault model with a limit of one error per EC. The solutions proposed to overcome retransmission failures still do not give guarantees when more realistic scenarios are considered, namely by considering multiple errors per EC.

So, when designing systems that must provide high reliability, the error scenarios should include more than one error per EC (per SW). To present enhanced transmission guarantees, the proposed solution consists in sending several replicas to attain the desired probability of success in each recovery try and consequently attain the required reliability target. This new strategy is represented in Figure 5.11.

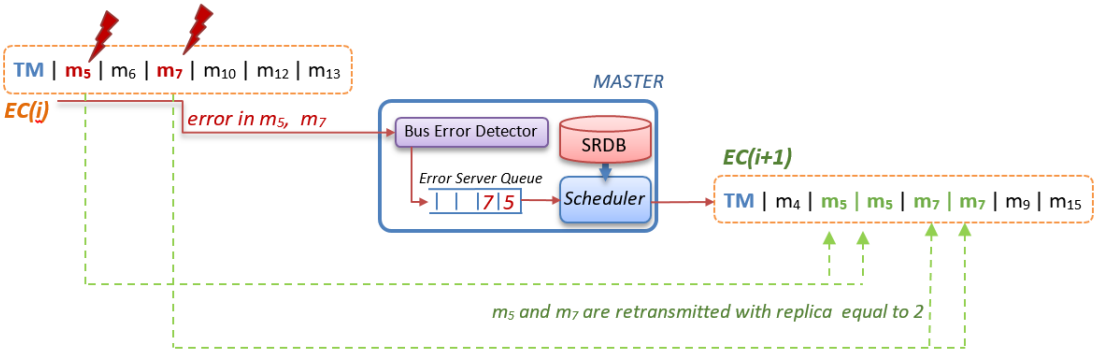

Figure 5.11: Recovery with multiple replicas.

The example in this figure shows that messages m_6, m_{10}, m_{12} and m_{13} are correctly received in EC(i). On the other hand, messages m_5 and m_7 suffer transmission errors. The master flags each message received successfully at the end of the SW by comparison with the planned messages for this cycle. Then the Bus Error Detector module identifies the failed messages, that consequently are inserted in the Error Server Queue. In EC(i+1) the retransmitted messages are sent twice, being therefore resilient to additional errors in this EC.

5.3.2 Update on Server Capacity Computation

The Server type, period and capacity for the single replica per error choosing was presented in Section 5.2.2, being still valid for this recovery version. The main difference now is that we need to take into account the need for multiple retransmissions to achieve a desired reliability level, as discussed in the previous section. Noting that it is not possible to foresee which messages will be affected by errors, the worst-case situation happens when all of them must be transmitted in the following EC, implying scheduling several replicas at once.

The number of errors foreseen per server period is obtained using Equation (5.2), with limit probability $p_{\epsilon s}$ and depicted in Figure 5.3, as we are still using the Poisson model for fault arrival. Then, accounting that $n_{replicas}$ copies must be sent for each found error, the new server capacity is then given by Equation (5.7), in number of maximum length messages.

$$C_S = n_{errors} \cdot n_{replicas} \tag{5.7}$$

For example, considering one server period ($T_S = 1/\lambda$) and a target of $p_{\epsilon s} = 10^{-10}$, the minimum value of n_{errors} that satisfies Equation (5.1) is 13 (see Figure 5.3). Assuming, for instance, 3 replicas per error (i.e., 3 replicas allow attaining the desired global reliability), the server must then have a capacity equal to $13 \cdot 3 = 39$ maximum length messages. If a LEC equal to 2.5 ms is used, then the allocated server capacity is 0.14% of the system bandwidth (with bit rate of 1 Mbps, $C_{MAX} = 135\mu s$ and $T_S = 1/\lambda = 3.85$ seconds), which, again, represents a negligible fraction of the available bandwidth.

5.3.3 Obtaining Worst Case Response Time of Messages with Server Interference

This section presents a method to compute the worst-case response time (WCRT) of messages, taking into consideration the errors and the interference of the server.

It starts by giving a generic equation to obtain the response time, that must be evaluated for each considered error scenario and interference pattern (corresponding to server execution), followed by the identification of the error scenarios and interference patterns, including the algorithms description.

5.3.3.1 Updating the Message Response Time Computation

In the FTT-CAN scope, the worst case response time of synchronous messages can be obtained using the classical Response Time Analysis [ABR+93], by applying the Blocking-Free Non-Preemptive scheduling model [AF01], where all the transmission times are inflated (see Equation 3.8), as described in Section 3.3.2.1.1. The Equation (3.12) applies to the error free transmission, that now must be adapted, accounting for all error and recovery scenarios. To apply the new equation is assumed, as before, that the message set is ordered by decreasing priority order.

Then, we must consider all the error scenarios and the interference patterns, by including these terms in Equation (3.12). This updated formulation results in Equation (5.8), where the server interference is represented by the intermediate term - the summation of $Interf\_Pattern$ - and the error signaling by the summation of $Err\_Scenario$. Note that this formulation also implies a recovery try with minimum delay possible, so each error in a particular EC is recovered in the following EC. So, the first term in the summation depends on the errors that occurred in the previous SW and on the necessary message replication number and the second one depends on the errors in the current SW. The response time of each message must be calculated for each error scenario and server interference, accounted by the l variable, being the response time of message i the greatest of all the calculated values.

$$R_i^{n+1}(l) = C_i^E + \sum_{j=1}^{EC_{number}(R_i^n(l))} \left(Interf\_Pattern(l,j) \cdot C_{MAX}^E + Err\_Scenario(l,j) \cdot C_{error}^E \right) + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^n(l)}{T_k} \right\rceil \cdot C_k^E \tag{5.8}$$

The limit on the summation $EC_{number}(R_i^n(l))$ is obtained by Equation (5.9), and represents the number of ECs that need to be analyzed in iteration n for message i.

$$EC_{number}(R_i^n(l)) = \left\lceil \frac{R_i^n(l)}{LEC} \right\rceil \tag{5.9}$$

In the following sections the number of replicas, the error scenarios and interference patterns are obtained in order to calculate the response time of all messages and determine the system schedulability.

5.3.3.2 Obtaining the Number of Replicas

Let's start by giving a simple example to motivate the generic procedure. Figure 5.12, represents a case that shows that multiple replica retransmission enhance the probability of error recovery success. In the top timeline a single copy retransmission is performed, that suffers an additional error in EC(2), leading to a message deadline miss. On the other way, if double replica was sent, bottom timeline, a successful message retransmission is obtained. As fault arrivals are modeled by a Poisson process, it is not possible to upper bound the number of faults in any given time interval, so a second error in EC(2) can not be ruled out and in this scenario the retransmission would fail again. So, the question is: how many replicas should be sent to give guarantees that the retransmission is sucessfull?

Of course, due to the random process of fault arrival, only a probabilistic guarantee can be given on the recovery success. As the fault arrival is modeled by a Poisson process, for a fixed time interval, increasing the number of considered faults decreases the corresponding probability. Therefore, it is possible to compute the number of replicas that guarantees

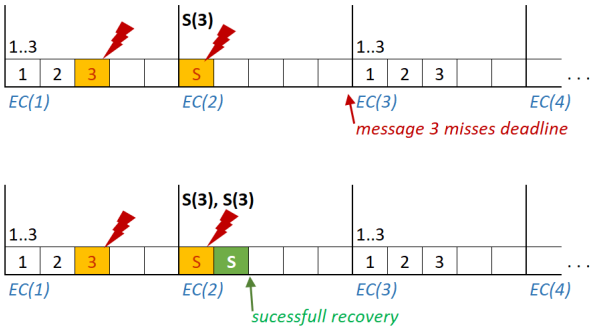

Figure 5.12: Message and replica hit by errors.

successful and timely transmissions to attain a global reliability goal $\rho > 1 - \epsilon_G$, where ϵ_G is the maximum probability of failure or system/global unreliability level.

Usually, the system reliability level is specified in acceptable errors per mission and a generally accepted mission time is one hour [TBEP10], thus leading to a common metric of acceptable errors per hour. The system unreliability objective ϵ_G can be converted to the error probability that each individual message may tolerate, when considering the messages' periods and mission time (MT seconds), being this probability named $p\epsilon_i$.

The transmission reliability can be calculated by Equation (5.10), that includes the contributes of all N messages and the number of invocations in the mission time (previously presented as Equation (4.6), that was repeated here for convenience). As the probabilities of insuccess, $p_{\epsilon i}$, are extremely small (even for Agressive environments) a good approximation of this equation can be obtained using the first two terms of the Taylor series expansion, as in Equation (5.11). The new formulation can be solved by upper bounding the i terms, using the smallest value of T_i and the biggest $p\epsilon_i$ (worst GP value will be obtained) and then making all N elements equal to this bound and applying again Taylor series approximation, resulting in Equation (5.12).

$$GP = \prod_{i=1}^{N} \left(1 - p_{\epsilon i}\right)^{\frac{MT}{T_i \cdot LEC}} \tag{5.10}$$

$$GP \approx \prod_{i=1}^{N} \left(1 - \frac{MT}{T_i \cdot LEC} \cdot p_{\epsilon i}\right) \tag{5.11}$$

$$GP \approx 1 - N \cdot \frac{MT}{T_i \cdot LEC} \cdot p_{\epsilon i} \tag{5.12}$$

$$1 - N \cdot \frac{MT}{T_i \cdot LEC} \cdot p_{\epsilon i} = 1 - \epsilon_G \tag{5.13}$$

$$p_{\epsilon i} = \frac{\frac{\epsilon_G}{\left(\frac{MT}{T_i \cdot LEC}\right)}}{N} \tag{5.14}$$

Then, by solving Equation (5.13), finally Equation (5.14) is obtained. This equation defines an acceptable failure probability for the message i, where T_i is the message period (in number of ECs), MT is the mission time and N is the number of messages subject to the recovery mechanism. This equation also shows that the most demanding messages, i.e., with lower $p_{\epsilon i}$, are those with the smallest period. Since the error-handling mechanism will be used for all messages, design decisions will be made considering the smallest $p_{\epsilon i}$ value, that this point forward will be denoted as p_ϵ.

5.3.3.2.1 Replica Level To obtain the replica level we will use a set of error and recovery scenarios, depicted in Figures 5.13, 5.14, 5.17 and 5.18, to obtain a general expression for computing the number of replicas needed to attain a given system global reliability level. In practice, the values of λ, LSW and p_ϵ limit the number of scenarios that must be considered.

Let us start with the one error/one replica scenario, shown on the top timeline of Figure 5.13. This event sequence occurs if one error affects a synchronous message, event with probability $P_\lambda(1; LSW)$, and the corresponding replica is also affected by an error, event with probability $P_\lambda(1; C_i)$. As errors are independent, the resulting probability ($p_{1/1}$) is given by the product of both probabilities. Therefore, the probability of this scenario is given by Equation (5.15), where message transmission times are upper bounded by C_{MAX}, to enable the derivation of generic equations.

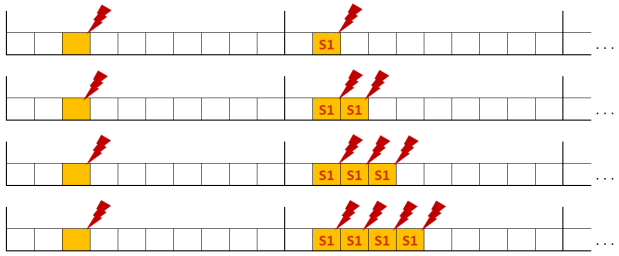

Figure 5.13: One error and recovery with 1 to 4 replicas in following cycle.

$$p_{1/1} = P_\lambda\big(1; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big) \tag{5.15}$$

$$p_{1/2} = P_\lambda\big(1; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^2 \tag{5.16}$$

$$p_{1/3} = P_\lambda\big(1; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^3 \tag{5.17}$$

$$p_{1/4} = P_\lambda\big(1; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^4 \tag{5.18}$$

$$p_{1/n_{replicas}} = P_\lambda\big(1; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^{n_{replicas}} \tag{5.19}$$

If the probability obtained via Equation (5.15) is lower than p_ϵ then a single replica

is enough to guarantee the desired message transmission reliability level. Otherwise, an additional replica must be sent. This scenario is shown in the second timeline of Figure 5.13. This case is a simple extension of the previous one, in which we consider the combined probability of both replicas being hit. So, the probability of occurrence of this scenario ($p_{1/2}$) is given by Equation (5.16). Following this line of reasoning we obtain ($p_{1/3}$) and ($p_{1/4}$), that also leads to the probability for the scenario one error/$n_{replicas}$ ($p_{1/n_{replicas}}$) given by Equation (5.19). The smallest number of replicas that makes Equation (5.19) lower than p_ϵ is sufficient to attain the desired global reliability level for this scenario.

Since the error model allows the occurrence of multiple errors in one EC, we will now consider the scenarios in which two synchronous messages scheduled for the same EC are affected by errors, as in Figure 5.14, that depicts all recovery failure scenarios with minimum number of errors. When considering first that a single replica per detected error is sent, if an error hits one of the replicas the recovery process fails. The probability of failure of the recovery process ($p_{2/1}$) is then simply obtained by adding the probabilities of both these combinations, each one with probability $P_\lambda(2; LSW)$, probability of having two errors in the SW, times $P_\lambda(1; C_{MAX})$ probability of one replica error, as expressed in Equation (5.20).

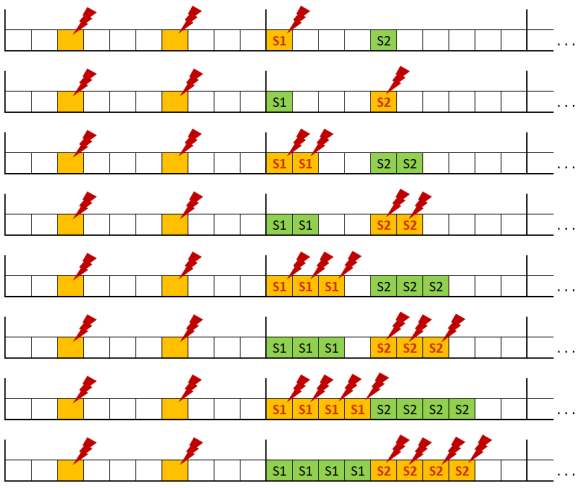

Figure 5.14: Two error and recovery with 1..4 replicas in following cycle - all fail scenarios with minimum number of errors.

$$p_{2/1} = 2 \cdot P_\lambda\big(2; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big) \tag{5.20}$$

Note that the scenario where both replicas fail transmission is also possible, as represented in the bottom timeline of Figure 5.15, was not accounted in Equation (5.20). So, including this scenario (with one more error than the minimum number of errors that makes the recovery process fail), the probability of scenario failure is calculated through Equation (5.21), as it includes the scenario with two errors, one hitting each recovery try.

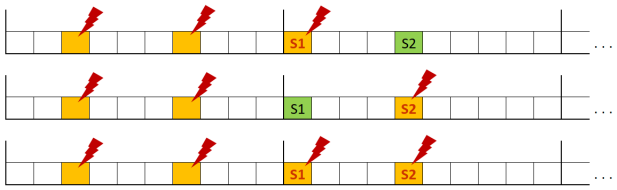

Figure 5.15: All scenarios for 2 errors and single replica that fails recovery (1 or 2 errors in recovery EC).

$$\begin{aligned} p_{2/1(fail)} &= P_\lambda(2; LSW) \cdot \left(P_\lambda(1; C_{MAX}) + P_\lambda(1; C_{MAX}) + P_\lambda(1; C_{MAX})^2\right) \\ &= P_\lambda(2; LSW) \cdot \left(P_\lambda(1; C_{MAX}) \cdot (1 + 1 + P_\lambda(1; C_{MAX}))\right) \\ &\approx 2 \cdot P_\lambda(2; LSW) \cdot P_\lambda(1; C_{MAX}) \end{aligned} \tag{5.21}$$

The approximation is valid since the $P_\lambda(1; C_{MAX})$ is much smaller than unity (10^{-5} order for maximum length CAN message in Agressive environment). So, the probability was truncated with the two first scenarios, the ones with minimum number of errors that makes the recovery transmission fail, resulting in Equation (5.20).

When considering now that two replicas per message are sent, a failure of the recovery process occurs only if both replicas of the same message are affected by errors, event with probability $p_{2/2}$, expressed in Equation (5.23), corresponding to the third and fourth timeline in Figure 5.14.

Again, Equation (5.23), accounts only for error recovery failure with minimum number of errors. Figure 5.16 represents all scenarios (with two, three and four errors in the recovery EC) that allow us to write Equation (5.22). Due to the presence of values much lower than unity resulting from the contribution of the third or fourth error, it is possible to simplify this equation by accounting only the scenarios with number of failures equal to number of replicas, since considering more errors has negligible increase in all error scenarios probability. This approximation of Equation (5.22) results then in Equation (5.23).

$$\begin{aligned} &p_{2/2(fail)} = \\ &P_\lambda(2; LSW) \cdot \left(2 \cdot P_\lambda(1; C_{MAX})^2 + 4 \cdot P_\lambda(1; C_{MAX})^2 \cdot P_\lambda(1; C_{MAX}) + P_\lambda(1; C_{MAX})^2 \cdot P_\lambda(1; C_{MAX})^2\right) \\ &= P_\lambda(2; LSW) \cdot \left(2 \cdot P_\lambda(1; C_{MAX})^2 \cdot \left(1 + P_\lambda(1; C_{MAX}) + 1/2 \cdot P_\lambda(1; C_{MAX})^2\right)\right) \\ &\approx P_\lambda(2; LSW) \cdot \left(2 \cdot P_\lambda(1; C_{MAX})^2\right) \end{aligned} \tag{5.22}$$

This line of reasoning is also applied in the next scenarios, so from this point on we will only consider the scenarios with minimum number of errors that make the process of error recovery fail, since the contribution of scenarios with greater number of errors is negligible.

Iterating the reasoning it is possible to obtain Equations (5.24) and (5.25) for the scenarios

Figure 5.16: All scenarios for 2 errors and 2 replicas per message that fails recovery, considering scenarios from 2 to 4 errors in the recovery cycle.

"2 errrors/3 replicas" and "2 errors/4 replicas", respectively. Finally Equation(5.26), allows for computing the probability ($p_{2/n_{replicas}}$) of non-recovery for the two errors/$n_{replicas}$ scenario.

$$p_{2/2} = 2 \cdot P_\lambda\big(2; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^2 \tag{5.23}$$

$$p_{2/3} = 2 \cdot P_\lambda\big(2; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^3 \tag{5.24}$$

$$p_{2/4} = 2 \cdot P_\lambda\big(2; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^4 \tag{5.25}$$

The generic expression for the scenarios with two errors is then:

$$p_{2/n_{replicas}} = 2 \cdot P_\lambda\big(2; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^{n_{replicas}} \tag{5.26}$$

The scenarios with 3 errors and recovery try, for increasing number of replicas (from one to three), with minimum number of errors in the recovery cycle, are depicted in Figure 5.17, being the corresponding failure probabilities given by the two following equations.

$$p_{3/1} = 3 \cdot P_\lambda\big(3; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big) \tag{5.27}$$

$$p_{3/2} = 3 \cdot P_\lambda\big(3; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^2 \tag{5.28}$$

$$p_{3/3} = 3 \cdot P_\lambda\big(3; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^3 \tag{5.29}$$

A generic equation for the scenario "3 errors/$n_{replicas}$" is then:

$$p_{3/n_{replicas}} = 3 \cdot P_\lambda\big(3; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^{n_{replicas}} \tag{5.30}$$

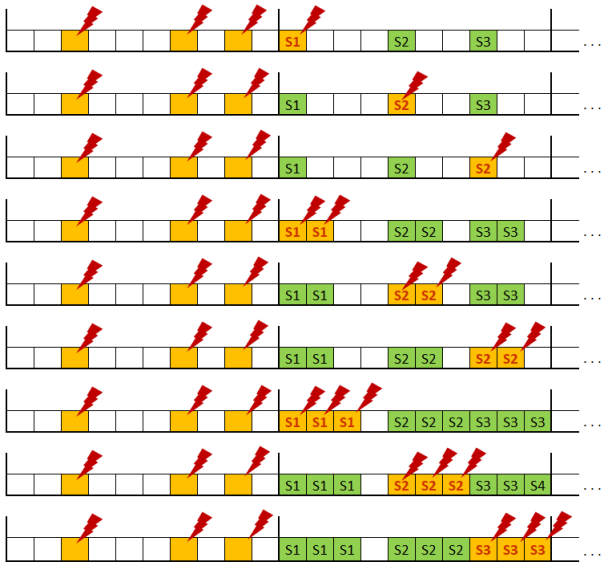

Figure 5.17: Three errors and recovery with 1..3 replicas in following cycle.

The scenarios with 4 errors followed by recovery try, with one and two replicas are presented in Figure 5.18.

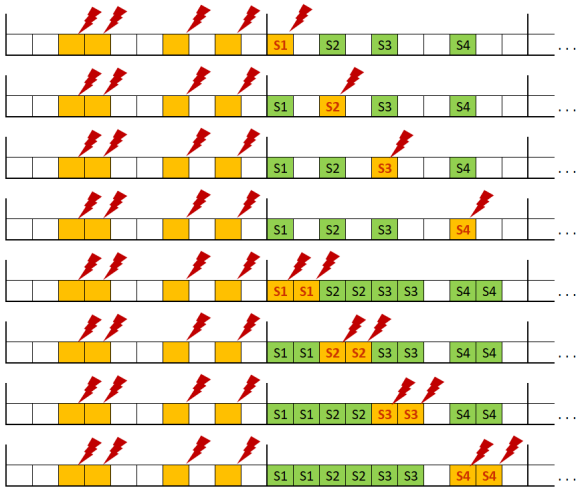

Figure 5.18: Four errors and recovery with 1..2 replicas in following cycle.

The corresponding scenario failure probabilities for the scenarios "4 errors/1 replica", "4 errors/2 replicas" and "4 errors/$n_{replicas}$" are given by the following equations:

$$p_{4/1} = 4 \cdot P_\lambda\big(4; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big) \tag{5.31}$$

$$p_{4/2} = 4 \cdot P_\lambda\big(4; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^2 \tag{5.32}$$

$$p_{4/n_{replicas}} = 4 \cdot P_\lambda\big(4; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^{n_{replicas}} \tag{5.33}$$

The same methodology can be applied to obtain the generic expression for scenarios with arbitrary number of errors n_{errors} and replicas $n_{replicas}$, with general format given by Equation (5.34).

$$p_{n_{errors}/n_{replicas}} = n_{errors} \cdot P_\lambda\big(n_{errors}; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^{n_{replicas}} \tag{5.34}$$

The Algorithm **Calc_RepLevel** calculates the necessary replica number for arbitrary scenarios. This algorithm is based on Equation (5.34), returning vector *RepLevel*, which contains the number of replicas necessary to obtain a probability of non-recovery below p_ϵ for i errors in the previous SW.

ALGORITHM Calc_RepLevel
Replica Level Calculation

Input: $M, LSW, \lambda, p\epsilon$
Output: *RepLevel* vector

1. Determine C_{MAX} in message set M
2. Obtain maximum value of Max_Errors in $P_\lambda(Max\_Errors; LSW) > p\epsilon$
3. **for** $n_{errors} = 1$ **to** Max_Errors **do**
4. j = 0
5. **do**
   j = j + 1
   $P = n_{errors} \cdot P_\lambda\big(n_{errors}; LSW\big) \cdot P_\lambda\big(1; C_{MAX}\big)^j$
   while ($P > p\epsilon$)
   $RepLevel(n_{errors}) = j$
   end_for
6. **return** vector *RepLevel*

For illustration purposes, Algorithm *Calc_RepLevel* was applied to an FTT-CAN system with a 1 Mbps bit rate, LEC = 2.5 ms, LSW = 1.25 ms, $\lambda = 0.26$ errors per second (Aggressive environment, Table 2.1) and 15 messages with period 5 ms and size 125 bits (including maximum bit-stuffing). The desired global unreliability level ϵ_G was set to 10^{-9}, which translates to a $p_\epsilon \approx 10^{-16}$ by applying Equation (5.14). Table 5.6 presents the obtained values. The vector returned by Algorithm ***Calc_RepLevel*** for the example above is *RepLevel*={3,3,2,1} .

For comparative purposes, we also considered an alternative Normal environment with a lower BER, leading to the results shown in Table 5.7. As expected, the number of errors that may affect synchronous messages is, probabilistically, much smaller, thus requiring a significantly smaller number of replicas to attain the same global reliability level.

FTT-CAN has several configuration parameters that can be tuned to suit the application requirements. Of particular relevance are the EC duration (LEC) and the maximum length of the synchronous window (LSW). To get insight about the impact of these parameters on

Table 5.6: Number of replicas needed for a target reliability level in an Aggressive environment

Scenario	***replica number (n msgs)***	p_{fail}	$\epsilon_G = 10^{-9}$	***overhead*** *(number msgs)*
1 error, single ret	1	$1.06 \cdot 10^{-08}$	X	1
1 error, double ret	2	$3.43 \cdot 10^{-13}$	X	2
1 error, triple ret	3	$1.12 \cdot 10^{-17}$	OK	3
2 errors, single ret	1	$3.43 \cdot 10^{-12}$	X	2
2 errors, double ret	2	$1.12 \cdot 10^{-16}$	X	4
2 errors, triple ret	3	$3.62 \cdot 10^{-21}$	OK	6
3 errors, single ret	1	$5.58 \cdot 10^{-16}$	X	3
3 errors, double ret	2	$1.81 \cdot 10^{-20}$	OK	6
4 errors, single ret	1	$6.04 \cdot 10^{-20}$	OK	4

Table 5.7: Same as Table 5.6, but for a Normal environment

Scenario	***replica number (n msgs)***	p_{fail}	$\epsilon_G = 10^{-9}$	***overhead*** *(number msgs)*
1 error, single ret	1	$1.50 \cdot 10^{-12}$	X	1
1 error, double ret	2	$5.82 \cdot 10^{-19}$	OK	2
2 errors, simple ret	1	$5.82 \cdot 10^{-18}$	OK	2

the system reliability, Algorithm *Calc_RepLevel* was applied to an FTT-CAN system like the one assumed for the results in Table 5.6 but with LEC, LSW and message periods 10 times bigger. The results are reported in Table 5.8.

Table 5.8: Same as Table 5.6, but with LEC = 25 ms and LSW = 12.5 ms

Scenario	***replica number*** *(number msgs)*	p_{fail}	$\epsilon_G = 10^{-9}$	***overhead*** *(number msgs)*
1 error, single ret	1	$1.05 \cdot 10^{-07}$	X	1
1 error, double ret	2	$3.42 \cdot 10^{-12}$	X	2
1 error, triple ret	3	$1.11 \cdot 10^{-16}$	OK	3
2 error, single ret	1	$3.42 \cdot 10^{-10}$	X	2
2 error, double ret	2	$1.11 \cdot 10^{-14}$	X	4
2 errors, triple ret	3	$3.61 \cdot 10^{-19}$	OK	6
3 error, single ret	1	$5.56 \cdot 10^{-13}$	X	3
3 errors, double ret	2	$1.81 \cdot 10^{-17}$	OK	6
4 errors, single ret	1	$6.02 \cdot 10^{-16}$	OK	4
5 errors, single ret	1	$4.89 \cdot 10^{-19}$	OK	5

Since the smallest message period is now 50 ms, p_ϵ becomes equal to 10^{-15} and the obtained ***RepLevel*** vector is {3, 3, 2, 1, 1}. We can also see that for longer LSW we must consider the possibility of more errors per SW, thus generating higher overhead and a degradation of the probability of failure for all considered scenarios. Therefore, we expect longer ECs and SWs to increase error recovery overhead.

5.3.3.3 Building the Interference Patterns

To build the interference patterns we need first to determine the necessary replication level. Afterwards, we have to compute how many errors must be handled in a single SW and also the maximum number of single errors that have to be accommodated in consecutive SWs, which are the two extreme cases (to see this just calculate the scenario probability by applying Equation (2.8) to all the considered error scenarios). Algorithm *Max_Errors* computes these values, termed *max_1cycle* and *max_cycles*, that will be used to build all possible error scenarios.

ALGORITHM MaxErrors

Maximum single consecutive errors and maximum errors in one cycle

Input: LSW, $\lambda, p\epsilon$

Output: max_1cycle, max_cycles

1. $max\_cycles = 0, p\_error = 1.0$
2. **while** ($p\_error > p\epsilon$) **do**
   $max\_cycles = max\_cycles + 1$
   $p\_error = P_\lambda(1; LSW)^{max\_cycles}$
   end_while
3. $max\_cycles = max\_cycles - 1$
4. $max\_1cycle = 0, p\_error = 1.0$
5. **while** ($p\_error > p\epsilon$) **do**
   $max\_1cycle = max\_1cycle + 1$
   $p\_error = P_\lambda(max\_1cycle; LSW)$
   end_while
6. $max\_1cycle = max\_1cycle - 1$
7. **return** max_cycles, max_1cycle

The algorithm accepts as inputs LSW, λ and p_ϵ. Lines 1 to 3 compute the maximum number of consecutive ECs that may be affected by one single error. The reasoning is similar to the one used to derive Equation (5.34). As in the Poisson process arrivals are independent, the probability of having exactly one error in n consecutive cycles is given by the product of the probability of having exactly one error in one cycle, given by $P_\lambda(1; LSW)$. Lines 4 to 6 compute the maximum number of errors in one SW, being a direct iteration of the Poisson probability function applied to one SW. Table 5.9 illustrates the results of the algorithm for several scenarios of LSW and λ. One can see that max_cycles and max_1cycle tend to increase with higher values of LSW and λ, as expected.

Table 5.9: Maximum consecutive cycles (max_cycles) with single error and maximum number of errors in one cycle (max_1cycle), for various values of LSW and λ using $p_\epsilon = 10^{-16}$

LSW(ms); λ	***max_cycles***	***max_1cycle***
2.5 ; 0.026	3	3
2.5 ; 0.26	5	4
25 ; 0.026	5	4
25 ; 0.26	7	6

After obtaining these two values, we can build the various error sequences or scenarios, *Error_Scenario*, that produce maximum interference. These are the result of all combinations with length max_cycles and a maximum of max_1cycle errors per EC. For instance, if we consider max_cycles equal to 4 and max_1cycle equal to 3, then the possible error combinations are the ones presented in Figure 5.19. The horizontal-axis in this figure represents the

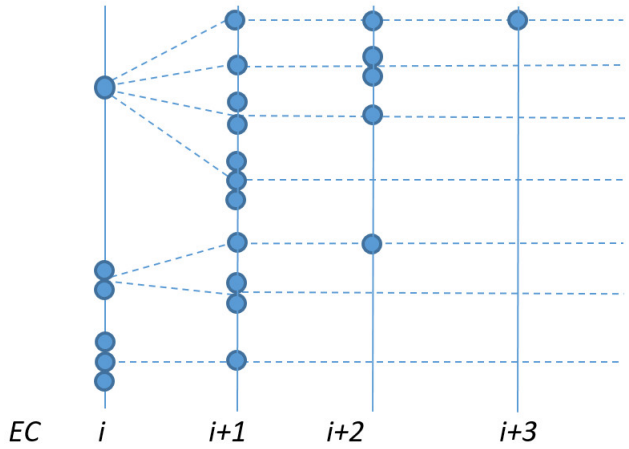

Figure 5.19: Possible error sequences in consecutive cycles.

ECs, while error sequences with probability greater than p_ϵ are represented by a solid circle. These sequences, when combined with the *RepLevel* vector, allow for us to build the set of *Interf_Pattern* required for computing Equation (5.8), as explained in the next section.

5.3.3.4 Server Interference

The error server execution may interfere with any message, having different interference patterns, depending in error sequence and server configuration. As in sections 5.2.3.1 and 5.2.3.2 , we define **Indirect Interference** on one message when this message does not suffer errors itself, but is delayed by the server executing on behalf of other messages, and the **Direct Interference** corresponds to scenarios where one error affects the message being analyzed. When calculating the response time with direct interference we must also account, at first, with the recovery of $(n-1)$ errors (in scenarios of Indirect interference), in a scenario with n errors in total. The worst-case response time for any message is the maximum of both types of interference. As we will see later on, direct interference is normally more penalizing but it is not always the case, thus the need to compute both scenarios.

5.3.3.4.1 Indirect Interference As already stated, the server execution may delay the dispatching of lower priority messages, thus causing interference on them, that is now bigger due to multiple replica sending and the possibility of multiple errors in the previous cycle. As the server is configured to use the highest priority, to minimize retransmissions latency, then all messages are subject to interference due to server execution. Figure 5.20 illustrates the indirect interference caused by the error scenarios presented in Figure 5.19. Possible error sequences in consecutive cycles, including all possible error combinations of max_1cycle errors that can occur in max_cycles cycles, with $RepLevel = \{3, 3, 2, 1\}$, are presented there. Using a smaller number of errors reduces the server load, and consequently, the indirect interference, thus we just need to consider the combinations depicted in Figure 5.19.

Algorithm *Ind_Interf_MultipleReplicas* assesses the schedulability when considering indi-

Figure 5.20: Possible error and recovery scenarios for indirect interference.

rect interference on message set M, having as inputs LSW, LEC, λ, p_ϵ and the $RepLevel$ vector. The parameter cut_errors is an auxiliary variable needed to allow for this algorithm to be used both for the Indirect ($cut\_errors = 0$) and Direct ($cut\_errors = 1$) Interference computation. Lines 1 to 4 determine the $Interf\_Pattern$ array, required to compute the WCRT of all the messages. Firstly Algorithm Max_Errors bounds the number of errors (per cycle and in consecutive cycles), and then the possible sequences of errors are built. Then the $Error\_Scenario$ array (the set of error scenarios) is combined with the vector $RepLevel$ to obtain $Interf\_Pattern$. This vector has size $Max\_Patterns$, which corresponds to the number of error scenarios that must be analyzed. Line 5 computes the values that are needed for the non-preemptive blocking free model (see section 3.3.2.1.1). Lines 6 to 6.1.3 apply the extended schedulability test - Equation(5.8) - to the message set, when considering each one of the error scenarios and interference patterns. If the test fails for any of the patterns, the algorithm returns $Schedulable$ = FALSE (line 6.1.3). Otherwise the algorithm returns $Schedulable$=TRUE together with the response time of each message (line 8), that is expressed in number of ECs (line 7). In fact, the timing granularity of FTT-CAN is the EC duration (LEC) and there is no guarantee on where within an EC a given message will be transmitted, being this done using Equation (5.35), applied to each element in the $Resp\_Time$ vector.

$$R^{EC}_{esp\_Time(i)} = \left\lceil \frac{Resp\_Time(i)}{LEC} \right\rceil \tag{5.35}$$

ALGORITHM Ind_Interf_MultipleReplicas
Response Time with Indirect Server Interference

Input: *M, LEC, LSW, Replevel, Cut_errors,* λ, p_ϵ
Output: *Schedulable (Boolean), RespTime*

1. RUN *MaxErrors* and obtain max_cycles and max_1cycle in LSW
2. $max\_cycles = max\_cycles - Cut\_errors$
3. Build $Error\_Scenario$ array, considering max_cycles and max_1cycle
4. Build $Interf\_Pattern$ array, by combining the $Error\_Scenario$ array and $RepLevel$ vector; $Max\_Patterns$ = number of $Interf\_Pattern$ lines
5. Compute C_{MAX} and obtain M^E, C^E_{MAX}, inflate all transmission times applying Eq. (3.8)
6. **for** p = 1 **to** $Max\_Patterns$ **do**
   - 6.1. **for** i=1 **to** N **do**
     - 6.1.1. Compute R_i (Equation 5.8) considering M^E and with $Interf\_Pattern(p,j) \cdot C^E_{MAX}$ and $Err\_Scenario(p,j) \cdot C^E_{error}$ being added as the cycles progress (j)
     - 6.1.2. **if** ($RespTime(i) < R_i$) **then** $RespTime(i) = R_i$ **end_if**
     - 6.1.3. **if** ($R_i > D_i$) **then** **return** Schedulable = FALSE **end_if**
   - **end_for**
   - **end_for**
7. Transform each $RespTime$ vector value from seconds to number of ECs
8. **return** $Schedulable = TRUE, RespTime$

5.3.3.4.2 Direct Interference In these scenarios, we consider that the WCRT of a given message occurs when that message suffers the maximum indirect interference from the error server, assigned with highest priority, and one error hits the message itself. To reach this conclusion, just consider the following scenario for an arbitrary message m_i:

1. Once ready, m_i suffers the maximum possible indirect interference (both from high-priority messages and from the error server), being scheduled for transmission in EC(k);
2. In EC(k):
   (a) there are no errors; thus, m_i is transmitted at EC(k);
   (b) message(s) other than m_i are affected by errors; thus, m_i is still transmitted in EC(k) (note that errors in EC(k) are handled in EC($k+1$));
   (c) m_i is affected by an error; thus, m_i and its replicas are scheduled for the following EC. The WCRT of m_i is then $k+1$.

Therefore, to assess the schedulability of the message set and obtain the WCRT of the messages considering direct interference, we use the procedure described in Algorithm *Direct_Interf_MultipleReplicas*.

The Direct Interference scenarios corresponding to the error sequences presented in Figure 5.19 are presented in Figure 5.21, where we have to consider a maximum of 3 errors with Indirect interference, plus the one that hits the message under consideration. The direct hit is represented with a "x" mark, that delays one additional EC the message response time.

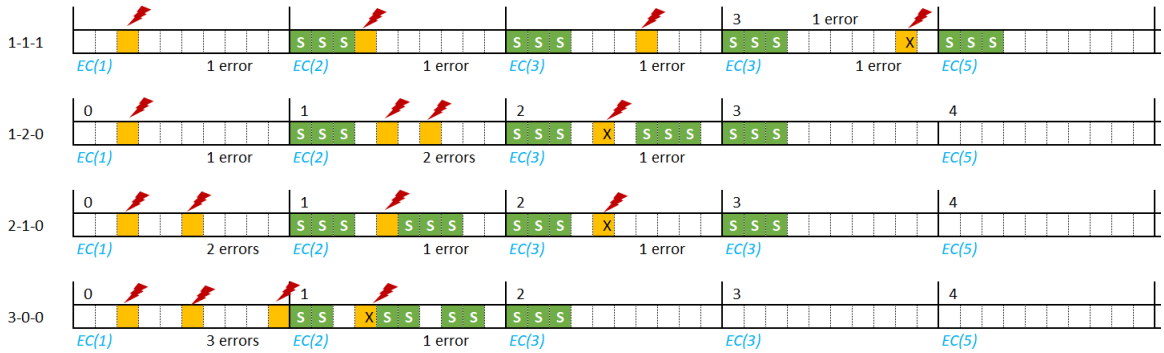

Figure 5.21: Possible error and recovery scenarios for Direct Interference.

ALGORITHM Direct_Interf_MultipleReplicas
Response Time considering errors Direct Interference

Input: *M, LEC, LSW, Replevel* λ, p_ϵ
Output: *Schedulable (Boolean), RespTime_Direct*

1. RUN Algorithm *Ind_Interf_MultipleReplicas* with arguments
   $(M, LEC, LSW, RepLevel, p_\epsilon, cut\_errors = 1)$
   obtaining $Schedulable$ and $RespTime$
   if ($Schedulable == FALSE$) **then**
   return $Schedulable = FALSE$
   end_if
3. **for** i=1 **to** N **do**
3.1. $RespTime\_Direct(i) = RespTime(i) + 1$
3.2. **if** ($RespTime\_Direct(i) > D_i$) **then**
   return $Schedulable = FALSE$
   end_if
   end_for
4. **return** $Schedulable = TRUE, RespTime\_Direct$

Firstly, we execute the Algorithm *Ind_Interf_MultipleReplicas* with a value equal to 1 in the parameter *cut_errors*, because firstly the interference due to indirect errors must be computed with maximum errors minus one, to account for the error directly affecting the message under analysis (Line 1). Remember that this algorithm already returns the response time of each message in numbers of ECs. Then, for each message we assess the impact of

the direct error (Line 3), being necessary to add one EC (Line 3.1) since the message will be recovered in the next cycle relative to the one where the error happens. The deadline violation is verified in (Line 3.2). If the message set is schedulable, the response time of all messages is then returned (Line 4), with the *Schedulable* flag signalling this.

5.3.3.5 System Schedulability Test

To assess the real-time characteristic of a FTT-CAN system with the proposed error recovery method (with multiple replica retransmission), Algorithm *Schedulability_Test* can be used. The inputs are the LEC, LSW, average error arrival rate, mission time MT and global reliability goal.

It starts by obtaining the acceptable message failure probability, p_ϵ (Line 1), immediately followed by the bounds on maximum error per cycle and in consecutive cycles, max_1cycle and max_cycles (Line 2). Using these two values, the error scenarios are built (Line 3), being these combined with the *RepLevel* vector (Line 4) to obtain all the necessary interference scenarios, both Indirect and Direct (in Line 5). Then, in Line 6, the Algorithms *Indirect_Interf_MultipleReplicas* and *Direct_Interf_MultipleReplicas* are used to obtain the worst case response of each message, that will be stored in the $RespTime$ vector, along with the flag $Schedulable$ that is set to TRUE if the message set is schedulable, under the stated conditions. Finally, in Line 7, these two variables are returned.

ALGORITHM Schedulability_Test
Check message set schedulabity for defined LSW

Input: *M, LEC, LSW,* λ, ϵ_G, MT
Output: *Schedulable (Boolean), RespTime vector*

1. Calculate p_ϵ from MT, ϵ_G and M
2. Determine max_1cycle and max_cycles (use Algorithm *Max_Errors*)
3. Build error scenarios
4. Obtain $RepLevel$ vector (use Algorithm *Calc_RepLevel*)
5. Build Interference Patterns - Direct and Indirect
   (Error Scenarios combined with *RepLevel*)
6. Schedulability tests for M^E plus S^E (all Indirect and Direct interference scenarios)
   - individual check for each interference pattern, calculating $RespTime$ vector
7. **return** $Schedulable, RespTime$

5.3.4 Resource Optimization - Obtaining Minimum LSW

An algorithm for minimizing the size of the synchronous window, with the objective to minimize the BW utilization and still achieving the reliability target, is presented next.

The LSW used in the previous algorithms can be optimized, finding a value that makes the system schedulable, when considering all worst case error scenarios. The optimum LSW

is then the minimum value that guarantees that the errors are recovered in the following EC, leaving as much bandwidth as possible to the asynchronous traffic. Algorithm *Minimum_LSW* carries out this optimization using a binary search approach. The algorithm has as inputs the message set M, LEC, the TM transmission time (LTM), $Guard$, λ and $Stop\_criteria$. $Guard$ is a technology-dependent minimum processing time that must be reserved to allow the slave nodes to decode and process the TM. $Stop\_criteria$ is the desired precision of the final result and allows for stopping the iterative process, being expressed as a percentage of LEC. The output is the minimum LSW necessary for making the system schedulable.

Firstly, the absolute lower and upper bounds for LSW are computed - LSW_{low} and LSW_{HIGH}. These are, respectively, C_{MAX} and LEC minus the overheads (LTM and $Guard$). Then, in Lines 3 and 4, the system is tested for feasibility, by giving the maximum time to the LSW. Of course, if the test fails, the system is not schedulable. On the other hand, if the system is feasible, the Bisection or Binary Search method is used to find a solution, using as starting point LSW_{low} and LSW_{HIGH}. As long as the stop criteria is not met (Line 5), the interval is halved (Line 5.1) and the schedulability assessed with this value as input, using the Algorithm *Schedulability_Test* (Line 5.2). If this test fails the intermediate point becomes the new lower bound for LSW, otherwise it becomes the upper bound (lines 5.3 and 5.4, respectively). Then, the process is iterated using the new bounds. When the while loop condition is evaluated FALSE, the cycle finishes and the minimum value that makes the system schedulable has been found, being returned in Line 6.

ALGORITHM Minimum_LSW
Obtain minimum LSW for a schedulable system

Input: *M, LEC, LTM, Guard,* $\lambda, Stop\_criteria$
Output: *LSW*

1. $LSW_{low} = C_{MAX}$
2. $LSW_{HIGH} = LEC - (LTM + Guard)$
3. $Sch\_High =$ Test Schedulability using LSW_{HIGH}
4. **if** ($Sch\_High == FALSE$) **then**
   return -1
   end_if
5. **while** ($LSW_{HIGH} - LSW_{low} > Stop\_criteria \cdot LEC$) **do**
5.1 $LSW_{test} = \frac{LSW_{HIGH}+LSW_{low}}{2}$
5.2 $Sch\_test =$ Test Schedulability using LSW_{test}
   if ($Sch\_test == FALSE$) **then**
5.3 $LSW_{low} = LSW_{test}$
   else
5.4 $LSW_{HIGH} = LSW_{test}$
   end_if
   end_while
6. **return** LSW_{HIGH}

5.4 Summary

In this chapter it was described:

- Error recovery with single replica retransmission, describing several policies when configuring the server;
- Error recovery with Multiple replica, server parameter choosing and obtaining *RepLevel*;
- Resource optimization - choosing minimum LSW.

.

In next chapter an assessment of the proposals is performed and presented.

Chapter 6

Simulation Study and Partial Experimental Validation

This chapter presents a simulation study of the proposal, describing firstly the simulator and giving initial results with single replica recovery. Along with the assessment of the Deferral Server type, that was our first choice for the error server, it was also evaluated the use of a Polling or a Sporadic Server. The results obtained with the Deferrable Server confirmed the limit on error recovery ratio obtainable with single replica retransmission, implying the introduction of multiple replica retransmission. Then, the updated simulator version, that also includes a fault generator with rare scenarios, is described and the results obtained are compared with other error recovery methods.

The developed simulator was built from scratch using MatLab and is specifically tailored for testing the error-recovery mechanisms proposed in this thesis.

6.1 Simulator - Single Replica Version

Simulation is a well-known and widely used technique for assessing and validating an immense variety of engineering problems, in any field. For the particular case of error recovery protocols, simulation is invaluable, given the difficulty of reproducing error scenarios that occur rarely, thus making experiments with real hardware extremely hard, cumbersome and long. It also has the advantage that the simulation ambient and characteristics, including the faults, can be reproduced, allowing the evaluation of the different methods in exactly the same conditions. Finally, the simulation in software can be performed in less time when compared to a test with hardware subject to a real environment.

The assessment of the proposed methods for error recovery was done using a discrete event simulator. The time advance is triggered by all system events, that produce changes in the internal states. For the FTT-CAN simulator the events of interest are:

- scheduling and TM generation;
- TM processing;

- message start;
- message finish;
- bus fault.

The scheduling events have the time instants fixed at the start of the simulation and are separated from each other by LEC seconds. Each time a scheduling event code is run, it generates the TM, specifying what messages should be transmitted in the SW of the following elementary cycle and LSW duration. The scheduling process must also include the scheduling of messages present in the *Error Server Queue*, if any.

The TM processing events immediately follow each scheduling event and their execution will define additional events corresponding to message transmit instants - start and finish - of each message marked in the TM, that will be inserted in the global event list, ordered by time instant. Of course, all these new message instants will precede the next scheduling event. The message start event just sets a flag that signals that a message is currently present in the bus. The message finish event just clears this flag.

The bus fault events correspond to single bit errors in the bus and are generated before the simulation starts. The time distance between consecutive events follows a exponential distribution, with average distance equal to $1/\lambda$ seconds. This is derived from the fact that the fault model considered has Poisson distribution with intensity λ. The processing of each bus fault event corresponds simply to check if the fault event coincides with a message transmission and which one it is. Then, the corresponding message ID is inserted in the *Error Server Queue*, that will be further processed at the next scheduling event.

The **implemented simulator** is graphically described in Figure 6.1, and implemented from scratch in MatLab.

The Algorithm *FTT-CAN_TimeSimulator* presents the simulator major processing steps, which are further described in the followings paragraphs.

The first step consists in loading a text file that describes the benchmark. This file starts with the FTT-CAN network configuration parameters, that includes the CAN bit rate (in kbps), the EC length and LSW (Length of Synchronous Window). Then, it follows the message set, defined by the number of messages, their periods, deadline, offsets, sizes (DLC - CAN payload size in bytes). The message characteristics are organized in lines, being assumed that the messages priorities are sorted by decreasing order.

At step two, the simulator internal structures are initialized, including the simulation length (in number of ECs), the server type, capacity and period and also the fault intensity. All other variables necessary to gather statistical information on all the details on error occurrence and recovery, are also initialized in this step.

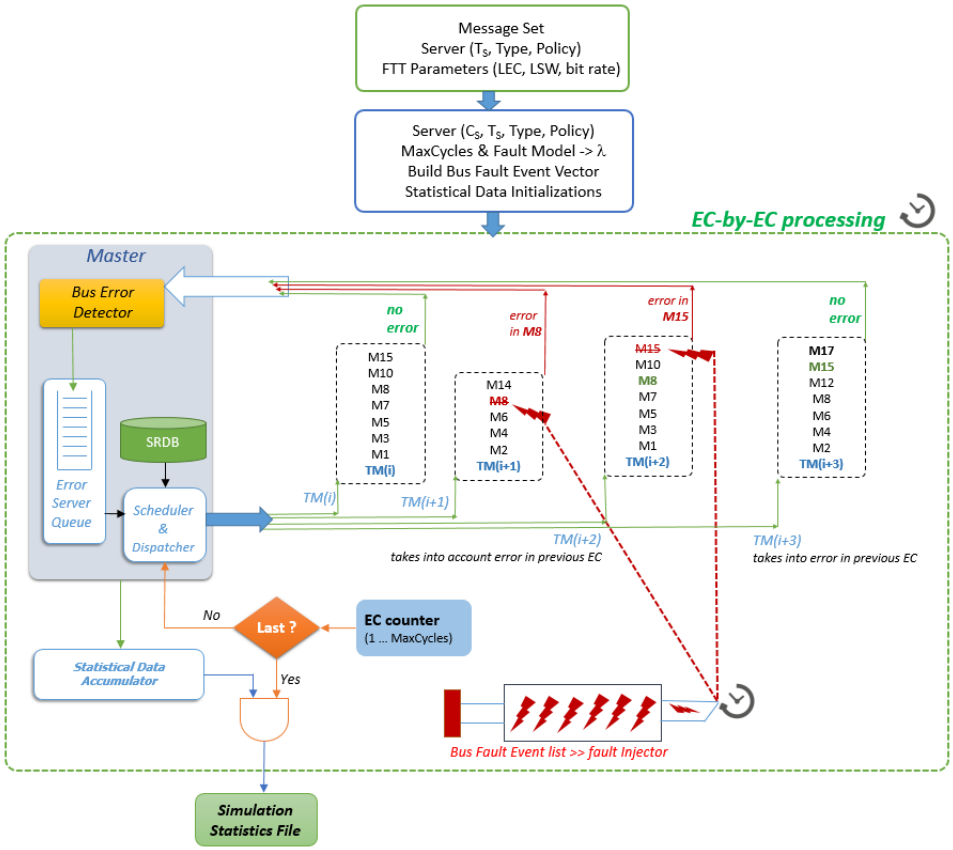

Figure 6.1: Simulator in MatLab.

Algorithm *FTT-CAN_TimeSimulator*

Input: Text file with message set, network configuration, Server type and parameters, BER (or λ)
MaxCycles, number of cycles to simulate (or time)

Output: Text file with collected statistics

1. LOAD network parameters, Server configuration, BER
   and message set from the input file
2. Initialize all internal data structures
3. Construct the bus faults event vector according to the error model and λ
4. **for** EC_count = 1 **to** MaxCycles **do**
4.1 Obtain schedule for next EC
4.2 Bus processing
   end_for
5. Write all statistical data gathered to a text file

The bus fault event vector is generated in step three. The fault instants are generated according to the Poisson distribution, with average fault inter-arrival time equal to $1/\lambda$, as previously described. To speed up simulation, each position in the bus fault event vector contains a structure with two fields: the EC number and the time instant, inside each EC, where the fault instant event occurs.

Step four is repeated *MaxCycles* times, once for each simulated EC. In this step the traffic is scheduled (step 4.1) and processed (step 4.2). The traffic is scheduled on a EC-by-EC basis, according to the rules defined by the FTT-CAN protocol. The scheduling process is performed by the scheduler module, that in the current implementation uses a Fixed Priority policy. This module generates the TM for the current EC, including the recovery of any pending messages inside the *ESQ*, checking and updating statistics on any deadline misses in the scheduling process. The bus processing (step 4.2) starts by checking if the next bus fault event is in the current EC. If this condition is false, a no-error scenario in this EC happens and the simulation time can be advanced to the next scheduling event. If true, then the TM data is used to generate the start and finish instants of the transmitted messages in the current EC. Then by cross-checking the fault instant with each message transmission start/finish events permits the identification of the message that suffers an error, which ID enters the *Error Server Queue*. Also, all relevant statistical counters are updated. In the current implementation, the errors are managed by a Deferrable Server with period T_S and capacity C_S, that are defined in the input file. By gathering all statistical information, every time a significant event is processed, a report file is built (step 5), which includes all relevant information about the message set used and statistics, namely: "measured" BER, number of errors and missed deadlines and the worst case message response times. It also has information about error incidence, missed deadline and WCRT per message. Other relevant information present in this file is: number of errors in consecutive cycles (total and incidence

on messages subject to recovery), distribution on number of errors per server period, server response time per message (worst case and average). The response time of each message is counted, for messages subject to direct and indirect interference. An example of one of these output files is provided as a text file, in the accompanying CD-ROM.

6.1.1 Used benchmarks

To test and assess the error recovery methods, at first, we used three benchmarks. Two of them were chosen because of relevance first, being a good representation of commercial cars. The third benchmark, VEIL, belongs to an electrical vehicle prototype, in the development of which the author of this thesis collaborated. The benchmarks are presented and characterized in the following sections.

6.1.1.1 *Updated_SAE* benchmark

The original SAE benchmark was firstly presented in [Int93], intended to be representative of an automotive communication system, in a prototype electric car. Since then this benchmark has been used in various studies and papers, but with new functionalities implemented in modern cars, as for instance Departure Lane, Adaptive Cruise Control and Traction Control, reveals itself outdated. A modified *SAE* benchmark was presented in [MN10], that introduces new functionalities present in modern cars. As this corresponds to a more "real" car, we use it instead of the original benchmark, being the corresponding message set the one presented in Table 6.1. More details on the *SAE* and *Updated_SAE* benchmarks are given in Appendix A, tables A.1 and A.2.

This benchmark presents a message set with $C_{MAX} = 115$ bits, minimum period and deadline both equal to 5 ms and a utilization of 27.9% when using a bit rate equal to 1 Mbps.

6.1.1.2 *PSA* benchmark

The *PSA* benchmark is presented in [CSSLC00] and corresponds to a network of a research vehicle developed by the PSA Peugeot-Citroën Automobile Company. The message set, is presented in Table 6.2. In the table A.3 of Appendix A, further details are presented regarding this benchmark, which are not relevant to our simulation study.

This benchmark message set presents $C_{MAX} = 135$ bits, minimum period and deadline both equal to 10 ms and an utilization of 11.9% when a bit rate equal to 1 Mbps is used.

6.1.1.3 *VEIL* benchmark

A small electric vehicle prototype - the VEIL - was developed in ISEC/IPC, being its architecture firstly presented in [STM+06]. It uses a single CAN bus, with the FTT-CAN protocol, to transmit all messages that control and monitor the vehicle functioning. The message set is presented in Table 6.3. More details on the VEIL project and vehicle power and communication architecture are presented in Appendix A, in Section A.4.

Table 6.1: *Updated_SAE* benchmark message set

ID	DLC (bytes)	T (ms)	T (LEC)	D (ms)	ID	DLC (bytes)	T (ms)	T (LEC)	D (ms)
1	1	50	20	5	19	6	10	4	10
2	2	5	2	5	20	2	10	4	10
3	1	5	2	5	21	3	10	4	10
4	2	5	2	5	22	2	10	4	10
5	1	5	2	5	23	2	12.5	5	12.5
6	2	5	2	5	24	2	12.5	5	12.5
7	1	5	2	5	25	2	12.5	5	12.5
8	1	5	2	5	26	2	12.5	5	12.5
9	1	7.5	3	7.5	27	4	12.5	5	12.5
10	1	7.5	3	7.5	28	5	12.5	5	12.5
11	1	7.5	3	7.5	29	3	12.5	5	12.5
12	1	7.5	3	7.5	30	1	50	8	20
13	1	7.5	3	7.5	31	4	100	40	100
14	4	7.5	3	7.5	32	1	100	40	100
15	4	7.5	3	7.5	33	1	100	40	100
16	4	7.5	3	7.5	34	3	1000	400	1000
17	1	10	4	10	35	1	1000	400	1000
18	2	10	4	10	36	1	1000	400	1000

Table 6.2: *PSA* Benchmark message set

Frame ID	DLC (bytes)	T (ms)	T (LEC)	D (ms)	Frame ID	DLC (bytes)	T (ms)	T (LEC)	D (ms)
1	3	10	2	10	13	8	50	10	50
2	5	10	2	10	14	8	50	10	50
3	5	10	2	10	15	8	50	10	50
4	8	10	2	10	16	1	100	20	100
5	2	15	3	15	17	6	100	20	100
6	4	15	3	15	18	7	100	20	100
7	3	20	4	20	19	7	100	20	100
8	4	20	4	20	20	7	100	20	100
9	5	20	4	20	21	2	150	30	150
10	5	40	8	40	22	4	150	30	150
11	5	50	10	50	23	4	200	40	200
12	5	50	10	50					

Table 6.3: *VEIL* benchmark message set

ID	DLC (bytes)	T (ms)	T (LEC)	D (ms)	ID	DLC (bytes)	T (ms)	T (LEC)	D (ms)
1	2	10	2	10	11	8	100	20	100
2	2	10	2	10	12	1	250	50	250
3	4	10	2	10	13	2	500	100	500
4	1	20	4	20	14	2	500	100	500
5	2	20	4	20	15	1	1000	200	1000
6	4	20	4	20	16	1	1000	200	1000
7	2	50	10	50	17	2	1000	200	1000
8	3	50	10	50	18	8	1000	200	1000
9	4	100	20	100	19	8	1000	200	1000
10	8	100	20	100					

This message set presents $C_{MAX} = 135$ bits, minimum period and deadline both equal to 10 ms and an utilization of 4.4% when a bit rate equal to 1 Mbps is used.

6.2 First Results with Poisson Model (limit of 1 fault per EC)

The first simulation runs were performed to assess the flawless working of the proposed method and additionally verify that the bandwidth used by the server is really small.

We started by choosing the Server period equal to the average time between errors, which is equal to $1/\lambda$ seconds and starting with C_S equal to one maximum message and increasing it until no error was observable. Table 6.4 presents the average results of ten simulation runs, using the *Updated_SAE* benchmark, with a bit rate equal to 1000 kbps, BER of $2.6 \cdot 10^{-7}$ (Agressive environment), 10 million ECs and a LEC equal to 2.5 ms (representing roughly seven working hours per simulation run and a total of 70 hours). It was used a Deferrable Server with maximum priority.

Table 6.4: Simulation results with a Deferrable Server for the *Updated_SAE* benchmark - C_S/C_{MAX} varying from 1 to 10

C_S/C_{MAX}	Unrecovered Errors	Message Error Recovery Ratio
1	225.1	87.592%
2	18.5	98.981%
3	1.2	99.934%
4	0.1	99.995%
5...10	0.0	100.000%

From this table it is observable that configuring the server capacity with $5 \cdot C_{MAX}$ and a period of $1500 \cdot LEC$ (approximately $1/\lambda$), allowed to obtain full error recovery using only

0.0153% of the available bandwidth, (with C_{MAX} = 115 bits and LEC = 2.5 ms) which is indeed a very small value.

To gain further confidence on the obtained results more simulations were performed, using a fixed server capacity equal to $5 \cdot C_{MAX}$. Then, using different seeds for the pseudo-random generator, 10 more simulations were performed (equivalent to an extra 70 working hours) and full error recovery was always achieved. Looking to the numbers gathered in the output simulator log files, it was observed that in 19 of them using C_S equal to $4 \cdot C_{MAX}$ was enough to attain a 100% recovery ratio and only in one simulation, and only one time, 5 errors in the server period happened, being this the only case where all server capacity was used. The two other benchmarks - *PSA* and *VEIL* - were also simulated in identical situations and the C_S/C_{MAX} were both 3, corresponding to a server configured bandwidth of 0.0108% (with C_{MAX} = 135 bits). Tables with simulation information as in Table 6.4 are presented for these two benchmarks in Appendix B, corresponding to tables B.1 and B.2.

A second point that we want to observe was how the bandwidth allocated to the server varies with different values of T_S and on the heuristic chosen for this parameter value - see Section 5.2.2.2. We have performed several simulations, for different (T_S, C_S) pairs, by firstly fixing T_S and increasing C_S until full error recovery is reached. The results are presented in Table 6.5, corresponding each table position to the average value of 10 simulation runs, for each (T_S, C_S) pair. The last line presents the minimum server allocated bandwidth, being represented as a percentage of the system available bandwidth in the same period, where we can see that the server bandwidth grows with decreasing server periods.

The values presented for the Server BW showed that the proposed method can achieve very low bandwidth overhead, less than 0.1% of the available bandwidth, in the evaluated cases. The considered server periods are much greater than the needed server responsivity (recovery in the following cycle), showing a decoupling between server period and responsitivity, as expected for this server type.

Table 6.5: Message recovery ratio with different (C_S, T_S) combinations for *Updated_SAE* benchmark.

	T_S/LEC			
C_S	250	500	750	1500
1	97.843%	95.806%	93.432%	87.592%
2	99.973%	99.863%	99.700%	98.981%
3	**100.000**%	99.986%	99.959%	99.934%
4	100.000%	**100.000**%	99.986%	99.995%
5...10	100.000%	100.000%	**100.000**%	**100.000**%
Server BW	0.0552%	0.0368%	0.0307%	0.0153%

Simulations performed for the other benchmarks allow us to obtain similar tables, that are presented in Appendix B, corresponding to tables B.3 and B.4, showing identical generic results.

Another interesting question is how the error recovery method behaves in case of higher BER than the one anticipated by the fault model and by how much it is necessary to raise the server capacity to be able to deal with a greater number of errors (per server period). We started by simulating again the *Updated_SAE* benchmark, but now using BER ten times bigger than what is considered an Aggressive Environment. The obtained results show that almost full-error recovery is possible using a server capacity at least equal to $12 \cdot C_{MAX}$, as presented in Table 6.6.

This corresponds to more than doubling the server allocated bandwidth, but for a ten fold increase in the average fault incidence, when compared to the previous results. The increase in server capacity is necessary since the error arrival is now greater and the server does not have sufficient capacity to recover all errors, being the more demanding simulation one that presented 12 errors in one server period, thus imposing a minimum C_S equal to $12 \cdot C_{MAX}$. Nevertheless, this value is still small - corresponding to 0.0368% of the available bandwidth - and could be an acceptable price to pay to cope with situations where the expected scenarios could be, at least for short periods of time, much worst than the expected ones.

Table 6.6: Error recovery ratio as a function of the server capacity for higher than Aggressive environment (10 runs with 10 Mcycles each, $T_S/LEC = 1500$ and $\lambda = 2.6$ faults/second)

C_S	Missed Deadlines	Message Error Recovery Ratio
1	11901.0	34.352%
2	6871.8	62.095%
3	3463.6	80.896%
4	1519.1	91.621%
5	589.1	96.751%
6	204.0	98.875%
7	65.1	99.641%
8	18.5	99.898%
9	4.8	99.974%
10	1.6	99.991%
11	0.9	99.995%
12 ... 25	0.8	99.996%

A second observation from these results is that full error recovery is not attained even for large capacity values, remaining a residual number of non-recovered errors present (it happened seven times out of 10 simulation runs). This was not observed in the simulations with the BER of Agressive Environment, where full-error recovery was observed in all the simulations. Looking carefully to the report files (with a BER 10 times greater than the one for Aggressive environment), we identified several scenarios where a message under recovery also suffers an error, leading to an addditional attempt to recover. For messages with short deadlines (specially for messages with periods/deadlines equal to 2 ECs) this attempt can not be performed leading to deadline miss for these messages.

On the other hand, in the 20 simulations performed with *Updated_SAE*, with the BER of Aggressive Environment, the occurrence of errors in consecutive cycles was restricted to 4 occasions and it was not observed any scenario where the second error hits a message being recovered. This was the reason why we considered that a server with capacity equal to $5 \cdot C_{MAX}$ was enough to obtain 100% error recovery success, as presented in Table 6.4.

To verify that this situation was not particular to the *Updated_SAE* benchmark, simulations with the other two benchmarks were performed, using a server period equal to $75 \cdot LEC$, being the error recovery ratio presented graphically in Figure 6.2. Again, the residual failure in error recovery is present, being its existence justified by successive errors affecting a message instance and its replicas, as explained before. Also, the obtained results for these simulations were organized as in Table 6.6, being presented in Appendix B - tables B.5 and B.6.

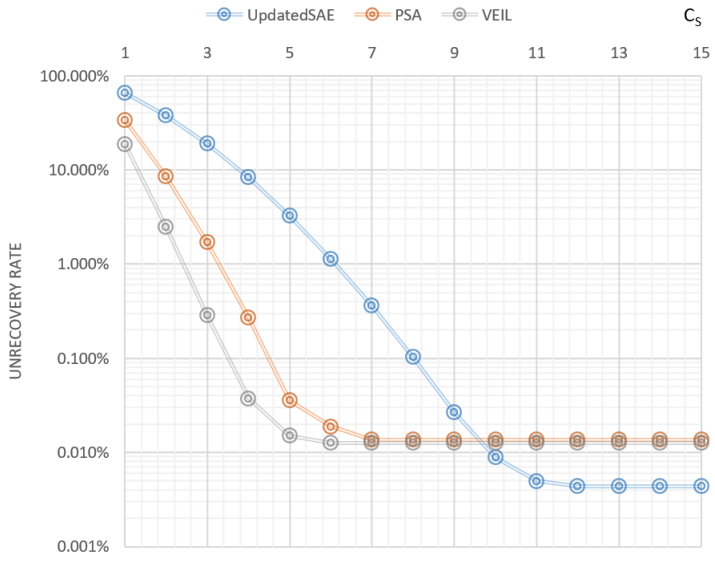

Figure 6.2: Limitation on achieving full error recovery

The limitation on obtaining full-error recovery was confirmed and must be overcome if the error recovery method is to be used in high-reliability systems, as obtaining this reliability level is clearly hindered by the identified error-and-recovery scenarios.

6.2.1 Server Policy

In this section we analyze the performance of different server types and some possible tweaking/priority assigning of the Deferrable Server.

6.2.1.1 Assessing the Polling Server

In the previous section a Deferrable Server was used, but other server types can also be adopted. In virtue of its simplicity and predicted interference pattern on lower priority messages, Polling Servers have been considered for evaluation. As the fault model limits the

number of faults to one, the simulation study will use a fixed value for the server capacity - $C_S = C_{MAX}$ - and varying the period T_S to observe the result in terms of recovered messages. Using the same configuration as in the previous simulations that led to Table 6.4 (except T_S value and server type), a summary of the results is presented in Table 6.7 corresponding to the average value of ten simulation runs for the first line and one run for the others.

Table 6.7: Polling server T_S choosing, with *Updated_SAE* benchmark, bit rate = 1000 kbps, BER = $2.6 \cdot 10^{-7}$ and 10 million cycles, LEC = 2.5ms.

T_S/LEC	Missed Deadlines	Message Error Recovery Ratio
1	0	100.00%
2	628	64.2%
5	1313	30.4%
10	1651	10.3%
100	1830	0.7 %

The most striking observation is that "full" error recovery can only be obtained with small server period, being equal to one LEC (first line) for the simulated benchmark and degrading fast with increasing T_S. Note that for T_S/LEC equal to 100, only less than 1% of the errors were recovered. This is due to the coupling effect between server activation and message period/deadlines, as the server must possess a period equal or less than any message period that needs to be recovered. If this relation is not verified, then a message that suffer an error is correctly placed in the *Error Server Queue* but may loose its deadline while waiting in the queue for the next activation instant of the Polling Server (that will occur too late).

Note also that increasing the server capacity would not solve the problem, as the server capacity can only be used in the cycle when its activation occurs and not along all of its period, contrary to the Deferrable Server, which can be activated at any time, as long it has available capacity.

As a mater of fact, full-error recovery would also be obtained with period equal to $2 \cdot LEC$, being in this case necessary to introduce an offset equal to LEC in the server activation instant. This is only effective if we can guarantee that all the messages with period equal to 2 ECs are scheduled in the EC where they become ready, and not latter. Simulations were performed that confirmed these details.

Comparing the necessary bandwidth for the Polling against the Deferrable Server, to obtain full-error recovery, the value is significantly greater, being equal to 4.6% ($C_{MAX} = 115\mu s$, LEC = 2.5 ms) against only 0.0153% for the Deferrable Server type (referring to the *Updated_SAE* benchmark). Similar results can be drawn for the two other benchmarks, being the simulation results presented in Appendix B, tables B.7 and B.8.

6.2.1.2 Assessing the Sporadic Server

Another interesting candidate for the Error Server is the Sporadic Server, that has the advantage of not imposing a penalty on schedulability bound, contrary to the Deferrable Server, as explained in Section 2.5.3. We have performed simulations with the same parameters as with the Deferrable Server and the results for the *Updated_SAE* benchmark are presented in Table 6.8.

Table 6.8: Error recovery ratio as a function of the server capacity of a Sporadic server, with *Updated_SAE* benchmark, bit rate = 1000 kbps, $T_S/LEC = 1500$, λ=2.6 and 10 million cycles, LEC = 2.5ms.

C_S	Missed Deadlines	Message Error Recovery Ratio
1	13344.0	26.402%
2	9977.3	44.971%
3	5732.7	68.382%
4	3207.0	82.312%
5	1467.2	91.908%
6	555.5	96.936%
7	146.7	99.191%
8	31.4	99.827%
9	18.2	99.900%
10	5.1	99.972%
11	2.5	99.986%
12.. 25	1.1	99.994%

For the other benchmarks, similar tables are presented in Appendix B - Tables B.9 and B.10.

The results shown are slightly worse when compared with the Deferrable Server, namely when the capacity is not enough to recover all the expected errors per server period. This suggests that there are time intervals where, due to high error numbers, the Deferrable Server uses it capacity back-to-back, but the Sporadic one is prevented to do that due to more strict replenishment rules. The bandwidth allocated, to obtain residual non-recovery is identical to the one of the Deferrable Server, so it is also a good choice for the Error Server. Nevertheless, due to the way that the capacity is managed, its implementation is more complex and computationally demanding than the Deferrable Server one, being necessary to track every chunk of used capacity separately (see 2.5.2 or [But11] for details).

6.2.2 Priority Assigning to the Deferrable Server

Regarding the error recovery server characteristics, the pair (C_S, T_S) defines its ability to recover all the expected errors, but the priority assigned to it should not be overlooked since it has implications on how fast the messages are recovered, and also on the interference on

regular messages (not hit by errors). In this section we analyze this perspective, trying to give some clues on the trade-offs on server priority choosing.

Moreover, we present a set of simulation results that aim at evaluating the performance and correctness of the proposed algorithms, described in Section 5.2.4. To this end, in addition to the more obvious numeric results regarding the number of recovered and non-recovered errors, we also include numerical results that allow assessing the response-time and internal effects of the diverse server scheduling policies.

The simulator used in the previous section was firstly designed to use a Deferrable Server with maximum priority, only. It was then necessary to update the server code to allow the use of the new priority assigning schemes, allowing this way to assess and compare them.

The evaluation was done using the three benchmarks already used in previous sections, that were described in Section 6.1.1. The general results obtained with the various message sets are analogous so, assuming an approach identical to the one done previously, more details are given for the *Updated_SAE* benchmark, being presented compact information on the two other benchmarks. Nevertheless, more details for the two other benchmarks are presented in Appendix B, Section B.2.

6.2.2.1 Recovered Errors and Interference

For each configuration, we start with the minimum LSW that allows the transmission of messages within their deadlines, without considering bus faults. Afterwards we generate the fault pattern, using a BER 10 times greater than the one of Aggressive environment [FAFF04] and successively increased the LSW until obtaining no errors due to insufficient bandwidth. The fault intensity was chosen above Aggressive environment in order to stress the error recovery mechanism, since initial simulations with this environment did not present a significant number of errors, what would not allow to draw any relevant conclusions.

Table 6.9 presents the deadline misses suffered by messages affected by errors, for the diverse priority assignment policies. The presented results correspond to the average value of ten simulation runs, each one done for 10 million ECs, where the BER used in the fault generator is ten times higher than the one corresponding to Aggressive environment in order to get a significant number of errors per simulation, as explained. In this case the server was configured according to the rules exposed in Section 6.2.2, with the fault incidence $\lambda = 2.6$ faults/second and choosing $T_S = 1/\lambda$ (close to this value), obtaining the (T_S, C_S) pair equal to $(150 \cdot LEC, 5 \cdot C_{MAX})$. Of course, parameter choosing was done so the capacity is never completely depleted in each server period.

Table 6.9: Deadline misses in messages hit by errors with *Updated_SAE* benchmark (LSW=36.5%), average with 10 simulation runs

	MaxPriority	*SamePriority*	*SameDMP*	*ServerEDF*
Total misses	1.5	73.6	1.8	1.4
Unrecoverable misses	1.5	1.6	1.7	1.4

The first line shows the total number of deadline misses, while the second one shows the number of errors that affected messages that had no slack, and thus that were not recoverable by any of the mechanisms. We can observe that for *MaxPriority* and *ServerEDF* priority assignment policies all misses correspond to unrecoverable messages, i.e. messages that suffer an error in the EC in which they have their deadlines. For the *SamePriority* policy, as the server inherits the original message priority, retransmissions can be delayed by several cycles and eventually lose their deadlines, as shown in the scenario depicted in Figure 5.7. This situation happened in fact, as attested by the relatively large number of deadline misses, when compared to the other policies. In fact, the simulation output files have information on messages that loose their deadlines while waiting in the error server queue, that accounts exactly these situations. In the ten simulations performed the *SameDMP* policy has resolved this issue, but this is not a guarantee. Clearly, the postponing of the message recovery instant leads to smallest intervals where the recovery is possible, and error hitting on the last feasible EC can not be ruled out.

We also performed simulations with larger LSW than the minimum value (that was found by trial-and-error) and the obtained results are more favorable (less errors with *SameDMP* politic). For instance, with an LSW equal to 39.6%, there are no differences in the numbers of missed deadlines of all the policies.

So it can be inferred that the number of unrecovered errors depends strongly on the bus load and this could be explained by the fact that the server execution does not cause a significant interference on normal messages, when there is more slack time (in average, per EC).

In Table 6.10 the WCRT of every message in this benchmark is shown, separating the values for messages that suffered errors (Direct Interference) and not hit by errors (Indirect Interference). To put in perspective the obtained numbers in each situation, the values corresponding to an error-free environment are also displayed in the first column.

The most noticeable in this table is that for the first 20 messages the Direct Interference increases the response time by one EC, in all policies, and that this interference increases for lower priority messages, except for *MaxPriority*, for which it is always one EC. Regarding Indirect interference scenarios, for almost all messages with ID lower than 23 there is no increase in response time, degrading slightly for the remaining messages (1 or 2 ECs for half of the remaining 14 messages). Generically speaking, this confirms that Direct interference implies a bigger degradation on the response time than Indirect interference.

Finally, Table 6.11, presents the average server response times for each server scheduling priority assignment policy. We can see that the *MaxPriority* policy shows the lowest value, always equal to one, as expected, since the server has the highest priority, therefore performing the recovery always in the next cycle. There are a few exceptions, too small to cause visible effects in this table, which correspond to scenarios in which the first error recovery attempt is not successful. These cases correspond to scenarios were a message retransmission is also hit by another error (error in consecutive cycles). On the other hand, the *SamePriority* and *SameDMP* policies show the worst performance, being *ServerEDF* a good compromise

Table 6.10: WCRT for all tested policies

msg	no error	Direct				Indirect			
		Max Priority	Same Priority	Same DMP	Server EDF	Max Priority	Same Priority	Same DMP	Server EDF
1	1	2	2	2	2	1	1	1	1
2	1	2	2	2	2	1	1	1	1
3	1	2	2	2	2	1	1	1	1
4	1	2	2	2	2	1	1	1	1
5	1	2	2	2	2	1	1	1	1
6	1	2	2	2	2	1	1	1	1
7	1	2	2	2	2	1	1	1	1
8	1	2	2	2	2	1	1	1	1
9	1	2	2	2	2	1	1	1	1
10	1	2	2	2	2	1	1	1	1
11	1	2	2	2	2	1	1	1	1
12	1	2	3	3	3	2	1	1	2
13	1	2	2	2	2	2	1	1	2
14	2	3	3	3	3	2	2	2	2
15	2	3	3	3	3	2	2	2	2
16	2	3	3	3	3	2	2	2	2
17	2	3	3	3	3	2	2	2	2
18	2	3	3	3	3	2	2	2	2
19	2	3	3	3	3	2	2	2	2
20	2	3	3	3	3	2	2	2	2
21	2	3	4	4	3	2	2	2	2
22	2	3	4	4	3	2	2	2	2
23	2	3	4	4	3	3	3	3	3
24	3	4	4	5	5	3	3	3	3
25	3	4	4	4	4	3	3	3	3
26	3	4	5	5	4	3	3	3	3
27	3	4	5	5	4	4	4	4	4
28	3	4	5	5	4	4	4	4	4
29	4	5	5	5	5	4	4	4	4
30	4	5	7	7	7	6	6	6	6
31	4	5	8	8	8	6	6	6	6
32	6	7	8	8	8	6	6	6	6
33	6	7	8	8	8	8	8	8	8
34	6	7	8	8	8	8	8	8	8
35	8	9	10	10	10	8	8	8	8
36	8	9	10	10	10	8	8	8	8

between these two extremes, as this last one shows faster response, but only for a subset of the messages (with IDs from 21 to 28). We can also observe that all the policies present a similar behavior regarding the high-priority messages (until message 20), which fit in the LSW even when the server executes, as expected. With *SamePriority*, *SameDMP* and *ServerEDF*, messages with lower priority than these ones see their recovery time increasingly degrading. This behavior was expected since, for these policies, the recovery can be postponed to a later EC, which is something that can not happen with *MaxPriority*.

In Appendix B, Section B.2 similar tables are presented for the two other message sets, with identical generic results.

Table 6.11: Server average response time - all policies

Policy/Message	**1...20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
MaxPriority	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
SamePriority	1.00	1.07	1.33	1.51	1.50	1.48	1.50	1.50	1.68
SameDMP	1.00	1.07	1.33	1.51	1.50	1.48	1.50	1.50	1.68
ServerEDF	1.00	1.00	1.00	1.15	1.16	1.16	1.16	1.18	1.18
Policy/Message	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**	
MaxPriority	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
SamePriority	1.80	2.36	2.65	2.70	2.63	3.03	2.55	2.67	
SameDMP	1.67	2.36	2.65	2.70	2.63	3.03	2.55	2.67	
ServerEDF	1.16	2.36	2.65	2.70	2.63	3.03	2.55	2.67	

6.2.2.2 Final remarks on priority assigning policies

In this section a numerical evaluation of the proposed algorithms, obtained via simulation, was performed for diverse priority assignment policies of the recovery server. The evaluation assessed the recovery response time, the ratio of the recovered messages, including the identification of scenarios where obtaining 100% recovery ratio is not possible.

It seems adequate that the error recovery policy that should be used must have a fast recovery time, give maximum guarantees in recovering all errors and minimize the WCRT of all messages. From the quantitative results of the previous section and the qualitative analysis of the preceding chapter, the best compromise seems to be *MaxPriority*. It also has the advantage of possessing the simplest algorithm, that also implies that its implementation is simple, runs fast and uses few resources.

6.3 Recovery Method with Multiple-Replica Retransmission

This section starts by first detailing the updated simulator version, including the new fault generator/injector and multiple replica retransmission, that allows the assessment of the proposed method with rare error scenarios, in a reasonable amount of time. The obtained results allows the comparison of the proposed method against other methods presented in

the literature. A method to optimize the LSW is also presented, assessed and compared with the results of other contending methods.

6.3.1 Updated Simulator Description

The simulator used in Section 6.1 uses a fault generator which as a restriction on the number of faults, one per EC, which excludes the possible occurrence of the fault scenarios that, despite less frequent, have to be considered in high-reliability systems. Even if this restriction is removed and a Poisson distribution is used, the simulation runs would have to be carried for a very long time, just to obtain a few instances of these scenarios, which is impracticable. This is due to the fact that these scenarios have extremely low probabilities, e.g. the probability of finding four errors in one cycle of 2.5ms is of the order 10^{-14}, thus appearing in average 1 time each 10^{14} ECs, making the simulation impractical, due to excessive simulation time necessary. It is then clear that a new fault generator and injector is necessary to create a considerable number of these rare fault scenarios in a reasonable amount of time.

The second limitation presented by the previous simulator implementation was that it only allowed single-replica retransmission (that limits the achievable transmission reliability), which is not appropriate to assess the updated error recovery method that uses multiple replicas.

The new simulator version used in this section is an extended version of the original simulator that was used in Section 6.1, that was extended to adress the limitations above identified. The updated simulator version includes the following new features:

- error detection process supports multiple errors per EC;
- fault generator and injector include random sequences of rare fault scenarios;
- server and scheduler functions now include the retransmission replication level;

Another addition to the implemented software package is a module for LSW optimization using binary search, that can be run separately of the simulator, and implements the Algorithm presented in Section 5.3.4. A graphical representation of the modified simulator is presented in Figure 6.3.

Simulation is an effective technique if it allows a significant number of runs in a reasonable time frame, allowing to assess the system in diverse environments and with different configurations. In our simulator, a simulation run must encounter multiple occurrences of the rare scenarios, that we need to verify to give probabilistic guarantees of transmission high-reliability. This is not true if the fault distribution follows exactly the Poisson distribution with the intensity derived from the real BER. Just to give an example, consider a system with LSW equal to 2.5ms, a bit rate of 1000 kbps, working in an Aggressive environment. One of the rare scenarios that we need to verify is one that presents four errors in one cycle, but this scenario, in the system just described, has a probability equal to $7.4 \cdot 10^{-15}$, meaning that in average this scenario would occur each 10.7 thousand years. To have confidence in the results, we must have, for instance, one thousand occurrences of this fault pattern per

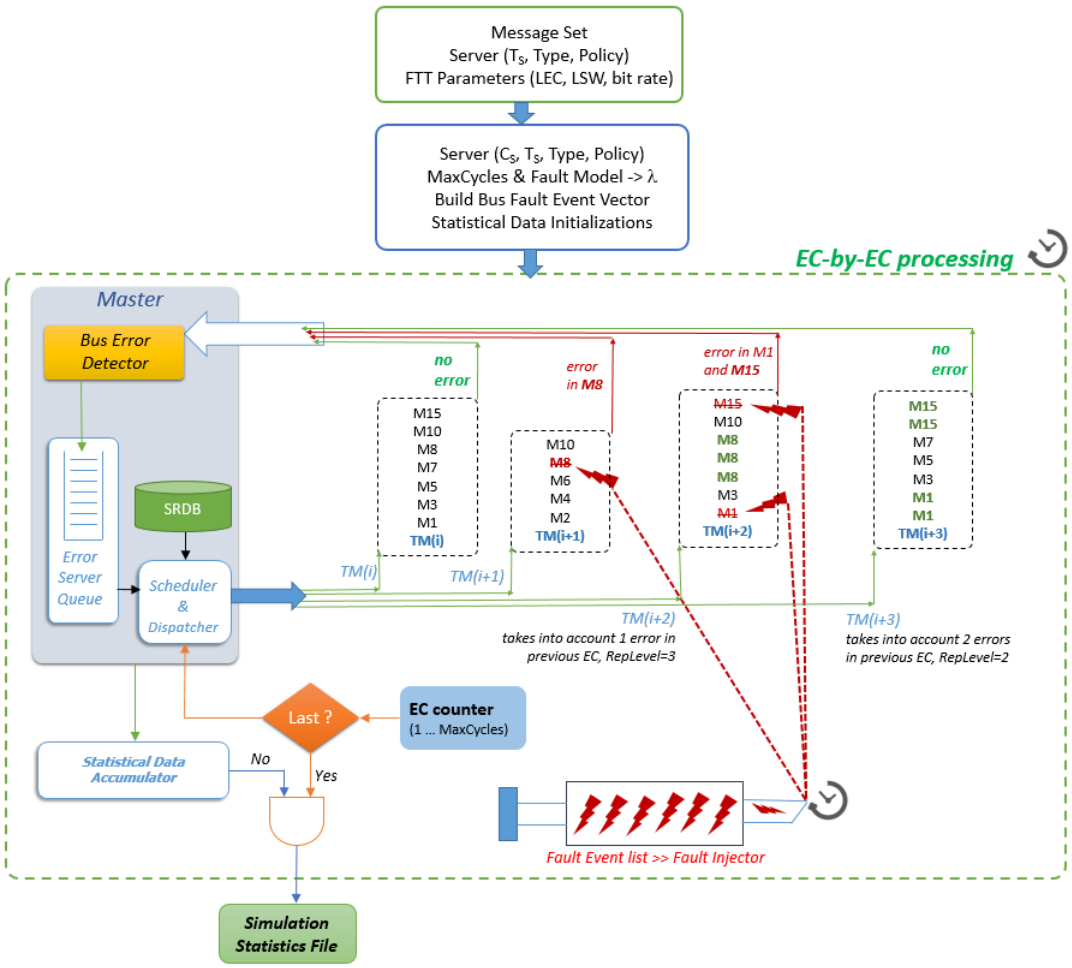

Figure 6.3: Simulator in MatLab - Multiple Error and Replica version.

simulation run, meaning that it needs a total simulation time in the order of ten million years. Even if our simulator has a speed-up factor equal to 1000 (ratio between simulation time and real time), we still end up with more than ten thousand years of simulation to test one particular scenario (ambient and specific characteristics). This is clearly not practical, so a way was devised to simulate these rare scenarios in a reasonable time, which is described in the following paragraphs and is depicted graphically in Figure 6.4.

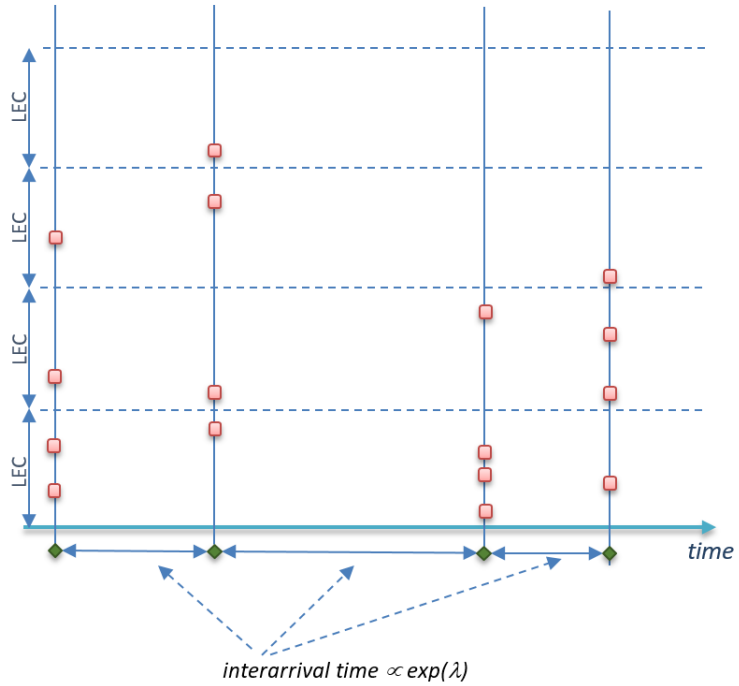

Figure 6.4: Error generation with compound fault model.

It starts by generating a list of fault time instants, according to a Poisson distribution with intensity equal to λ, meaning that faults are exponentially separated - these are the green diamonds. Following this, the fault list is transformed in the correspondent EC number where the fault would occur. Then, by using a compiled vector of the fault scenarios (described in Section 5.3.3.3), for each element of this derived list a rare scenario is associated, that is chosen randomly (with uniform distribution), from all the rare scenarios which transforms each original fault in a sequence of faults, that happens in consecutive cycles. Lastly, to each fault is attributed a time instant inside the EC, chosen randomly with uniform distribution inside the EC - red squares in the picture.

Some caution must be taken to not introduce scenarios with a sequential number of errors beyond the limit defined by the analysis in section 5.3.3.3. Then, if the base distance between two events is less than max_cycles, that value is discarded in the base fault event list. This has to be done in order to prevent that after the introduction of the rare scenarios the number of faults be greater then the limit value specified (in one cycle or in consecutive cycles). For instance if we are considering the limit fault scenarios with 4 faults per EC and maximum of 4 faults in consecutive cycles, if we accept a base event with separation less than 4, then

we could end up, for example, with a maximum of 8 faults in two consecutive ECs, a value that is much greater then the intended maximum and the system would be tested beyond the intended limits, leading possibly to an over-designing of the system, assigning thus more resources than the ones strictly necessary to obtain the intended transmission reliability level.

So, we end-up with a global fault pattern that has the base ECs distributed according to Poisson, with superimposed worst case scenarios, just like having bursts of faults for each base event [NSS00].

So, the simulation can run in a short time to assess the recovery method, as this allows observing the impact of the rare patterns injected in random positions of the message stream. The number of found occurrences should be in sufficient large number for each scenario occurrence, getting this way confidence on the obtained results.

6.4 Assessing the Error Recovery Method with Compound Fault Model and Multiple Message Retransmission

To assess the recovery methodology presented in Section 5.3 we used the benchmarks *Updated_SAE*, *PSA* and *VEIL*, as previously.

The first benchmark used was *Updated_SAE*, an Aggressive environment was considered, with BER $= 2.6 \cdot 10^{-7}$ or equivalently a fault incidence $\lambda = 0.26$ faults per second, for a bit rate of 1000 kbps.

Prior to any simulation run, it is necessary to correctly configure all system characteristic values. Since the recovery mechanism can only act in the cycle following the error detection, it is necessary to configure the LEC with a value that is at most half the period of the fastest messages. So, for this benchmark, choosing 2.5 ms for this parameter allows enough time to retransmit the message. Concerning the Deferrable Server period and capacity, the (T_S, C_S) pair, the guidelines given in Section 5.3.2 where used, being configured with $T_S = 1500 \cdot LEC$ (round value near $T_S = \frac{1}{\lambda}$ seconds equal to 1538 ECs) and $C_S = 3 \cdot 12 \cdot C_{MAX}$.

Finally, the LSW value must be obtained, being this typically the minimum LSW that makes the system schedulable, including the occurrence of all worst case fault scenarios previewed by the considered fault model. So to compute the LSW, *Minimum_LSW* algorithm was used, that was described in Section 5.3.4. On his turn, this algorithm triggers the execution of other algorithms, that produce a set of results that are interesting to analyze and comment. For instance, the maximum errors in one cycle - max_1cycle - and the maximum consecutive cycles with single errors - max_cycles - are both four, that were obtained using Algorithm *MaxErrors*, that runs inside Algorithm *Minimum_LSW*. In these conditions, Algorithm *Calc_RepLevel* returns $RepLevel = \{3, 3, 2, 1\}$. Using the combinations for max_cycles with max_1cycle to build a list of rare fault scenarios sequences and after combining it with *RepLevel*, the corresponding interference pattern array (with only worst case/rare scenarios, faults in first cycles) is the one given in Table 6.12.

These interference patterns correspond to the server execution in response to the rare fault

Table 6.12: *The interference pattern*

Interf_Pattern = {
3, 3, 3, 3;
3, 3, 6, 0;
3, 6, 3, 0;
3, 6, 0, 0;
6, 3, 3, 0;
6, 6, 0, 0;
6, 3, 0, 0;
4, 0, 0, 0 }

scenario sequences, that were triggered by the the correspondent fault scenario, interfering with the messages in the scheduling process.

The optimization process leads to a minimum LSW value equal to 55.1% of the LEC to schedule the message set, including the error server, and the C_S value being adjusted to to $12 \cdot 3 \cdot C_{MAX}$. Table 6.13 present the data generated by the LSW optimization process, where the diverse interference patterns, the response time for each pattern, with Direct and Indirect interference and WCRT for each message are presented. The 0 errors column is included for reference, presenting the response times in a fault free environment.

Comparing column WCRT with the Deadline one, we can observe that all messages present individual WCRT lower than the corresponding deadline, considering all error interference patterns, guaranteeing real-time behaviour, which is a requirement for the system.

As expected, the results show a degradation of the WCRT in almost all messages, when compared with the no errors scenario. The first eight messages have a penalty of one EC, since they fit in the EC, in which they become ready, even when the server executes. Notice that this is strictly necessary, as these messages must be sent in the EC where they become ready, if not there is no slack for retransmission in case of error hit, as they possess deadline equal to 2 ECs. The messages with lower priorities, e.g., 30 to 36, suffer a stronger response time penalty as they are affected by a higher interference level. Table 6.13 also confirms that Direct Interference normally dominates Indirect Interference, as WCRT's with Direct Interference are greater or equal for all messages.

This last aspect is not always verified. To show this, we re-run the optimization process but now using *RepLevel* equal to { 4, 3, 2, 1 }. It was observed that there were messages in which Indirect Interference dominated. The correspondent table is presented in Table B.17 in Appendix B, where messages with ID 31, 32 and 33 obtain the WCRT with Indirect interference scenarios. This confirms the need to always compute both kinds of interference to determine a safe upper bound to the WCRT.

The design process was repeated for the two other benchmarks and equivalent data results are presented in Appendix B, in tables B.18 and B.19. This algorithm, implemented in MatLab and using a Pentium i7-2670QM computer with 6 GB of memory, took less than 1

Table 6.13: WCRT of all messages in *Updated_SAE* benchmark, for each rare scenario of Indirect and Direct Interference.

	Indirect Interference								Direct Interference								
msg	**3-3-3-3**	**3-3-6-0**	**3-6-3-0**	**3-6-0-0**	**6-3-3-0**	**6-6-0-0**	**6-3-0-0**	**4-0-0-0**	**3-3-3-0**	**3-6-0-0**	**6-3-0-0**	**6-0-0-0**	**IND**	**DIR**	**WCRT**	**no error**	**Deadline**
1	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
2	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
3	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
4	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
5	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
6	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
7	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
8	1	1	1	1	1	2	1	1	2	2	2	2	2	2	2	1	2
9	1	1	1	1	2	2	2	1	2	2	3	3	2	3	3	1	3
10	1	1	1	1	2	2	2	1	2	2	3	3	2	3	3	1	3
11	1	1	1	1	2	2	2	1	2	2	3	3	2	3	3	1	3
12	1	1	1	1	2	2	2	2	2	2	3	3	2	3	3	1	3
13	1	1	2	2	2	2	2	2	2	3	3	3	2	3	3	1	3
14	2	2	2	2	2	2	2	2	3	3	3	3	2	3	3	1	3
15	2	2	2	2	2	2	2	2	3	3	3	3	2	3	3	1	3
16	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	1	3
17	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	1	4
18	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	2	4
19	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	2	4
20	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	4
21	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	4
22	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	4
23	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	5
24	2	3	3	3	3	3	3	2	3	4	4	3	3	4	4	2	5
25	3	3	3	3	3	3	3	2	4	4	4	4	3	4	4	2	5
26	3	4	4	3	4	4	3	2	4	4	4	4	4	4	4	2	5
27	3	4	4	3	4	4	3	2	4	4	4	4	4	4	4	2	5
28	3	4	4	3	4	4	3	3	4	4	4	4	4	4	4	2	5
29	4	4	4	4	4	4	3	3	4	4	4	4	4	4	4	2	5
30	4	4	4	4	4	4	4	3	5	5	5	4	4	5	5	2	20
31	4	4	4	4	4	4	4	3	5	5	5	4	4	5	5	2	40
32	4	4	4	4	4	4	4	3	5	5	5	4	4	5	5	2	40
33	4	4	4	4	4	4	4	3	5	5	5	4	4	5	5	2	40
34	5	5	5	4	5	5	4	3	5	5	5	4	5	5	5	3	400
35	5	5	5	4	5	5	4	3	5	5	5	5	5	5	5	3	400
36	5	5	5	4	5	5	4	3	5	5	5	5	5	5	5	3	400

second to complete, with 0.1% precision as stopping criterion for the LSW computation.

6.4.1 Assessing by Simulation the Design Method

After correctly configuring the system, we simulated *Updated_SAE*, for 10 million cycles of operation, corresponding to 7 hours (each simulation run) of system operation. The simulation runs were repeated 20 times (by simply using different seeds in the pseudo random generator) for all three benchmarks without observing any missed deadlines, thus with all errors recovered in time, as expected. These results indicate a correct functioning of the recovery method and a good choice of parameters.

To assess the tightness of the design process, we compared the WCRT generated analytically with the maximum observed response-time in all the simulations. Table 6.14 presents this comparison for the *Updated_SAE* message set, showing that the analytic WCRTs are tight for the messages with higher priority with the potential pessimism growing for the lower priority messages. Nevertheless, note that the values obtained from simulation are not guaranteed to be the absolute maxima, due to the limited simulation time, thus some of the reported differences between computed and observed values may be smaller. Identical results were observed in the other benchmarks, being presented the corresponding tables B.18 and B.19 of Appendix B.

Table 6.14: Comparing analytic WCRT with the one observed in simulations for the *Updated_SAE* message set with LSW = 55.1% of LEC, considering an Aggressive environment

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
Design	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3
Simulation	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3
msg	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
Design	3	4	4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5
Simulation	3	3	4	3	3	3	3	4	3	4	4	4	4	4	4	4	4	4

Another measure of the efficiency of our design method is the tightness of the minimum LSW needed to schedule the message sets. Thus, we compared the analytic value obtained from our design approach with the minimum value that we could obtain in simulation, reducing the LSW just until deadline misses started to occur. Table 6.15 presents these values for the three benchmarks, showing that the analytic minimum value for LSW was roughly between 12% and 16% larger than the one obtained in simulation. Again, note that these differences are not upper bounds to the real differences since there is no guarantee that the simulation captured all of the actual worst-case situations. Table 6.15 also shows the minimum LSW needed to schedule the messages sets without errors, showing the impact of error recovery. This impact is particularly large in an Aggressive environment, as considered here, to guarantee timely error recovery. In fact, the minimum LSW that is generated by our approach to account for the Aggressive error scenario is 235%, 208%, and 45% larger than the minimum LSW needed to schedule the corresponding message sets without errors,

respectively, *VEIL*, *PSA*, and *Updated_SAE*. We can also observe that the relative impact of the error recovery mechanism decreases when the message set bandwidth utilization grows. This last detail should be expected as for the same interference patterns, the time used locally for error recovery is proportionally larger compared with the configured LSW in an error free environment. For example, the VEIL benchmark can be configured with an LSW of 7.1%, equivalent to 355 bits per EC when an error free environment is considered. In an Aggressive environment the LSW must be enlarged due to errors and server execution, that can corresponds to sending 6 messages in 2 consecutive cycles. With $C_{MAX} = 135$ bits, it demands capacity to schedule a total of 810 bits (in 2 ECs), which is a bigger value than the base value with no errors.

Table 6.15: Minimum LSW configuration value by design and simulation in Agressive Environment

Message set	***VEIL***	***PSA***	***Updated_SAE***
LEC (ms)	5	5	2.5
RepLevel	3-2-2-1	3-3-2-1	3-3-2-1
Bandwidth utilization (@1Mbit/s)	4.40%	9.10%	27.90%
Error server bandwidth(configuration)	0.11%	0.11%	0.11%
LSW/LEC without errors (by design)	7.10%	11.90%	37.90%
LSW/LEC with errors+server (by design)	23.80%	28.00%	55.10%
LSW/LEC with errors+server (simulation)	21.10%	24.80%	48.40%
Pessimism (design over simulation)	12.80%	12.70%	13.90%

Nevertheless, we must stress that, thanks to the dynamic scheduling feature, this extra bandwidth configured is only used when errors do occur. So, the bandwidth effectively used by the recovery mechanism is not determined by the configured LSW value, but instead by the average features of the fault model and corresponds to the error server bandwidth, which in this case is less than 0.11% of the available bandwidth (Table 6.15) e.g., the server for the *Updated_SAE* message set has $T_S = 1/\lambda$ and $C_S = 12 \cdot 3 \cdot C_{MAX}$. In a real system, the server configuration bandwidth is seldom used, because situations where all the server capacity is consumed or the maximum number of errors occurs in a server period are rare, having in average one of these situations in every 100 thousand years of system functioning, for this benchmark and with the configuration parameters derived earlier.

6.4.2 Comparison with other Methods

The work in [Fer05] presented another error recovery method for FTT-CAN, based on the native CAN automatic retransmission of messages affected by errors - named *Automatic Retransmission*. This mechanism reserves extra time in every SW for the recovery of eventual errors, where the extra time is left unused in the absence of errors, and is thus less efficient than our proposal - *Controlled Retransmission*, as shown in Table 6.16. For instance, when considering the fault model presented in this paper and an Aggressive environment, the

Table 6.16: Comparison of minimum LSW and BW requirement with different design methods

	Message set	***VEIL***	***PSA***	***Updated_SAE***
Minimum LSW	*Controlled Retransmission*	23.8%	28.0%	55.1%
	Automatic Retransmission	19.8%	24.5%	60.0%
	Static TT	22.3%	41.4%	**X**
Configuration BW	*Controlled Retransmission*	0.105%	0.105%	0.108%
	Automatic Retransmission	12.6%	12.6%	25.2%
	Static TT	15.1%	29.5%	**X**

Updated_SAE benchmark message set with the *Automatic Retransmission* mechanism would require space for four retransmissions in every EC. This represents a constant bandwidth of 22.1% (or 12.6% for the other message sets, which have LEC equal to 5 ms). Conversely, our *Controlled Retransmission* mechanism consumes a maximum bandwidth equal to the one assigned to the error server (thirty six maximum messages ($3 \cdot 12 \cdot C_{MAX}$) in the server period), configured, which is only 0.108% in this case. The average bandwidth consumption is only 1/12 of this value.

The method presented by Tanasa et al. [TBEP10] implies the use of a fixed number of replicas per message period, and is applied in a TDMA fashion - *Static TT* in Table 6.16. The number of replicas is determined applying Equation (4.6), for a defined mission time and global reliability goal. For the three referred benchmarks and with a global reliability equal to 10^{-9}, we need to send always four copies of each message, which corresponds to more than 300% overhead, accounting also with error signaling. The three extra messages sent and their transmission time are always wasted when there are no errors, which is the most common scenario, for the considered BER, e.g. in average there is one error each 1538 ECs, for *Updated_SAE* benchmark. This happens because the scheduling is static and needs to cope with every error, without knowing when and where they will occur. Thus, enough message copies are sent to get a probabilistic assurance that the reliability goal is attained. This is clearly in opposition to *Controlled Retransmission* proposal where the bandwidth is only used when needed, i.e., when errors do occur. Observing Table 6.16, we note that this method requires a minimum LSW similar to our method for the lightest set (*VEIL*). However, for a set with medium bandwidth utilization (*PSA*), our method already requires a minimum LSW that is 32.4% less than the one required by Tanasa's method. For the set with higher utilization (*Updated_SAE*) Tanasa's method cannot even generate a schedulable solution, which was expected as this message set has an utilization greater than 25% and we need to send a total of 4 copies per message to attain the reliability goal. Comparing the bandwidth overhead of 15.1% and 29.5% for the two other sets (that corresponds always to more than 300% increase when referenced to each message set utilization bandwidth), with the reserved bandwidth of less than 0.11% used by our method is even more revealing. Further, using the results available in the statistic files of the performed simulations just presented, the bandwidth that was used in the recovery process was also calculated. The

average value found was very low, being 0.00047%, 0.00093%, and 0.0025% of the available bandwidth for the *VEIL*, *PSA*, and *Updated_SAE* benchmarks, respectively. These values are much lower than the bandwidth configuration server values of 0.105% and 0.108%, as the full server capacity was not used, most of the time.

Looking at other results, we observe that the message average response time in *Controlled Retransmission* and *Automatic Retransmission*, when configured with the same *LSW*, are different. For instance consider the *Updated_SAE* benchmark with a $LSW = 60\%$, that is the minimum configuration value for these two methods in Agressive environment (see Table 6.16). The average response time obtained by simulation are the ones presented in Table 6.17, which was obtained with 10 simulation runs, each one with 10 million cycles. The first observation is that for messages with higher priority (ID from 1 to 13) they present an average response time equal to one, as these messages are scheduled without interference, as they fit in the EC where they become ready, even when the server executes. The very small difference between the two methods is due to the recovered messages, that in the *Controlled Retransmission* method always suffer one EC delay. All other messages are slower in *Automatic Retransmission*, being this increasing difference justified by the fact that in average the scheduling process uses all the 60% of the LSW in the *Controlled Retransmission* where in the *Automatic Retransmission* only 37.9% can be fully used, as there is a reserve in every cycle with time length necessary to transmit the maximum number of messages found with error (plus the error signalling messages), which in this case is four, being a total of $4 \cdot (115 + 23)$ bits or 22.1% of the LEC.

Table 6.17: Comparing average response time of for the *Updated_SAE* benchmark with LSW=60.0% of LEC, considering an Aggressive environment - *Controlled Retransmission* vs *Automatic Retransmission*

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Automatic	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Controlled	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0001	1.0001	1.0002
msg	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
Automatic	1.0000	1.0500	1.5000	1.5000	1.3333	1.3333	1.3333	1.3333	1.4667	2.0000	1.3333	1.3333
Controlled	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0003	1.0002	1.3336	1.3335	1.0835	1.0835
msg	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
Automatic	1.5000	1.6667	1.8333	2.0000	2.0000	3.3333	3.3333	4.0000	4.0000	4.6667	4.6667	4.6667
Controlled	1.0835	1.0835	1.0836	1.3336	1.3336	2.0002	2.0003	2.0002	2.0002	2.0002	2.0002	2.0002

One disadvantage of the method proposed in this thesis is that it detects errors in the end of the EC and thus the retransmissions can occur at best in the following EC, which leads to an error recovery latency of one EC. This has clearly an impact on the WCRT, as for instance no message can present a WCRT lower than 2. Conversely, the other two methods allow error recovery in the EC in which they occur, but this does not mean that for the same parameter configuration, namely the same *LSW*, it has always lower WCRT. Let's then compare the WCRT for the *Updated_SAE* benchmark, with the same conditions

as previously, that are presented in Table 6.18. As can be seen there, the messages with higher priority have lower WCRT (message with ID 1 to 15 and 18 to 24), other present the same value and for the 7 lower priority messages the situation reverts, despite not having a cycle delay in error recovery. Just a note on messages with ID 28 and 29, that most of the time present WCRT equal to 3, being the value 4 present in the table due to only one occurrence each, where the recovery message was hit by an additional error, so a delay equal to 2 cycles in the recovery process happened.

Results for the other benchmarks are presented in Appendix B, Section B.3.1.

Table 6.18: Comparing worst case response time of for the *Updated_SAE* benchmark with LSW = 60.0% of LEC, considering an Aggressive environment - *Controlled Retransmission* vs *Automatic Retransmission*

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
Automatic	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2
Controlled	2	2	2	2	2	2	2	2	2	3	3	2	3	3	3	2	2	3
msg	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
Automatic	2	2	2	2	2	2	3	3	3	3	3	4	4	4	4	6	6	6
Controlled	3	3	3	3	3	3	3	3	3	4	4	3	3	3	3	3	3	3

6.4.3 LSW Optimization and BW Required by the Error Recovery Mechanism

The LSW optimization was first applied to the three message sets, being the results shown in Table 6.19. This table presents the configuration characteristics, *RepLevel* and minimum *LSW* of the 3 benchmarks in the 3 considered ambients, observing the increasing values of max_1cycle, max_cycles and *RepLevel* as the foreseen error incidence grows, for the same reliability goal. Another noticeable observation is the poor performance of the *StaticTT* method, as it needs a bigger "equivalent" LSW than the other methods, in almost all situations.

6.4.3.1 LSW Optimization with Random Sets

Multiple runs of the LSW optimization module were carried out to assess the performance of the three methods referred in the previous section, concerning the minimum LSW value required to attain the desired error responsivity, as a function of the message set utilization. The bandwidth utilization range varied between 5% and 70%, in steps of 1%. Message payload length varied from 1 to 8 bytes and message periods from 2 to 15 ECs, with implicit deadline (i.e., equal to period), both with uniform distribution. LEC was set to 2.5 ms and one thousand message sets were generated for each utilization value.

The results presented in Figure 6.5, represent the requirement for minimum LSW obtained (average value). There, *Static TT* shows a better performance than *Automatic Retransmis-*

Table 6.19: Main characteristics of the 3 benchmarks, in different environments

Updated_SAE						
Ambient	*max_1cycle*	*max_cycles*	*RepLevel*	*Controlled LSW(%)*	*Automatic LSW (%)*	*Static TT LSW (%)*
Error free				37.9	37.9	37.9
Benign	2	2	2-1	42.8	49.0	73.0
Normal	2	3	2-1	44.9	49.9	**X**
Agressive	4	4	3-3-2-1	55.1	60.0	**X**
PSA						
Ambient	*max_1cycle*	*max_cycles*	*RepLevel*	*Controlled LSW(%)*	*Automatic LSW (%)*	*Static TT LSW (%)*
Error free				11.9	11.9	11.9
Benign	2	2	1-1	14.0	18.2	22.0
Normal	2	2	2-1	16.7	18.2	31.6
Aggressive	4	4	3-3-2-1	28.0	24.5	41.3
VEIL						
Ambient	*max_1cycle*	*max_cycles*	*RepLevel*	*Controlled LSW(%)*	*Automatic LSW (%)*	*Static TT LSW (%)*
Error free				7.1	7.1	7.1
Benign	2	2	1-1	10.3	13.4	12.5
Normal	2	2	2-1	13.0	13.4	17.3
Aggressive	4	4	3-2-2-1	23.8	19.8	22.2

sion (smaller minimum required LSW) but only for small utilization values (less than 8%). The *Controlled Retransmission* method always shows better performance than all the other ones. When compared to *Automatic Retransmission*, it is slightly better for sets with utilization lower than 15% and the performance improvement grows for larger utilizations, requiring up to about 16% less time in the LSW parameter. The picture also shows a very important aspect, as *Controlled Retransmission* method allows attaining higher utilization levels than the competing ones, for the same reliability goal, allowing up to 70% utilization against 20% and 55% from *Static TT* and *Automatic Retransmission*, respectively.

Figure 6.6 also illustrates the average LSW requirement, for the *Controlled Retransmission* method, but now including the greatest and the smallest LSW value, for two different ambients: Normal and Aggressive. As the scenarios for Normal ambient have error and recovery scenarios less dense (lower values of *max_cycles* and *max_1cycle*), that's the reason why the difference between maximum and minimum value are smaller, as expected.

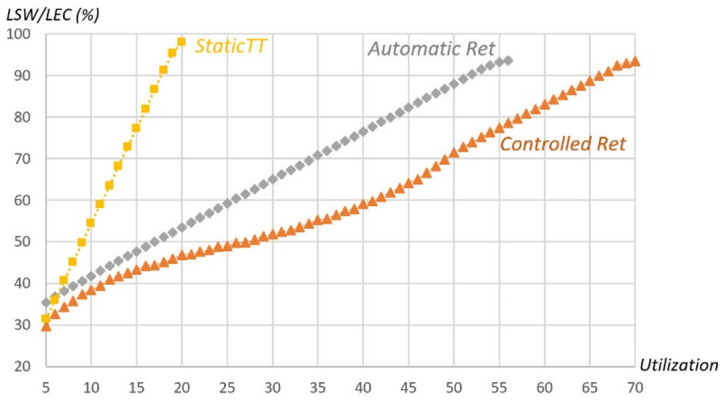

Figure 6.5: Average requirement for minimum LSW vs message set bandwidth utilization, in Aggressive environment.

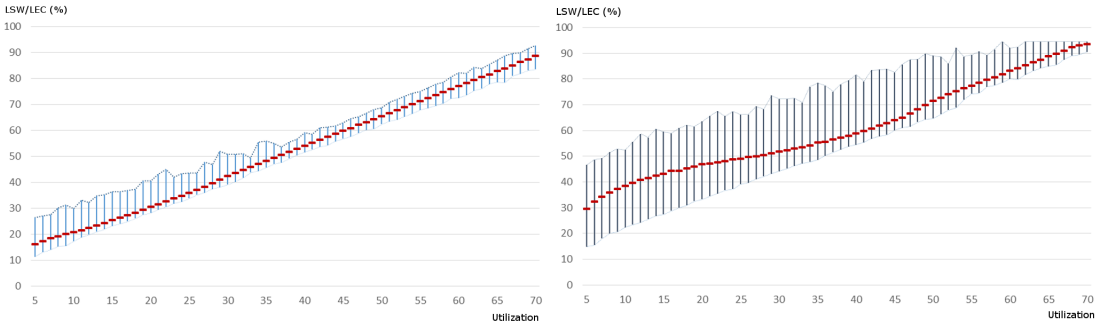

Figure 6.6: LSW required in Normal (left) and Aggressive Ambient (right) for *Controlled Retransmisison*: minimum, average and maximum values.

Figure 6.7 presents the average value for the minimum LSW configuration value required of each method, for each ambient considered. The green line represents this value for the no-error scenario.

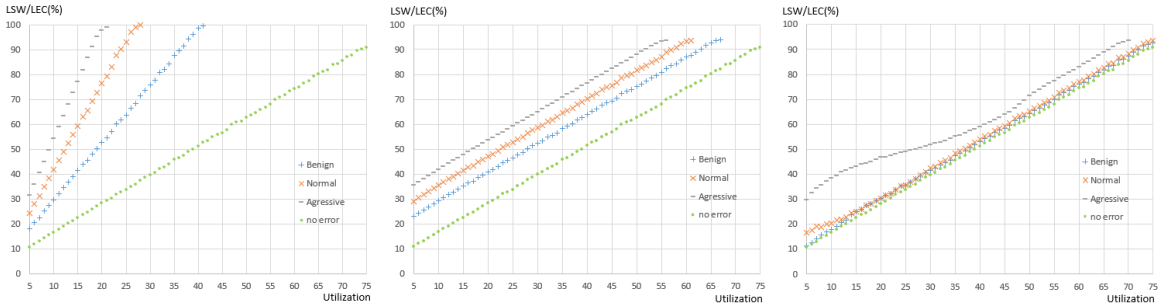

Figure 6.7: Minimum required LSW value (average) used by each method in different ambients. Methods from left to right: *Static TT*, *Controlled Retransmission* and *Automatic Retransmission*.

In Figure 6.8 the same information with different grouping is presented for the minimum LSW configuration value, where each sub-figure compare the three methods (average value) in each ambient.

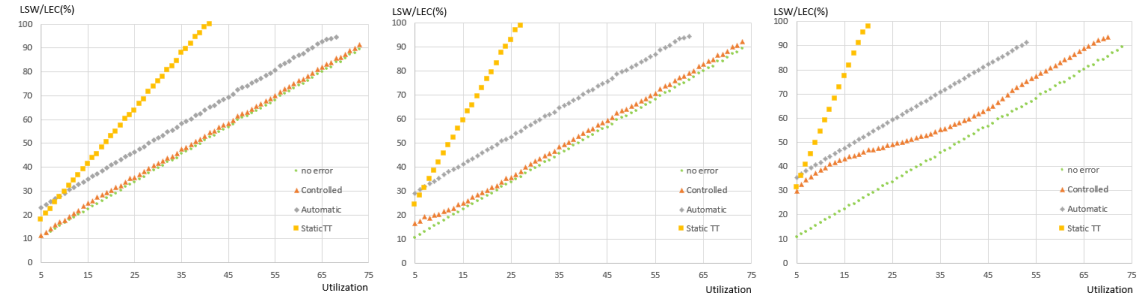

Figure 6.8: Minimum required LSW value (average): method comparison by ambient. From left to right Benign, Normal and Aggressive

Finally, Figure 6.9 represents the bandwidth required by the different error recovery mechanisms as a function of the message set utilization, being obtained with random message sets using LEC equal to 2.5 ms, an Aggressive environment, and the other parameters as in the previous experiences. In *StaticTT*, the required bandwidth is proportional to the number of copies needed for each message, also including error signaling, being always more than 300% of the corresponding message utilization bandwidth, as a total of four copies per message are required to attain the desired reliability level. Thus, more than 70% of the total bandwidth is allocated to the recovery mechanism, even for message sets with a utilization of only 20%. *Automatic Retransmission* reserves slack time in each EC to recover the maximum number of errors considered (equal to *max_1cycle*), which in the studied cases is 3 or 4, corresponding to 19.0% or 25.3% of the available bandwidth. Therefore, the overhead is essentially constant, having a step in the utilization transition from 19% to 25%, corresponding to the situation in which the errors that need to be handled changes from 3 to 4 per EC.

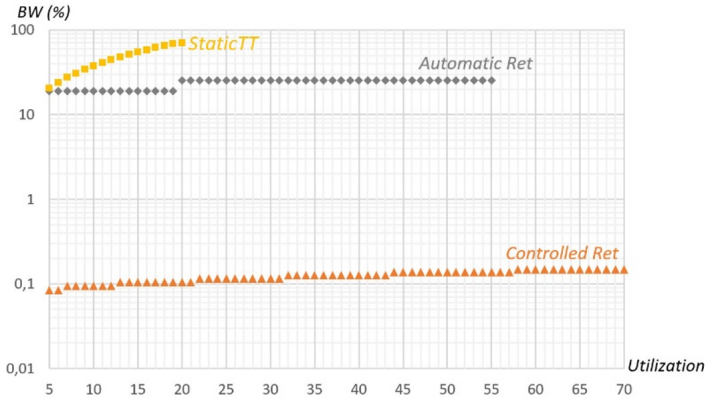

Figure 6.9: Bandwidth required by each method, with LEC=2.5ms and λ=0.26.

As for the *Controlled Retransmission* mechanism, the reserved bandwidth is the one of the error server. The server configuration depends on the fault arrival rate and maximum *RepLevel* configuration value, and setting the Server period T_S equal to $1/\lambda$, the necessary capacity varies between $8 \cdot 3 \cdot C_{MAX}$ and $14 \cdot 3 \cdot C_{MAX}$, which is only 0.084% and 0.147% of the 1 Mbps available bandwidth. This figure clearly shows the superiority of our method, since its required bandwidth is at least two orders of magnitude inferior to the other methods.

Again, the value presented in the graph for the *Controlled Retransmission* method corresponds to the maximum value of bandwidth that can be used in one server period, being the average used bandwidth (considering several cycles) much less, as it only uses it when errors occur. Otherwise, in the two other methods, the average bandwidth used is fixed, being always equal to the configured value, as it is reserved for error recovery and cannot be used for other purposes.

6.5 Issues in Master Implementation

In this section we analyze and test some critical implementation issues in the Master node, namely the modifications necessary to implement the method proposed in this thesis.

6.5.1 First experiments

We have used a reduced network, with just 3 nodes as represented in Figure 6.10. A 125 kbps bit rate was used and one of the nodes was simply counting messages and displaying it on a LCD with 2 character lines, for visual feedback of the experience outcome. The nodes were implemented with a 18F2680 microcontroller from the PIC18 Microchip family [Inc07], which has an 8 bit architecture, 10 MHz clock, 64 Kbytes of EEPROM, 3Kbytes of RAM and integrated CAN controller. This microcontroller was used due to low cost, but also because an FTT-CAN source code implementation was available in [FTT18], that was then modified by introducing a module that implemented the proposed methodology for error recovery. Despite simple, this experimental setup allows us to measure several timing characteristics and verify some requirements to guarantee implementability of the recovery method.

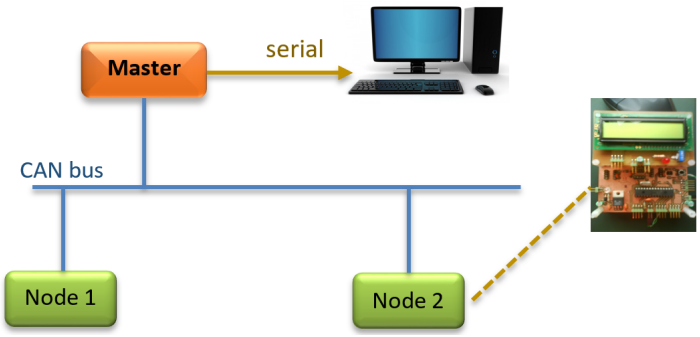

Figure 6.10: Small network architecture for first experiments.

The first experiment carried out aimed at assessing the overhead incurred by the Master in detecting messages in the bus. The experimental results depicted in Figure 6.11 present in the first line the 4 CAN messages and the middle line the dispatcher running. The bottom line corresponds to the processing time of the message reception, that includes adding the message ID to the vector of received messages. A Tektronix TDS2024 oscilloscope was used, and the observed duration for the reception processing was equal to 50 μs (worst case), which is acceptable for baud rates of 500 kbps or lower. For higher baud rates the reception processing time approaches the transmission time of the shortest data messages (55μs for a zero data byte message at 1000 kbps) and thus error detection may fail, in cases where more than one of these messages is sent in sequence. Otherwise, the message signalling process would work without problems.

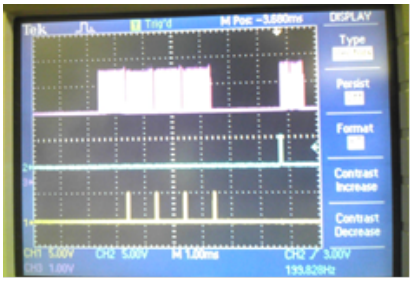

Figure 6.11: Observing transmission and detection timings in oscilloscope.

6.5.2 Optimizing the Scheduler to Obtain Minimum Latency

The initial modification introduced maintained the scheduler positioning in the EC, running during the Asynchronous Window. Because of this, the rescheduling of failed messages presented a latency equal to 2 ECs, when maximum priority is used. This is clearly disadvantageous as it will delay the message recovery by 2 ECs, which is not desirable as we want to make the recovery as fast as possible. Of course, if we still want to use it this way, then the LEC value must be chosen so that it is less or equal than 1/3 of the fastest message, or else the recovery may fail.

In order to reduce the latency to one EC, which is the minimum possible value with this

method, one possibility is to re-write partially the Master's code in order to run the scheduler module immediately after the end of the SW, just after the last received message has been processed. These two versions are presented in Figure 6.12. In **a)** minimum code change, implying two ECs delay in recovery, as explained, and in **b)** the modified version that was just described and is the one considered in this thesis.

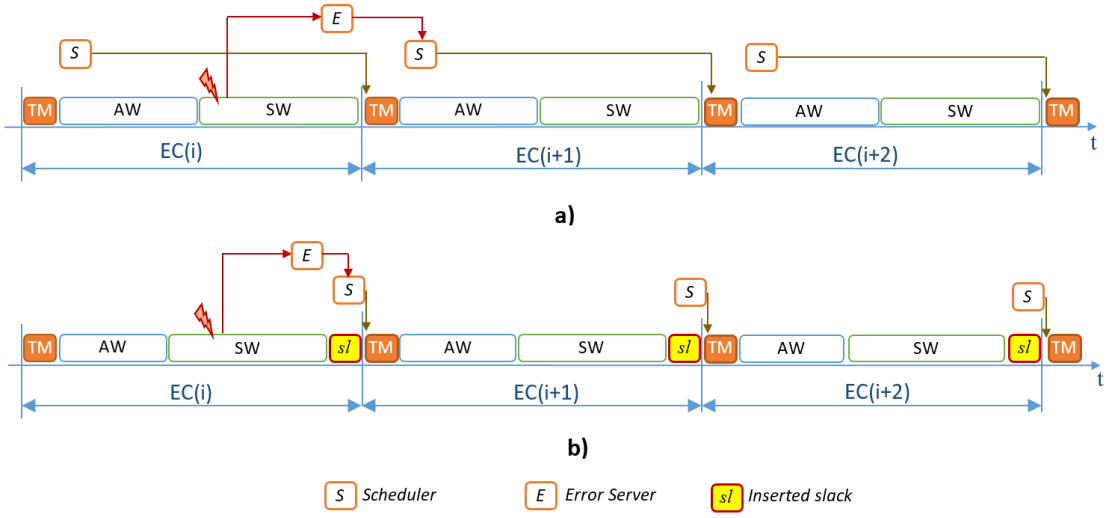

Figure 6.12: Scheduler timing inside EC.

Then, experimental tests have been carried out to evaluate the scheduler and dispatcher worst-case execution times. The obtained values were a 600 μs for the scheduler and 100 μs for the dispatcher. The results here obtained are similar to the ones presented in [Sil10], for a mobile robot of the middle size soccer league with a FTT-CAN network with two Masters (main plus backup), 6 slave nodes, 9 messages with periods between 20 ms and 1 second and with and EC duration of 5 ms. The tests presented there showed a scheduler worst-case execution time of 628 μs.

Considering a network bit rate of 1Mbps, and considering the summation of the three execution components, with a total of 750 μs (accounts for last message reception, scheduler and dispatcher execution), which is the slack time necessary at the end of the SW. This inserted slack corresponds to an impact equivalent to almost six messages with 8 bytes of payload per EC, which is significant penalty, specially for small LSW's (e.g., 5 ms). For lower baud rates, however, the penalty can be tolerable. Tables 6.20 and 6.21 exemplify the penalties involved, assuming the current implementation based in an 8 bit microcontroller and also extrapolating the values to a possible code porting to a 32 bit microcontroller, from the same vendor, with even small retail price, that can run 12 times faster [Inc17].

The characteristics of these microcontrollers are displayed in Table 6.22, that shows that there is now devices with higher speed, more Flash and RAM memory, more peripheral including independent CAN controllers, with a small increase in price (PIC32 family) or even a lower price. For instance, the ATSAME51J18A (ATMEL family), with ARM4 architecture, 120 MHz clock, 2 integrated CAN FD controllers and 4 times the Flash memory, has a cost

Table 6.20: Reduction in number of possible scheduled messages function of the bit rate (considering 8 bytes of payload and maximum bit-stuffing).

bit rate (Kbps)	**8 bit μC (10 MHz)** (number of messages)	**32 bit μC (120 MHz)** (number of messages)
125	0.69	0.06
250	1.39	0.12
500	2.78	0.23
1000	5.56	0.46

Table 6.21: Penalty for displacing scheduler in time (Reduction in available time for message transmission vs EC Length)

EC Length (ms)	**8 bit μC (10 MHz)** PIC18F2680	**32 bit μC (120 MHz)** ATSAME51J18A
2.5	30.0%	2.50%
5	15.0%	1.25%
10	7.5%	0.63%
20	3.8%	0.31%
100	0.8%	0.06%

even almost 30% lower than the 8 bit counterpart. This also opens the possibility to use the new CAN-FD standard, with minor code modifications, as this family of microcontrollers already possesses two internal CAN-FD controllers. Moreover, even if the price is higher it only affects one module (the Master node). So, it seems that there is no reason to not use these new devices in a near future.

Table 6.22: Characteristics comparison between microcontrollers - 8 bit vs 32 bit

μcontroller	PIC18F2680	PIC32MX775F512H	ATSAME51J18A
Architecture (bits)	8	32	32
CAN Controllers	1	2	2
Type	CAN 2.0B	CAN 2.0B	CAN FD
Clock (MHz)	10	80	120
Flash (KB)	64	512	256
Price (>5K)	4.10€	5.16€	2.97€
Cost* (comp PIC18F)	-	+26%	**-28%**

*prices obtained in MicrochipDirect website, on 15.april.2018.

From the analysis of the tables it can be concluded that if the scheduler is placed at the end of the synchronous window, in order to reduce the scheduling latency from two to one EC, there is clearly a bandwidth penalty. For small EC lengths and 8 bit microcontrollers such penalty in unbearable (up to 30% of the LEC). However, using a 32 bit microcontroller

the penalty becomes small, or even negligible, for any EC length and baud rate.

An implementation of the scheduler was done for a PIC32 microcontroller, that was compiled with MikroC for PIC32 compiler and run on a EasyPIC FUSION v7 MCUcard with PIC32MX460F512L and 80 MHz clock. By monitoring the scheduler, including the rescheduling process, a execution time less than 61μs for a message set with a total of 32 messages was obtained. In Figure 6.13 the pink line represents the time taken by the scheduling process, accumulating diverse situations in the number and priority of messages being recovered.

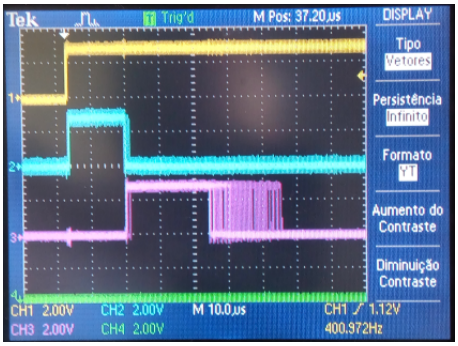

Figure 6.13: Scheduler running in PIC32 μcontroller with 80 MHz clock.

This confirms the extrapolation referenced before, concluding that this modification is feasible using a fast microcontroller, which is available in mass market and with low cost tag price.

6.6 Summary

This chapter presented an assessment of the *Controlled Retransmission* method, performing a simulation study and comparing it against other methods. It started by presenting the first version of the simulator, which uses single replica when recovering errors and afterwards three benchmarks that will be used in the simulation study. Using this simulator version a correct functioning of the proposed method was observed, validating also the error server type initially chosen - the Deferrable Server. For the same goals, the Polling Server presents much higher bandwidth use and the Sporadic Server shows a much greater execution overhead. In what concerns the priority choosing for the server, the assignment of maximum priority to the retransmissisons, allows the fastest response with minimal execution overhead. However, the single replica retransmission limits the achievable reliability, as scenarios with errors in consecutive cycles makes probable that messages with short deadlines may loose their deadlines, which was observed. It is followed by the presentation of the second simulator version, with multiple replica retransmission, where the number of replicas is chosen according to the fault scenario, to attain a global goal on transmission reliability. The assessment starts by simulating the three benchmarks, showing a correct behaviour and very low average bandwidth use. The proposed method, was then compared with other methods, *Automatic Retransmission* and *Classic TT*, presenting globally a lower bandwidth in the allocated resources and much

lower in execution time, as the proposed method uses resources only when errors occur and the other two, reserve the resources for worst case scenarios. Nevertheless, the *Controlled Retransmission* has a recovery latency of 2 cycles and the other ones have in-cycle recovery. Finally, a small study was performed to determine the necessary characteristics required by the Master node to implement the proposed method, namely in terms of used resources to detect and reschedule the failed message transmissions in one cycle.

Chapter 7

Generic Model and Applicability to TT Protocols

The method presented and assessed in the previous chapters was based on the FTT-CAN protocol. Despite being tailored for this particular protocol, the fundamental characteristics of the recovery method can be abstracted and generalized to other TT based protocols. So, this chapter firstly describes the generic model and after this its applicability to TTCAN and to FlexRay protocols.

7.1 Generic Model

The proposed method has a module inside the Master that listens to all messages sent and builds a list of failed message transmissions, that are stored in an error queue. When the scheduler is executed, the schedule for the next EC is obtained, taking into account the messages that are stored in the error queue, being the recovery dynamically controlled together with all the other regular messages. The dissemination of the message transmission triggering is done by the TM message, having a Master/multi-slave control, on a EC-by-EC basis. Figure 5.11, presents and overview of the error recovery process.

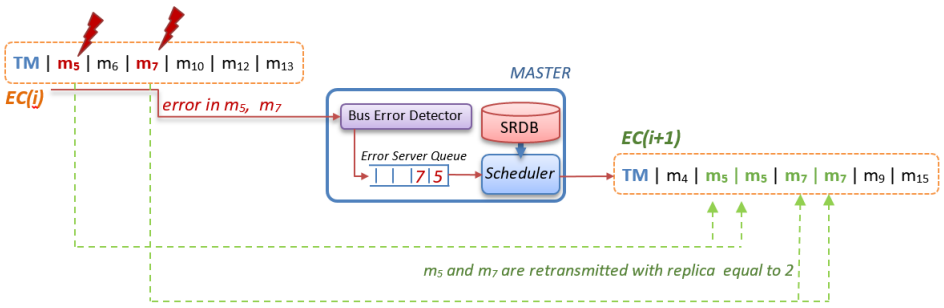

Figure 7.1: Proposed method for error recovery in FTT-CAN scope.

Some of the above procedures are specific to FTT-CAN. In fact, the fundamental operating principles of the protocol, which translates to requirements that must be satisfied by

any target protocol, are:

(i) on-line scheduling, permitting a dynamic control of message retransmissions;

(ii) message transmission organized in cycles, with schedule dissemination each cycle;

(iii) failed message transmission detector, requiring then a broadcast medium.

Additional resources and processing power are also necessary, namely for the error detection module execution and inclusion of queued messages in the scheduling process, that are relatively easy to find in modern microcontrollers.

In the next two sections we show that the presented generic method can be applied to other protocols. These possess some specificities and then the recovery mechanism must be adapted accordingly.

When describing the necessary adaptations it is considered that is possible to obtain *max_1cycle*, *max_cycles* and the *RepLevel* vector in a way similar to the one described in Chapter 5 for FTT-CAN, that will limit the possible error and recovery scenarios.

7.2 Error Recovery Applied to TTCAN

In TTCAN (Section 3.3.1) the schedule is obtained off-line, being fixed at system startup and cannot change during normal operation. Each message must be transmitted in a reserved time window, with fixed duration, and the message must be transmitted in single-shot mode to guarantee the strict windows timing. The sequence of windows is arranged in basic cycles, that always start with a special message, the Reference message, that sets a time reference with adequate resolution. A sequence of basic cycles, that repeats itself, forms the TTCAN Matrix.

The TTCAN protocol also includes Arbitration Windows, which use the native CAN arbitration scheme, and thus do not require a pre-defined message allocation. Thus, these windows allow scheduling event-triggered messages, as the ones associated with the error recovery mechanism.

To comply with the requirements stated on Section 7.1 the following adaptations must be made:

(i) introduction of a new node, termed *RetNode*, responsible for online scheduling of message retransmissions. The limitation here is that this scheduling cannot be integrated with the regular message scheduling, as this one is static;

(ii) the organization in basic cycles already fulfills this requirement, being nevertheless necessary to introduce a new window to transmit a message that triggers message retransmissions, that is termed *RetM*;

(iii) as the transmission medium is a bus, this guarantees that the bus error detector that resides in the *RetNode* listens to all transmitted messages and promptly detects any failed transmission, assuming consistent message receptions.

A graphical representation of the network architecture and the new windows placement is represented in Figure 7.2, that will be further discussed next.

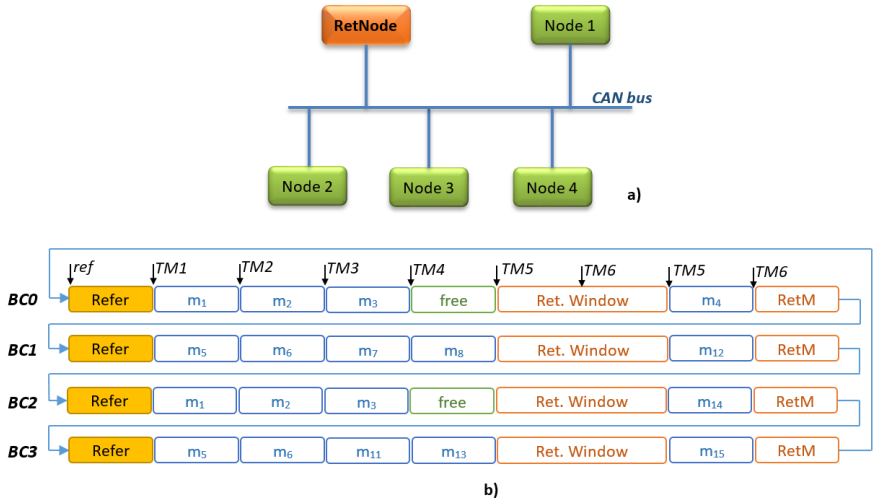

Figure 7.2: Proposal for error recovery in TTCAN protocol, with central *RetNode*. a) Network architecture; b) Basic Cycles with windows for message retransmissions and *RetM* message.

7.2.1 Windows Placement and Size

The original message set must be extended with one or more windows to send the *RetM* message and the messages that need to be retransmitted, referred above.

We start by analyzing the characteristics of the *RetM* message and window, that will be followed by the window for messages retransmission.

7.2.1.1 *RetM* Message and Window

Due to the random nature of faults, errors can happen in any instant, in any cycle. As it is desirable to promptly recover the errors, before the correspondent message deadlines, the request to retransmit must be done as soon as possible. Considering that the fast messages have periods/deadlines equal to 2 basic cycles, then to have a prompt response, a window specific to *RetM* message transmission must exist in every basic cycle. This is the supposition that we will use in the following analyses, which constitutes an upper bound, since considering that the fastest messages present larger periods, this requirement for window placement could be relaxed.

We assume that it is possible to transmit the *RetM* in the same basic cycle where the error is detected, placing it at the end of the cycle which will allow sufficient time to the *RetNode* to process the error information, to build the *RetM* message and place it in the CAN controller transmission buffer on time to be transmitted in the predefined window.

This can place a restriction on the messages that can be recovered in the following cycle, as for instance the messages that are transmitted in the window immediately before the

RetM window may not have errors processed and subsequent operations done on time, so to be included in the *RetM* message. One possible solution to overcome this is to grow the last window with the time needed by the *RetNode* to do all error processing work and still transmit *RetM* in the current cycle, otherwise this message would have a recovery latency equal to two cycles. This detail must be analyzed for the particular microcontroller used, as different families can have very different computing capacities.

The *RetM* conveys the information on what messages should be retransmitted, its identifier (part of CAN ID) and the number of copies that must be sent in the next cycle. This can be accomplished, e.g., using only 1 byte per message, coding the ID with 6 bits (for a system with a maximum of 64 messages) and the remaining 2 bits representing the number of copies (e.g., 0b00 - 1; 0b01 - 2; 0b10 - 3 and 0b11 - 4), if 4 copies are enough to attain the global recovery objective.

Then, assuming *max_1cycle* errors per cycle, using 1 byte per error in the *RetM* message payload and considering worst case bit stuffing, the message size is given by Equation (7.1).

$$RetM_{MAX} = 55 + 10 \cdot max\_1cycle \tag{7.1}$$

To guarantee that errors affecting this message do not hinder its transmission, this window can also be extended to allow more than one copy to be transmitted.

7.2.1.2 Retransmissions Window

One possible approach for message retransmission is to define a window for each node, with a duration equal to the longest message transmitted by this module, allowing for the maximum *RepLevel* number of messages. This strategy is not bandwidth efficient, and even with a small number of nodes it would rapidly reserve too much bandwidth. For instance, a system with 5 nodes, each one transmitting a maximum size message equal to 135 bits and maximum *RepLevel* of 3, would need 5 windows, each one with size 474 bits (that includes error frames of 23 bits), making a total of 2370 bits in each cycle. If the basic cycle has a length of 2500 bits (2.5 ms with bit rate of 1000 kbps), then this allocation scheme uses a significant part of the available bandwidth, restricting severely the system schedulability.

A more wise approach would be to use a large window, that will be shared by all nodes, as it is not necessary to have in all cycles reserved capacity for each node, as the fault model used has a bound on maximum number of faults per cycle, that surely is lower, even for Agressive environments, than the number of nodes. Of course, to allow that any node can send messages in this window, it must be of arbitration type.

So, the arbitration window used for message retransmission must be large enough to accommodate the number of replicas necessary to guarantee a successful retransmission. This can be obtained by a similar process described for the *Controlled Retransmission* method, obtaining the number maximum of errors in one cycle and the *RepLevel* vector. Then the vector *N_Ret* is constructed, Equation (7.2), that describes the number of messages copies that should be retransmitted, for all possibilities in number of errors. Considering that

maximum size messages need to be retransmited, we obtain the window length - Equation (7.3), that must also include a term for the maximum number of error frames, where r is a number between 1 and max_1cycle, that accounts for errors in previous cycle (limited by the fault scenarios sequences). Thus the arbitration window used for retransmission should have this length and be placed after the *RetM* window, as can be observed in Figure 7.2.

$$N\_Ret = \{1 \cdot RepLevel(1), 2 \cdot RepLevel(2), ..., max\_1cycle \cdot RepLevel(max\_1cycle)\} \quad (7.2)$$

$$W\_Ret = MAX_{i=1..max\_1cycle}(N\_Ret) \cdot MAX_{i=1..n}(C_i) + (max\_1cycle - r) \cdot C_{error\_frame} \quad (7.3)$$

The maximum bandwidth overhead is the one of the *RetM* window, considering *RepRetM* copies, plus a large enough arbitration window to cope with highest combination of necessary retransmissions in the following basic cycle, that is given by Equation (7.4).

$$Overhead\_BW = RetM_{MAX} \cdot RepRetM + W\_Ret \quad (7.4)$$

7.2.2 Application Example and Additional Comments

Using an example system working in an Aggressive environment, that presents max_1cycle equal to 4 and $RepLevel = \{3, 3, 2, 1\}$, we obtain using Equation (7.3) a window with size equal to 856 bits, which corresponds to 34.2% of a basic cycle, if this has a length equal to 2500 bits. We also have to consider the window for *RetM* message and its copies, that add another 354 bits to the recovery method overhead (3 copies were considered, each with a payload equal to 4 bytes, including error frames). So we end up with a total of 1210 bits, which is approximately 48.4% of the 2500 bits cycle. This seems a very large overhead, but in fact must be compared with static TT strategies, that tend to consume much greater bandwidth, as presented before. Of course, with bigger cycles this overhead would be more acceptable.

Applying this approach to *Updated_SAE* benchmark we will get an overhead equal to 43.6% on top of the necessary BW to schedule regular messages. This value is larger than the one encountered for FTT-CAN, which was 17.2%, having as a plus the operational flexibility, that is not available in TTCAN. Remember also that a pure static TT approach could not get a feasible schedule for this benchmark.

When a pure static approach is used, a system in an aggressive environment with high-reliability goal, each message is sent with four copies per message period, thus limiting the maximum bandwidth utilization to less than 25%. With the approach for TTCAN presented here, and considering the same ambient that accounts for maximum *RepLevel* equal to 3, then the maximum bandwidth utilization is bounded to less than 51.6%, which is a good improvement when compared with the 25% bound of pure static TT approaches.

The protocol can be made more bandwidth efficient if the *RetM* payload could be sent on a message that was already transmitted in all basic cycles. The TTCAN Reference message has available payload - 8 bytes for Level 1 and 4 bytes for Level 2 (see Section3.3.1.1), that can be used to convey *RetM* content. In this case, the Retransmission Window must be separated with sufficient time so the slowest node can decode the *RetM* message contents, prepare the message to retransmit and put it in the CAN buffer, so preferably this window should be placed towards the end of the basic cycle. This is illustrated in Figure 7.3, that corresponds to the same example presented in Figure 7.2, with this alteration introduced.

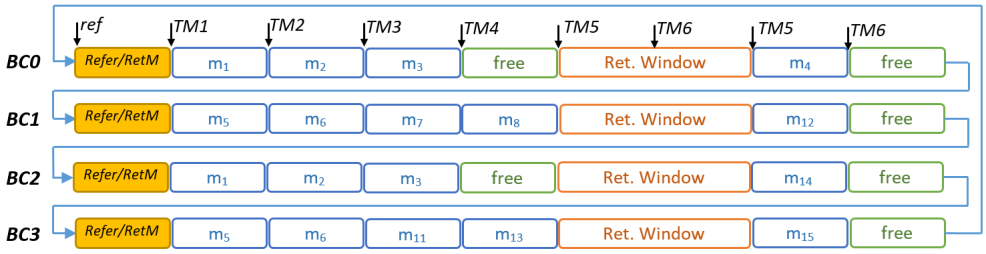

Figure 7.3: Error recovery in TTCAN, using payload of TTCAN Reference message to transmit *RetM* message.

This corresponds to an effective improvement on protocol efficiency, as the total time used decreases by the size of the *RetM* copies, that is now free for scheduling other messages (last slot in each basic cycle). This would increase the bound for bandwidth utilization, in example given previously, to 65.8% of the available bandwidth, instead of only 51.6%.

Moreover, this also implies that the scheduling process would run in the various nodes that are Time Master's candidates, giving enhanced guarantees in availability of the recovery mechanism triggering, since a Time Master (principal or substitute) must be available at all times, for correct TTCAN functioning.

Other aspect refers to the bandwidth waste in no-error scenarios, as the retransmission window is reserved for message recovery, and cannot be used to transmit other traffic. So there is a waste in available bandwidth equivalent to the reserved time. This waste can be somehow mitigated by allowing the sending of any existing asynchronous traffic in this window, additionally to the windows reserved for it. This will allow a reduction of the average response time of this type of messages, as any unused time by the retransmission messages can then be used by event-triggered messages. Possible interference with retransmission messages in this window is avoided by assigning higher IDs to the event traffic messages.

If nodes with sufficient resources and computing power are available in the system, then the actions performed by this node can be delegated on an already existent node. In this case, the introduction of the error recovery method is cost free.

The error recovery for TTCAN method also implies the introduction of new code in the slave nodes, to process the *RetM* message and to execute the actions triggered by this same message, that must be checked for implementability, specially if some slave nodes possess limited processing power.

Finally we refer that the original message set, that must be extended with the window for message *RetM* and copies and the window for message retransmissions, can be scheduled by any available algorithm, as for example the ones referred in Section 3.3.1.2, imposing restrictions on this windows position in the basic cycle, as pointed out when the window position allocation was discussed.

7.3 Error Recovery Applied to FlexRay Protocol

In FlexRay, at the communication cycle level, the time is divided in four segments: Static Segment, Dynamic Segment, Symbol Window and Network Idle Time (Section 3.3.3.3). Messages can be transferred in the two first segments, being the access in the first one made by TDMA, with a fixed number of slots, all of the same size. In the Dynamic Segment the access is made through a mini-slotting scheme, being this segment intended to transmit event based messages.

Firstly we assume that the messages that we want to provide guarantees on successful transmission are the ones of TT nature, that are sent in the Static Segment.

To apply the proposed recovery mechanism, that uses temporal redundancy, we must discuss several aspects:

- the necessity of introducing new nodes;
- what segment to use for retransmissions, considering protocol specification and bandwidth efficiency;
- timing limits and protocol overhead estimation.

Similarly to the recovery method proposed for TTCAN protocol, in previous Sections, an additional node is needed, termed *Master_R*, that will be responsible for triggering the messages that need retransmission. So, the dynamic scheduling is only applied to the eventual retransmissions.

An example system architecture is depicted in Figure 7.4, where it was considered a broadcast medium, a bus, being the network composed by 3 nodes plus the *Master_R* node.

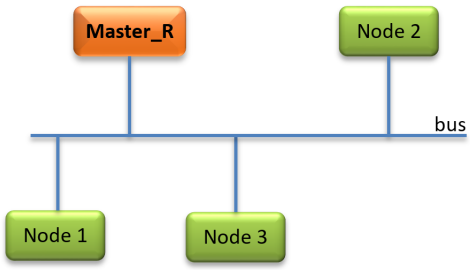

Figure 7.4: Example of FlexRay network with 3 nodes plus *Master_R*.

This node uses the FlexRay Communications Controller capabilities to listen the messages transmitted in the Static Segment and detect all failed transmissions. The *Master_R* node

must possess information on Static Segment schedule of the messages that it protects and the *RepLevel* vector, necessary to adjust the necessary number of retransmissions. After failed transmission detection, the respective message information is inserted in an error queue. Immediately after the end of the Static Segment, *Master_R* builds a special message, called Retransmissions Message - *RetMsg* - running an adequate scheduler over the information held in the error queue, and sends this message to all nodes. The information content of *RetMsg* are the IDs of messages that need to be retransmitted in the next cycle (in the next Dynamic Segment) and the number of copies of each one. It is possible to send several replicas of *RetMsg*, being the number of copies chosen according to the expected error incidence, as defined by the fault model.

When a slave node receives the *RetMsg* message, processes it and, if instructed to do so, places the unsuccessful TT message in the output queue for transmission in the Dynamic Segment. With this sequence of events, depending also on the retransmission scheduling algorithm, in the best case, the retransmission will be carried out in the next cycle. This corresponds to a recovery latency equal to one cycle, with the introduction of jitter as the retransmitted message uses a different segment and cycle than the regular ones. The *RetMsg* message also includes information on the number of copies that should be sent, guaranteeing a restransmission success (within a given probability). So the node must transfer this number of copies of the message to the communication controller buffers as fast as possible.

Figure 7.5 illustrates the error recovery process, where the message sent in Static Slot number 2 suffers an error in cycle i. The *Master_R* node promptly detects this occurrence, placing it in the error queue and after scheduling it sends the *RetMsg* with parameters $\{ID = 2; Rep = 2\}$, in the same cycle, using the allocated minislot of the Dynamic Segment, in this case the second minislot of this segment. This way, the node that produces message m_2 is instructed to prepare the message (assembling it and putting in the correspondent Communication Controller transmission buffers), that afterwards, in cycle $(i + 1)$, will retransmit the message copies, using specific minislots allocated for this node. In the example presented, it uses the minislots with ID 9 and 15, supposing that these minislots are the ones associated with the node that uses the static slot number 2. Also, in this example, we are considering the number of copies per retransmission equal to 2.

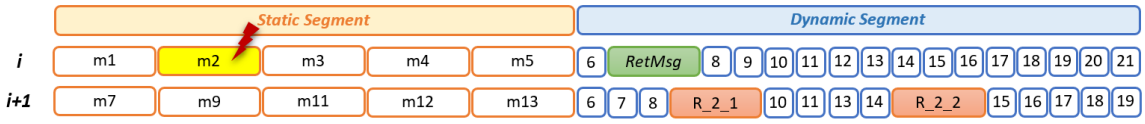

Figure 7.5: Error recovery in next cycle, using retransmission in the Dynamic Segment.

Notice that the *RetMsg* does not use the first slot in the Dynamic Segment, but a later one. This provides *Master_R* time to detect and process errors, execute the scheduling algorithm and send the message. The most stringent situation is when the error occurs in the last slot of the Static Segment. More, the *Master_R* module can be a new physical hardware node or a virtual one. The later is possible if all its functionalities could be integrated in a existing

node as an extra software module. Either way, the functions performed will be the same and the option physical/virtual will be made only in the implementation phase, after analyzing the computing power and of the available hardware nodes.

7.3.1 Segment Choosing and Slot Configuration

The schedule for the Static Segment, as defined in the protocol, is defined at pre-runtime, thus restricting the applicability of the generic method since it is not possible to apply freely dynamic scheduling, due to the association between slots and nodes. This introduces some limitations on method implementation in this protocol, that are further discussed next.

A key aspect is to decide in which segment to send the messages. Regarding the *RetMsg* message, using the Static Segment implies reserving one or more slots, every FlexRay cycle for this purpose, in order to have a prompt recovery. This leads to a significative bandwidth waste, since most of the time this message would be empty. To illustrate this waste, consider a simple system example: bit rate = 2.5 Mbps; Cycle = 5ms; $BER = 2.6 \cdot 10^{-7}$. On average there will be 1 error every 307 cycles, meaning that 306 out of 307 times, the copies of *RetMsg* messages would be empty as there is no need to trigger retransmissions, wasting several full static slots with no purpose. The wasted bandwidth is even greater if we consider more benign environments as, in average, the time interval between error occurrences is greater.

A better alternative is to send the *RetMsg* message (and its copies) in the Dynamic Segment. Due to the nature of this segment, the chosen minislot must be available every cycle and in case there is no need to transmit (no error in current cycle) the wasted bandwidth is minimal as only one minislot time is really wasted, being this duration much smaller than the duration of a static slot. Therefore, the obvious choice is to use the Dynamic Segment to send *RetMsg* messages, since it uses the available bandwidth more efficiently.

In what concerns the message retransmissions, a similar reasoning as for the *RetMsg* can be done concerning the segment in which the nodes should send their retransmissions, specially as the number of messages to retransmit is greater. Therefore, for the sake of bandwidth efficiency and prompt recovery, we also opted to use the Dynamic Segment.

So, the *RetMsg* copies and the message retransmissions are all transmitted in this segment.

The next issue is how to configure the minislot IDs reserved for the retransmissions of messages in the Dynamic Segment, including the *RetMsg* copies. Instead of allocating a minislot per message, a more efficient option in terms of bandwidth is to reserve one minislot per node, which is used by that node to recover any of its messages, following a line of reasoning similar to the approach in TTCAN proposal.

To get a measure on the method overhead it can be defined a virtual slot in the Dynamic Segment, that contains the following components:

- time necessary to transmit the *RetMsg* and its copies;
- retransmission of a defined number of copies per failed message transmissions (in the previous cycle), considering maximum size messages;

- minislots unused, reserved for nodes that do not retransmit messages.

Considering a system with a total of l nodes, the virtual slot length is given by Equation (7.5), that corresponds to the worst case length of the Virtual slot. In this equation $Size()$ is a function that gives the duration of a message transmission in μs, N_RetMsg is the number of copies of $RetMsg$ and n is the number of TT messages (transmitted in the Static Segment). The N_Ret is the vector in Equation (7.6), which defines the total number of messages used to recover errors in the previous cycle, with adequate replication level. Then, this vector represents the number of copies to retransmit, across all error and recovery scenarios.

$$\begin{aligned} VirtualSlot_{MAX} = {} & N\_RetMsg \cdot Size(RetMsg) + \\ & + MAX_{i=1..max\_1cycle}(N\_Ret) \cdot MAX_{i=1..n}(C_i) + \\ & + (l \cdot MAX(Replevel) - MAX_{i=1..max\_1cycle}(N\_Ret)) \cdot Size(minislot) \end{aligned} \tag{7.5}$$

$$N\_Ret = \{1 \cdot RepLevel(1), 2 \cdot RepLevel(2), ..., max\_1cycle \cdot RepLevel(max\_1cycle)\} \tag{7.6}$$

This formulation was obtained considering that a frame packing algorithm was performed in the initial message set and it is possible to obtain a compact message set where only one message per node and per cycle exists. For other schemes Equation (7.5) would have to be adapted.

In error free scenarios, the time used by the proposed recovery mechanism is given by Equation (7.7), which represents a minimum wasted time and the remaining time allocated in this segment could be used by other messages. Any event triggered traffic would then benefit from the remaining segment time and obtain a lower average response time. Notice that most of the time, as shown in the start of this section, this is the waste that occurs.

$$\begin{aligned} VirtualSlot_{min} = {} & N\_RetMsg \cdot Size(minislot) + \\ & + l \cdot MAX(RepLevel) \cdot Size(minislot) \end{aligned} \tag{7.7}$$

Figure 7.6 represents an example of events and message transmission, considering a system with 5 nodes and using 3 copies per message retransmitted, as the $RepLevel$ considered is $\{3, 3, 2, 1\}$ and max_1cycle and max_cycles both equal to 4. Considering the error and recovery scenario with 2 errors in consecutive cycles, the messages transmitted are the ones presented in this figure and the longest virtual slot occurs in cycle $(i + 1)$, being its value given by Equation (7.5).

Figure 7.6: Error recovery in next cycle - maximum overhead for the example considered.

7.3.2 Application Example and Protocol Efficiency Assessment

Following an example presented by BMW [Sch07], which specifies a cycle of 5 ms, frames with 16 bytes of payload and a MacroTick (MT) equal to $2\mu s$, being the payload capacity considered a fair value for new applications and a good compromise to connect to other networks, namely CAN which has maximum payload of 8 bytes. It was also considered a total of 10 nodes, connected by a single bus and using a bit rate of 2.5 Mbps. More, the Static segment has duration equal to 3 ms, that corresponds to 27 slots and complying with the constraints imposed by the FlexRay specification [Fle10], a minislot with 5 MT ($10\mu s$) was chosen, which is a conservative value for this bit rate, having approximately 200 minislots in the Dynamic Segment.

Thus, applying Equation (7.5) the maximum virtual slot is $1140\mu s$, i.e., 22.8% of the time available in a cycle, being this obtained using $max\_1cycle = 4$ and maximum $RepLevel = 3$. This represents a significant improvement over the static TT approaches, as for instance the one presented in [TBEP10], which corresponds to more more than 300% bandwidth increase, when 4 copies are needed per message period.

The bandwidth use depends on how (how many and when) errors effectively occur, depending also in specific message size and on the number of errors per cycle that must be tolerated. Nevertheless, we can enforce an upper bound to the bandwidth use, e.g. via a server, and thus give real-time guarantees to event-triggered traffic. For instance, keep following the same example, a guaranteed (2000-1140) μs are available for event messages, every Dynamic Segment. The average response should be better, as the available time is equal to the Dynamic Segment duration minus the minimum Virtual slot duration, which is (2000-340) μs, in this example. The 1140 μs and 340 μs correspond to the maximum and minimum Virtual slots, obtained by Equation (7.5) and (7.7), respectively.

As shown, only a fraction of the available bandwidth is necessary to implement the recovery mechanism, not compromising future system expansion. The recovery time is fast, since an error detected in a given cycle is triggered for retransmission in the same cycle and recovered in the following one.

An improvement in recovery latency is possible by retransmiting the messages in the same cycle where errors were detected. This is represented in Figure 7.7, where the *RetMsg* triggers the retransmission of message m_1 and m_2 with 2 copies each, supposing that these were the messages that suffered errors in the Static Segment. In Figure 7.7, *RetMsg* is also configured with 2 copies transmission.

The method would work in the following way:

- *Master_R* node must detect failed messages, build the *RetMsg* and transmit it, starting

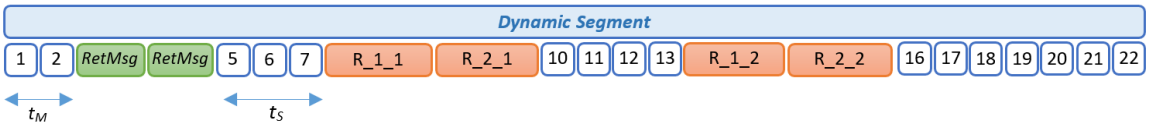

Figure 7.7: Constraints to obtain in-cycle recovery.

in the earliest possible minislot;

- slaves must decode the *RetMsg* message and put immediately the message(s) to retransmit in the correspondent transmission buffers.

The first restriction corresponds to the time interval, say t_M, that must exist in the beginning of the Dynamic Segment, prior to the first *RetMsg* transmission. This is necessary since the *Master_R* node must detect the latest possible error, that would occur in the last slot of the Static Segment, process it, build and transfer the *RetMsg* to the communication controller buffer. The second one, the t_S interval, accounts for the time the slaves need to receive and process the *RetMsg* message and place in the communication controller the messages to retransmit. To make the process faster, each slave node should maintain in memory copies of the last messages just transmitted in the Static segment, so if instructed to retransmit, the message is already assembled.

This will introduce additional restrictions on choosing the slots positions in the recovery method, that are acceptable if fast microcontrollers are used. For instance a microcontroller with 100 MHz clock and single cycle instruction processes 500 instructions in 5 μs, that should be enough for the slaves to decode the *RetMsg*. These are small values, that do not compromise the method applicability, being a small price to pay for the recovery latency improvement obtained. Nevertheless, this should be measured experimentally, to guarantee correct functioning.

7.4 Summary

This chapter started by presenting a generic model to attain high-reliability using time redundancy in Time-Triggered systems. It discusses next its application to two known protocols: TTCAN and FlexRay. Details on how the method could be applied, their limitations and overhead were also presented. As for FTT-CAN, in both cases retransmissions are scheduled centrally. This also implies the introduction of additional messages and code in the slave nodes, necessary to comply with message retransmission triggering.

Resuming, the proposal for error recovery in TTCAN and FlexRay uses the following functionalities to implement time redundancy:

- failed messages detection, done by a specific node, *RetNode* or *Master_R*;
- Messaging trigger mechanism in the nodes, for retransmssions only;
- Compliance, by the slave nodes, to the triggers sent in special messages *RetM* or *RetMsg*.

The proposal achieves a low error recovery time, typically equal to one cycle, and also the bandwidth used is very small when compared to extra static windows/slots in static scheduling approach. In FlexRay, an alternative on the recovery method would allow in-cycle error recovery, with very small overhead increase.

It would be interesting to analyze the applicability of the proposed recovery method to emerging Ethernet based protocols with real-time guarantees, as for instance TTEthernet, AVB or FTT-HaRTES.

Chapter 8

Conclusions and Future work

Distributed Embedded Systems (DES) are pervasive in technological and advanced societies, controlling all sorts of equipment and machinery. The computing nodes that constitutes the DES use a communication network to exchange messages, cooperating between them to fullfill its intended function.

The DES are often used in systems that are safety-critical or at least need to present high-levels of reliability, so the underlying network has also to present them. But, as in any real network, transmission errors will always be present, which must be resolved to obtain the intended reliability of this subsystem. The network has to assure the timely and correct delivery of messages, despite the error occurrence, being this typically obtained using fault tolerance techniques.

The thesis here presented argues that is possible to obtain real-time functioning, with adequate reliability level, recurring to message retransmission that are centrally scheduled. Also, obtaining this goal is compatible with prompt error recovery and attaining bandwidth efficiency.

One contribution is this new method, that uses a CAN network with a Time-Triggered setting, using the FTT paradigm, taking advantage of its online scheduling of the time-triggered traffic. Then by adding a module to detect errors in each Elementary Cycle and a server to manage the errors detected, it is possible to make error recovery in the following cycle. The retransmissions requests are then fed to the online scheduler by the error server, which integrates these requests in the scheduling process of regular TT messages. The server choice and its parametrization is obtained using limit error scenarios bounded by the fault model used, identifying the most demanding error scenarios and determines the level of replication needed to guarantee error recovery in the next FTT-CAN elementary cycle. All the necessary steps were discussed in detail, including the corresponding algorithms.

To assess the proposed recovery method an event simulator of FTT-CAN networks was built, which constitutes also a contribution of this work. The simulator includes an error pattern generator, with single bit-errors and limited to one per EC or with compound sequences of multiple errors in consecutive cycles, corresponding to the fault scenarios previously identified.

Then, the proposed error recovery mechanism was simulated with several well-known benchmarks, as well as random message sets and is compared with other methods available in the literature. The simulations showed that the proposed method is effective in promptly recovering errors in time-triggered messages, using approximately two orders of magnitude less average bandwidth than other approaches, while the instantaneous bandwidth required by our method (average of the random sets) is also always lower than the one used by other methods, allowing for the application of the error recovery method to message sets with greater bandwidth utilization.

On the other hand, our method presents a recovery latency equal to one cycle while other approaches can recover in the cycle in which the error occurred. This has implications on Elementary Cycle time choosing and may force our mechanism to use shorter cycles to handle fast messages appropriately.

The proposed method was tailored for a specific protocol, the FTT-CAN, but possess some generic characteristics that were identified, which allowed it to be abstracted from specific protocols, making it generic to Time-Triggered protocols. Then, with some adaptations it can be applied to different protocols, being this discussed for the TTCAN and FlexRay protocols, which were previously presented in Chapter 3.

As these protocols use static schedules, contrary to FTT-CAN, it is not possible to schedule retransmissions together with regular TT traffic. An additional module must be added to the system, that listens to all bus traffic and schedules the retransmissions, sending a special message to trigger retransmissions in the slaves, that must be updated with additional code to respond accordingly. The TTCAN protocol uses a single Arbitration window, shared by all nodes and where they can send their retransmissions. In FlexRay networks, due to TDMA access, it is used the Dynamic Segment, where reserved minislots per node are used to retransmit messages. The performed analysis in both protocols allows to conclude that the new recovery method is more bandwidth efficient than the static Time-Triggered approaches.

8.1 Future work

An investigation work is never completed, as the research for solutions for the initial problem always give rise to a myriad of new unresolved questions. Due to lack of time, material limitations or simply because they are slightly out of the scope of the proposed work, these new directions are, most of the time, not followed.

Some of the directions that can be pursued in the future are the following:

- Admit different error detection scenarios, as for instance the ones when there is an inconsistent view of errors by the diverse recipients;

- Extend our approach to other error models than the Poisson single bit error model, so that we can accurately apply our method to situations with error bursts and periods of sustained higher interference;

- The proposal in this thesis copes only with transient faults, so recurring to the proposal of FTT with multiple buses [Sil10], and use it as a base to extend the fault tolerance mechanism to tolerate permanent failures, buses, communication controllers and nodes, increasing this way the overall system dependability;
- The optimization and reduction of the complexity in the schedulability and analysis techniques is another direction to pursue.

By joining the subjects studied in this work, with the author teaching activities and interest in communication networks for DES, a line of possible future work is also to develop a implementation of a FTT-CAN FD version, with multiple bus access (two just to start), for a 32 bit microcontroller family and test it with a hardware fault injector. After development and correct functioning verification, this FTT-CAN FD implementation would be made publicly available, free of charge.

Bibliography

[ABR+93] N.C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[AF01] L. Almeida and J. A. Fonsec. Analysis of a simple model for non-preemptive blocking-free scheduling. In *Proceedings of the ECRTS01 (EUROMICRO Conf. Real-Time Systems)*, pages 233–240, Delft, The Netherlands, June 2001. 13-15 June.

[AFD05] Aircraft data network, part 7: Avionics full duplex switched ethernet (afdx) network. arinc specication 664, 2005.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):1–23, January 2004. Jan-March 2004.

[AP12] F. Ataide and C. Pereira. Ftt-can: Estudo de uma aplicao automotiva. *Revista Controle & Automação*, 2012.

[APF02] L. Almeida, P. Pedreiras, and J. A. Fonseca. The ftt-can protocol: why and how. *IEEE Transactions on Industrial Electronics*, 49(6):1189–1201, December 2002.

[APL+06] F. Ataide, C. Pereira, W. Lages, , and A. Assis. On the design of an embedded ftt-can platform with improvement of its inherent jitter. In *Proceedings of the 4th International IEEE Conference on Industrial Informatics*, Singapore, August 2006.

[BB99] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 6878, Phoenix, AZ, USA, December 1999. 13 December 1999.

[BB03] I. Broster and A. Burns. The babbling idiot in event-triggered real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium - Work-In-Progress Session*, 2003.

[BBB03] E. Bini, G. Buttazzo, and G. Buttazzo. Rate monotonic scheduling: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, July 2003.

[BBRN02] I. Broster, A. Burns, and G. Rodriguez-Navas. Probabilistic analysis of can with faults. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, page 269278, Austin, USA, December 2002. 35 December.

[BBRN04] I. Broster, A. Burns, and G. Rodrguez-Navas. Comparing real-time communication under electromagnetic interference. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS04)*, Catania, Sicily, Italy, June 2004. June 30 - July 2, 2004.

[BDBP06] A. Ballesteros, S. Derasevic, M. Barranco, and J. Proenza. First implementation and test of reintegration mechanisms for node replicas in the ft4ftt architecture. In *Proceedings of the 21th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*, Berlin, 2006.

[BG91] Robert Bosch GmbH. Controller Area Network (CAN) specification - version 2.0. Technical report, Bosch GmbH, Robert, 1991.

[BG12] Robert Bosch GmbH. CAN with flexible data-rate - specification version 1.0. Technical report, Bosch GmbH, Robert, 2012.

[BPG00] J. Berwanger, M. Peller, and R. Griessbach. byteflight - a new protocol for safety critical applications. In *Proceedings of the FISITA World Automotive Congress*, Seoul, Korea, June 2000. 12-15 June.

[BS14] Unmesh D. Bordoloi and Soheil Samii. The frame packing problem for can-fd. In *Proceedings of the 2014 IEEEReal-Time Systems Symposium (RTSS), 2014 IEEE*, Rome, Italy, December 2014. IEEE.

[But11] G. C. Buttazzo. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*. Springer USA, New York, 3rd edition, 2011.

[CBF01] F. Coutinho, J. Barreiros, and J.A. Fonseca. Scheduling for a ttcan network wtth a stochastic optimization algorithm. In *Proceedings of the IFAC international conference on fieldbus systems and their applications*, Amsterdam, The Netherlands, 2001.

[CSSLC00] P. Castelpietra, Y.-Q Song, F. Simonot-Lion, and O. Cayrol. Performance evaluation of a multiple networked in-vehicle embedded architecture. In *Proceedings 2000 IEEE International Workshop Factory Communication Systems*, pages 187–194, Porto, Portugal, September 2000. IEEE.

[DNGG12] H. Di Natale, M.and Zeng, P. Giusto, and A. Ghosal. *Understanding and Using the Controller Area Network Communication Protocol Theory and Practice*. Springer-Verlag, 1st edition edition, 2012.

[FAF+06] J. Ferreira, L. Almeida, J. A. Fonseca, P. Pedreiras, E. Martins, G. Rodriguez-Navas, J. Rigo, and J. Proenza. Combining operational flexibility and dependability in ftt-can. *IEEE Transactions on Industrial Informatics*, 2(2):95–102, May 2006.

[FAFF04] J. Ferreira, O. Arnaldo, P. Fonseca, and J.A. Fonseca. An experiment to assess bit error rate in can. In *Proceedings of 3rd International Workshop of Real-Time Networks (RTN2004)*, pages 15–18, 2004.

[Fer05] J. Ferreira. *Fault-Tolerance in Flexible Real-Time Communication Systems*. PhD, Universidade de Aveiro, Aveiro, 2005.

[Fle05] FlexRay Consortium. Flexray communications system, electrical physical layer specication, version 2.1. Technical report, FlexRay Consortium, May 2005.

[Fle10] FlexRay Consortium. Flexray communications system, protocol specication version 3.0.1 revision a. Technical report, FlexRay Consortium, December 2010.

[FPAF02] J. Ferreira, P. Pedreiras, L. Almeida, and J. A. Fonseca. Achieving fault tolerance in ftt-can. In *4th IEEE International Workshop on Factory Communication Systems (WFCS'2002) Proceedings*, pages 125–132, Västerås, Sweden, August 2002.

[FTT18] http://paginas.fe.up.pt/ ftt/sections/repository/index.html, 2018.

[GPBB19] D. Gessner, J. Proenza, M. Barranco, and A. Ballesteros. A fault-tolerant ethernet for hard real-time adaptive systems. *IEEE Transactions on Industrial Informatics*, 15(5):2980–2991, May 2019.

[HMFH02] F. Hartwich, B. Mller, T. Fhrer, and R. Hugel. Timing in the ttcan network. *CAN Newsletter*, 2002.

[HP03] M. G. Harbour and J.C. Palencia. Response time analysis for tasks scheduled under edf within fixed priorities. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 2003. 5.December.

[iA15] CAN in Automation. CAN in Automation(CiA): Controller area network (can). http://www.can-cia.org, July 2015.

[Inc07] Microchip Technology Inc. *PIC18F2585/2680/4585/4680 Data Sheet*. Microchip Technology Inc, Chandler, AZ, USA, 2007.

[Inc17] Microchip Technology Inc. *ATSAME51J18A Data Sheet*. Microchip Technology Inc, Chandler, AZ, USA, 2017.

[Int93] SAE International. Class c application requirement considerations - sae technical report j2056/1. Technical report, SAE International, PA, USA, June 1993.

[KAGS05] H. Kopetz, A. Ademaj, P. Grillinger, and K. Steinhammer. The time-triggered ethernet (tte) design. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005 (ISORC 2005)*, Seattle, WA, USA, May 2005. IEEE.

[KB03] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, January 2003.

[KCM05] J. Kaiser, B. Cristiano, and C. Mitidieri. Cosmic: A real-time event-based middleware for the can-bus. *Journal of Systems and Software*, 77(1):27–36, July 2005.

[KL99] J. Kaiser and M. Livani. Achieving fault-tolerant ordered broadcasts in can. In Jan Hlavička, Erik Maehle, and András Pataricza, editors, *Dependable Computing — EDCC-3*, pages 351–363, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer US, 2nd edition, 2011.

[Lap95] J.-C. Laprie. Dependable computing: Concepts, limits, challenges. In *Proceedings of the 25th Int. Symp. on Fault-Tolerance Computing (FTCS-25)*, Pasadena, USA, June 95.

[Law13] Wolfhard L. Lawrenz. *CAN System Engineering: From Theory to Practical Applications*. Springer, 2nd edition, 2013.

[LH02] G. Leen and D. Heffernan. Ttcan: a new time-triggered controller area network. *Microprocessors and Microsystems*, 26(2):77–94, March 2002.

[Lin12] T. Lindenkreuz. Can fd - can with flexible data-rate (vector kongress 2012 presentation). on-line: https://vector.com/portal/medien/cmc/events/commercial_events/VectorCongress_2012/VeCo12_ 2012.

[LL73] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):40–61, January 1973.

[LNZ+09] W. Li, M. Natale, W. Zheng, P. Giusto, A. Sangiovanni-Vincentelli, and S. Seshia. Optimizations of an application-level protocol for enhanced dependability in flexray. In *Proceddings of Design, Automation & Test in Europe Conference & Exhibition*, pages 1076–1081, Nice, France, April 2009. 20-24 april.

[LSS87] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 261–270, San Jose, CA, USA, December 1987.

[MAF+06] R. Marau, L. Almeida, J. A. Fonseca, J. Ferreira, and V. Silva. Assessment of ftt-can master replication mechanisms for safety-critical applications. *SAE 2006 Transactions Journal of Passenger Cars: Electronic and Electrical Systems*, April 2006. E XTRA-INFO-OPTIONAL.

[Mar09] R. Marau. *Real-time communications over switched Ethernet supporting dynamic QoS management*. PhD, Universidade de Aveiro, Aveiro, 2009.

[MB76] Robert Metcalfe and David Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[MBSP02] R. Maier, G. Bauer, G. Stöger, and S. Poledna. Time-triggered architecture: A consistent computing platform. *IEEE Micro*, 22:36–45, 2002.

[MGL+12] P. Milbredt, M. Glab, M. Lukasiewycz, A. Steininger, and J. Teich. Designing flexray-based automotive architectures: A holistic oem approach. In *Proceedings of the DATE12 - Design, Automation & Teste in Europe Conference*, Dresden, Germany, March 2012.

[MN10] U. Mohammad and N. Nizar. Development of an automotive communication benchmark. *Canadian Journal on Electrical and Electronics Engineering*, 1(5), August 2010.

[MVPA13] L. Marques, V. Vasconcelos, P. Pedreiras, and L. Almeida. Error recovery in time-triggered communication systems using servers. In *Proceedings 8th IEEE International Symposium on Industrial Embedded Systems (SIES'13)*, Porto, Portugal, June 2013.

[Nol03] Thomas Nolte. *Reducing Pessimism and Increasing Flexibility in the Controller Area Network*. PhD thesis, Malardalen University, 2003.

[NSS00] N. Navet, Y.-Q. Song, and F Simonot. Worst-case deadline failure probability in real-time applications distributed over controller area network. *Journal of Systems Architecture*, 46(7):607–617, April 2000.

[Obe05] R. Obermaissser. *Event-Triggered and Time-Triggered Control Paradigms*. Springer, 2005.

[PAG02] P. Pedreiras, L. Almeida, and P. Gai. The ftt-ethernet protocol: merging flexibility, timeliness and efficiency. In *Proceedings 14th Euromicro Conference on Real-Time Systems*, Vienna, Austria, June 2002. 1921 June 2002.

[PAK08] O. Pfeiffer, A. Ayre, and C. Keydel. *Embedded Networking with CAN and CANopen*. Greenfield: Copperhill Technologies Corporation, 1st edition, 2008.

[Par07] Dominique Paret. *Multiplexed Networks for Embedded Systems - CAN, LIN, Flexray, Safe-by-Wire ...* Chichester: John Wiley & Sons, 1st edition, 2007.

[Ped03] P. Pedreiras. *Supporting Flexible Real-Time Communication on Distributed Systems*. PhD thesis, Universidade de Aveiro, 2003.

[PF04] J. Pimentel and J. A. Fonseca. Flexcan: A flexible architecture for highly dependable embedded applications. In *Proceedings of the Third International Workshop on Real-Time Networks, Held in Conjunction with the 16th Euromicro International Conference in Real-Time Systems*, Catania, Italy, June 2004.

[PHN00] S. Punnekkat, H. Hansson, and C. Norstrom. Response time analysis under errors for can. In *Proceedings of the 6th Real-Time Technology and Applications Symposium (RTAS)*, 2000.

[PKOS04] P Peti, H. Kopetz, R. Obermaisser, and N Suri. From a federated to an integrated architecture for dependable embedded systems. Technical report, Technische Universitat Wien, 2004.

[PMJ00] J. Proenza and J. Miro-Julia. Majorcan: A modication to the controller area network protocol to achieve atomic broadcast. In *Proceedings IEEE Int. Workshop on Group Communications and Computations*, Taipei, Taiwan, April 2000.

[RVA+98] J. Rufino, P. Verissimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in can. In *Digest of Papers of the 28th IEEE International Symposium on Fault-Tolerant Computing Systems*, pages 150–159, Munich, Germany, June 1998. 2325 June.

[San11] R. Santos. *Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications*. PhD, Universidade de Aveiro, Aveiro, 2011.

[SBCH13] A. Sheikh, O. Brun, M. Chramy, and P.-E. Hladik. Optimal design of virtual links in afdx networks. *Real-Time Systems*, 49(3), 2013.

[Sch07] A. Schedl. Goals and architecture of flexray at bmw. Slides presented at the Vector FlexRay Symposium, March 2007.

[SF06] V. Silva and J.A. Fonseca. Using ftt-can to combine redundancy with increased bandwidth. In *Proceedings of the 6TH IEEE International Workshop on Factory Communication Systems*, Stockholm, Sweden, November 2006.

[SFF06] V. Silva, J.A. Fonseca, and J. Ferreira. Using "ftt-can" to the flexible control of bus redundancy and bandwidth usage. In *Proceedings of the 11TH International "CAN" Conference*, Torino, Italy, June 2006. June 28-30.

[SFF07a] V. Silva, J.A. Fonseca, and J. Ferreira. Adapting the ftt-can master for multiple-bus operation. In *Proceedings of the IEEE 5th International Conference on Industrial Informatics*, page 305310, Patras, Greece, December 2007.

[SFF07b] V. Silva, J.A. Fonseca, and J. Ferreira. Flexible bus media redundancy. In *Proceedings of the International Workshop on Dependable Embedded Systems (In Conjunction with the 26th Symposium on Reliable Distributed Systems)*, Beijing, China, April 2007.

[SFF07c] V. Silva, J.A. Fonseca, and J. Ferreira. Master replication and bus error detection in "ftt-can" with multiple buses. In *Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation*, page 305310, Vienna, Austria, July 2007. 23-27 July.

[Sil10] V. Silva. *Flexible Redundancy and Bandwidth Management in Fieldbuses*. PhD, Universidade de Aveiro, Aveiro, 2010.

[SLS95] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhancing aperiodic responsiveness in hard-real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, January 1995.

[SS07] K. Schmidt and E.G. Schmidt. Systematic message schedule construction for time-triggered can. *IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY*, 56(6):3431–3441, November 2007.

[SS10] M. Short and I. Sheikh. Computing optimal window sizes to enforce dependable communications in time-triggered controller area networks. In *Proceedings of the 9th International Workshop on Real-Time Networks*, page 3843, Brussels, Belgium, July 2010.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, July 1989.

[ST16] M. Steen and A. Tanenbaum. A brief introduction to distributed systems. *Computing*, 98:967–1009, August 2016.

[STA04] ISO INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO 11898-4 :2006 Road vehicles Controller Area Network (CAN) Part 4: Time-triggered communication, 2004.

[STM+06] F. Santos, J. Trovao, A. Marques, P. Pedreiras, J. Ferreira, L. Almeida, and M. Santos. A modular control architecture for a small electric vehicle. In *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'2006)*, pages 139–144, Prague, Czech Republic, September 2006.

[TB94] K. Tindell and A. Burns. Technical report ycs 229 - guaranteed message latencies for distributed safety-critical hard real-time networks. Technical report, Department of Computer Science, University of York, 1994.

[TBEP10] B Tanasa, U. Bordoloi, P. Eles, and Z. Peng. Scheduling for fault-tolerant communication on the static segment of flexray. In *31st IEEE Real-Time Systems Symposium*, pages 385–394, San Diego, CA, USA, December 2010.

[Vos08] Wilfried Voss. *A Comprehensible Guide to J1939*. Copperhill Technologies Corporation, 2008.

[WI11] B. M. Wilamowski and J. D. Irwin, editors. *Industrial Communication Systems*. CRC Press, second edition, 2011.

[WL88] J. Welch and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77:1–36, 1988.

[Zel17] Holger Zeltwanger. Market research - "the only statistics you can trust are those you falsified yourself". *CAN Newsletter*, (2):42–44, 2017.

Appendix A

Benchmarks

A.1 SAE

The benchmark defined in [Int93] was intended to be representative of an automotive communication system, namely in an electric car communication system. In fact, SAE class C was defined for the real-time and fault-tolerant aspects of communication in the car industry. The SAE Benchmark describes the set of signals exchanged on point-to-point links in a prototype electric car, and has been later adapted to CAN in [TB94]. The benchmark is comprised of 7 subsystems exchanging 53 messages. A optimized version, with signal packing, presented in [TB94] is in Table A.1. However, the characteristics of the SAE benchmark are not exactly the ones of a typical automotive CAN network and it is no more realistic in terms of data exchanged and number of nodes.

Table A.1: *SAE* benchmark message set

Signal Numbers	ID	DLC (bytes)	T (ms)	D (ms)
14	1	50	5	
8, 9	2	5	5	
7	3	5	5	
43, 49	4	5	5	
11	5	5	5	
32, 41	6	5	5	
31, 34, 35, 37, 38, 39, 40, 44, 46, 48, 53	7	10	10	
23, 24 , 25, 28	8	10	10	
15, 16, 17, 19, 20, 22, 26, 27	9	10	10	
41, 43, 45, 47, 49, 50, 51, 52	10	10	10	
18	11	50	20	
1, 2, 4, 6	12	100	100	
12	13	100	100	
10	14	100	100	
3, 5, 13	15	1000	1000	
21	16	1000	1000	
33, 36	17	1000	1000	

This benchmark was used in several studies.

A.2 Updated_SAE

The previous benchmark reveals itself outdated, as with new functionalities implemented in modern cars, as for instance Departure Lane and ADAS. A revised message set with the

introduction of these new functionalities was presented in [MN10]. The full list of messages is presented in Table, with the identification of each message parameter (payload in bytes, period and Deadline) and the node that transmits each message. The ID assigning was done using fixed priorities, namely Deadline Monotonic.

Table A.2: *Updated_SAE* benchmark message set

Source Node	ID	DLC (bytes)	T (ms)	D (ms)
Body Control Module	1	1	50	5
Hydraulic Brake Control Unit	2	2	5	5
Body Control Module	3	1	5	5
Engine Control Module	4	2	5	5
Transmission Control Module	5	1	5	5
Engine Control Module	6	2	5	5
Throttle Control unit	7	1	5	5
Traction Control unit	8	1	5	5
Front-Left Wheel Module	9	1	7.5	7.5
Front-Right Wheel Module	10	1	7.5	7.5
Rear-Left Wheel Module	11	1	7.5	7.5
Rear-Right Wheel Module	12	1	7.5	7.5
Body Control Module	13	1	7.5	7.5
Electronic Brake Control Module	14	4	7.5	7.5
Traction Control unit	15	4	7.5	7.5
ESP/ROM	16	4	7.5	7.5
Body Control Module	17	1	10	10
Body Control Module	18	2	10	10
Engine Control Module	19	6	10	10
Engine Control Module	20	2	10	10
Engine Control Module	21	3	10	10
Active Frame Steering	22	2	10	10
Front-Left Wheel Module	23	2	12.5	12.5
Front-Right Wheel Module	24	2	12.5	12.5
Rear-Left Wheel Module	25	2	12.5	12.5
Rear-Right Wheel Module	26	2	12.5	12.5
Active Suspension unit	27	4	12.5	12.5
ESP/ROM	28	5	12.5	12.5
Adaptative Cruise Control	29	3	12.5	12.5
Hydraulic Brake Control Unit	30	1	50	20
Body Control Module	31	4	100	100
Hydraulic Brake Control Unit	32	1	100	100
Transmission Control Module	33	1	100	100
Body Control Module	34	3	1000	1000
Engine Control Module	35	1	1000	1000
Transmission Control Module	36	1	1000	1000

A.3 PSA

In [CSSLC00]a study on a research vehicle conducted for PSA Peugeot-Citroen Automobile Company was presented, being the messages exchanged in the network the ones presented in table 6.2. Messages are referred as X, Y and Z that were not disclosed due to confidentiality reasons. The network architecture is presented in Figure A.1 has two different buses, one CAN and one VAN, interconnected by a module termed "Intelligent Switching Unit" that is responsible for protocol adaptions for the messages exchanged between buses. The VAN protocol is similar to CAN in many aspects being distinct from the later by essentially the following characteristics: maximum payload equal to 28 bytes, IDs with 12 bits and possibil-

ity of in-frame response. It also uses Enhanced Manchester bit coding and no bit-stuffing is necessary. The VAN protocol was in the meanwhile dropped by its proponents, essentially the PSA group, and is no longer used.

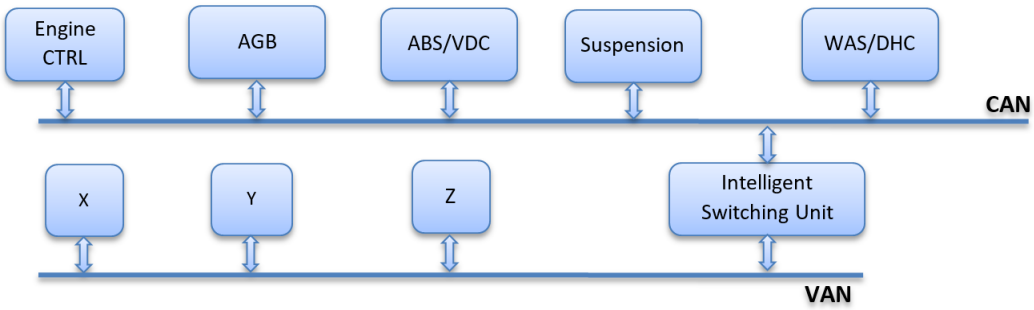

Figure A.1: PSA prototype network.

Table A.3: *PSA* Benchmark message set (original)

Source Node	Frame ID	DLC (bytes)	T (ms)	Network
Engine Control	1	8	10	CAN
Wheel Angle Sensor/DHC	2	3	14	CAN
Engine Control	3	3	20	CAN
Automatic Gear Box (AGB)	4	2	15	CAN
ABS/VDC	5	5	20	CAN
ABS/VDC	6	5	40	CAN
ABS/VDC	7	4	15	CAN
Body Application Controller (ISU)	8	5	50	CAN
Suspension Controller	9	4	20	CAN
Engine Control	10	7	100	CAN
Automatic Gear Box (AGB)	11	5	50	CAN
ABS/VDC	12	1	100	CAN
Body Application Controller (ISU)	13	8	50	VAN
Body Application Controller (ISU)	14	10	10	VAN
Y	15	16	50	VAN
X	16	4	150	VAN
X	17	4	200	VAN
Z	18	20	100	VAN
Z	19	2	150	VAN

From the original network and bus we derived a new message set, corresponding to all the messages that are transmitted in the CAN bus only, where the Intelligent Switching Unit is responsible for sending all messages from the X, Y and Z modules, adapting it to the CAN protocol limitations, namely by breaking messages that have a payload greater than 8 bytes in several messages.

The obtained message set is the one presented in Table A.4

A.4 VEIL

The VEIL is a small electric vehicle with a variety of energy sources and is equipped with X-by-wire controls for which the proposed fault-tolerance techniques are particularly relevant.

Table A.4: *PSA* Benchmark message set (adapted)

Source Node	Frame ID	DLC (bytes)	T (ms)	D (ms)
Engine Control	1	8	10	10
Wheel Angle Sensor/DHC	2	3	14	10
Engine Control	3	3	20	20
Automatic Gear Box (AGB)	4	2	15	15
ABS/VDC	5	5	20	20
ABS/VDC	6	5	40	40
ABS/VDC	7	4	15	15
Body Application Controller (ISU)	8	5	50	50
Suspension Controller	9	4	20	20
Engine Control	10	7	100	100
Automatic Gear Box (AGB)	11	5	50	50
ABS/VDC	12	1	100	100
Body Application Controller (ISU)	13	8	50	50
Body Application Controller (ISU)	14, 15	10 (5, 5)	10	10
Y	16, 17	16 (8, 8)	50	50
X	18	4	150	150
X	19	4	200	200
Z	20, 21, 22	20 (7,7,6)	100	100
Z	23	2	150	150

Figure A.2: VEIL - Small Electric Vehicle Prototype.

Table A.5: *VEIL* benchmark message set

Function	*Frame ID*	*DLC (bytes)*	*T (ms)*	*D (ms)*	Function	*Frame ID*	*DLC (bytes)*	*T (ms)*	*D (ms)*
ang_stw	1	2	10	10	volt_bat_bnk1	11	8	100	100
ang_whl	2	2	10	10	pwr	12	1	250	250
pos_brk+pos_acc	3	4	10	10	eng_bat	13	2	500	500
veil_veloc	4	1	20	20	enb_conv+enb_vfd	14	2	500	500
vpl_dcl	5	2	20	20	tmp_vfd	15	1	1000	1000
cur_im + vel_im	6	4	20	20	tmp_bat	16	1	1000	1000
eng_sc	7	2	50	50	eng_sp	17	2	1000	1000
aes	8	3	50	50	tmp_bat_bnk1	18	8	1000	1000
vol_dcl+cur_dcl	9	4	100	100	tmp_bat_bnk2	19	8	1000	1000
volt_bat_bnk1	10	8	100	100					

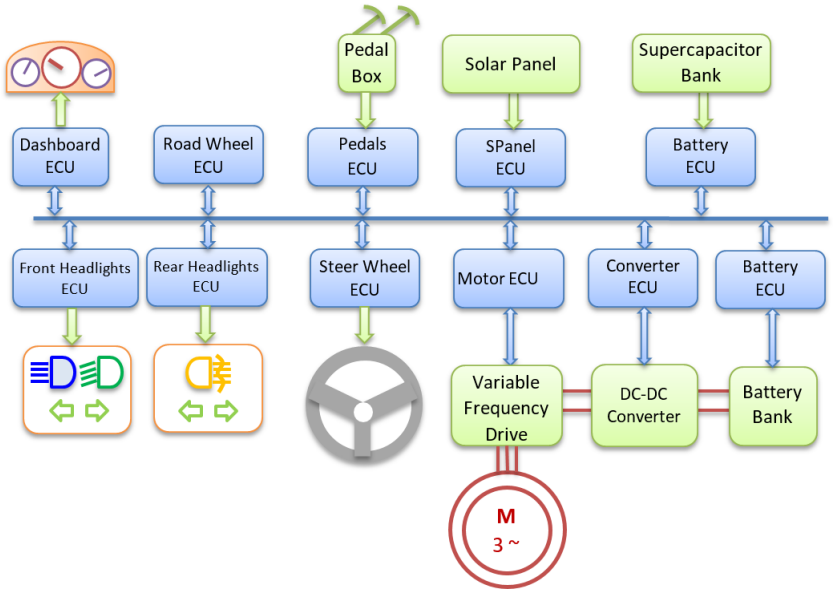

Figure A.3: Power and communication network in VEIL.

Appendix B

Other Simulation Results

B.1 First Results With Poisson Model

Simulation using simple model, BER = 2.6· and 10^{-7}, 10 million EC's, LEC = 5 ms, 10 simulation runs each.

Table B.1: Error recovery ratio for *PSA* benchmark - C_S/C_{MAX} varying from 1 to 10.

C_S/C_{MAX}	Unrecovered Errors	Message Error Recovery Ratio
1	46.5	95.987%
2	0.9	99.923%
3 ... 10	0.0	100.000%

Table B.2: Error recovery ratio for *VEIL* benchmark - C_S/C_{MAX} varying from 1 to 10.

C_S/C_{MAX}	Unrecovered Errors	Message Error Recovery Ratio
1	11.1	98.028%
2	0.1	99.983%
3...10	0.0	100.000%

Choosing possible values for T_S for the PSA and VEIL benchmarks - Tables B.3 and B.4.

Table B.3: Message recovery ratio with different (C_S, T_S) combinations for *PSA* benchmark

	T_S/LEC			
C_S	125	250	375	750
1	99.301%	98.829%	98.216%	95.987%
2	**100.000**%	**100.000**%	99.970%	99.923%
3 ... 10	100.000%	100.000%	**100.000**%	**100.000**%
Server BW	0.0432%	0.0216%	0.0216%	0.0108%

Simulations to obtain error recovery ratio as a function of the server capacity with higher than Aggressive environment. Performed 10 runs with 10 Mcycles each, $LEC = 5ms$, $T_S/LEC = 750$ and $\lambda = 2.6$ faults/second - results in tables B.5 and B.6

Table B.4: Message recovery ratio with different (C_S, T_S) combinations for *VEIL* benchmark

	T_S/LEC			
C_S	125	250	375	750
1	99.940%	99.820%	99.407%	98.028%
2	**100.000**%	**100.000**%	**100.000**%	99.983%
3 ... 10	100.000%	100.000%	100.000%	**100.000**%
Server BW	0.0432%	0.0216%	0.0216%	0.0108%

Table B.5: *PSA* benchmark - Error recovery ratio as a function of the server capacity.

C_S	Missed Deadlines	Message Error Recovery Ratio
1	3939.4	66.402%
2	991.0	91.458%
3	199.8	98.296%
4	31.6	99.731%
5	4.2	99.964%
6	2.2	99.981%
7...10	1.6	99.986%

Table B.6: *VEIL* benchmark - Error recovery ratio as a function of the server capacity.

C_S	Missed Deadlines	Message Error Recovery Ratio
1	1072.4	81.274%
2	141.1	97.537%
3	16.4	99.714%
4	2.1	99.965%
5	0.9	99.985%
6...10	0.7	99.987%

B.1.1 Polling Server

Simulations results using a Polling Server. Simulatios performed with bit rate = 1000 kbps, BER = $2.6 \cdot 10^{-7}$, 10 million cycles, LEC = 5ms.

Table B.7: Polling server T_S choosing, with *PSA* benchmark.

T_S/LEC	Missed Deadlines	Message Error Recovery Ratio
1	0	100.0%
2	502	55.2%
5	594	47.0%
10	938	16.3%
100	1114	0.6%

Table B.8: Polling server T_S choosing, with *VEIL* benchmark.

T_S/LEC	Missed Deadlines	Message Error Recovery Ratio
1	0	100.0%
2	315	44.7%
5	335	41.2%
10	471	17.4%
100	561	1.6%

B.1.2 Sporadic Server

Simulations results using a Sporadic Server. Simulations with bit rate = 1000 kbps, $T_S/LEC = 750$, BER = $2.6 \cdot 10^{-6}$ and 10 million cycles, LEC = 5ms - 10 simulation runs, each C_S.

Table B.9: Sporadic Server, *PSA* benchmark - Error recovery ratio as a function of the server capacity.

C_S	Missed Deadlines	Message Error Recovery Ratio
1	7934.4	32.479%
2	4392.2	62.622%
3	752.7	93.594%
4	157.4	98.660%
5	21.2	99.820%
6	7.1	99.940%
7 ... 10	1.3	99.989%

Table B.10: Sporadic Server, *VEIL* benchmark - Error recovery ratio as a function of the server capacity.

C_S	Missed Deadlines	Message Error Recovery Ratio
1	3548.9	37.923%
2	1277.9	77.647%
3	32.8	99.426%
4	4.2	99.926%
5	1.7	99.971%
6 ... 10	0.6	99.989%

B.2 Simulation Results for Priority Assignment of Deferrable Server

The simulations were performed with the following parameters Simulations with bit rate = 1000 kbps, $T_S/LEC = 750$, $\lambda = 2.6$ faults/sec and LSW=10.5% for *PSA* benchmark and $\lambda = 8$ faults/sec and LSW=6.3% for *VEIL* benchmark, 10 million EC's and LEC = 5ms - 10 simulation runs.

Table B.11: Deadline misses in message hit by errors of *PSA* benchmark (average)

	MaxPriority	*SamePriority*	*SameDMP*	*ServerEDF*
Total misses	5.9	10.1	4.9	4.0
Unrecoverable misses	4.8	4.9	3.5	3.6

Table B.12: Deadline misses in message hit by errors of *VEIL* benchmark (average).

	MaxPriority	*SamePriority*	*SameDMP*	*ServerEDF*
Total misses	12.3	17.0	11.0	11.0
Unrecoverable misses	12.0	13.0	11.0	11.0

Server average response time, test for all server policies are presented in tables B.15 and B.16.

Table B.13: *PSA* benchmark - WCRT for all tested policies.

		Direct				Indirect			
msg	no error	Max Priority	Same Priority	Same DMP	Server EDF	Max Priority	Same Priority	Same DMP	Server EDF
1	1	2	2	2	2	1	1	1	1
2	1	2	2	2	2	1	1	1	1
3	1	2	2	2	2	1	1	1	1
4	1	2	2	2	2	2	1	1	2
5	1	2	2	2	2	2	1	1	2
6	2	4	3	3	3	2	2	2	2
7	2	3	3	3	3	2	2	2	2
8	2	3	4	4	3	2	2	2	2
9	2	3	4	4	3	4	2	2	4
10	2	3	4	4	4	4	4	4	4
11	4	5	6	6	6	4	4	4	4
12	4	5	8	8	6	4	4	4	4
13	4	5	8	8	6	6	6	6	6
14	6	7	8	8	8	8	8	8	8
15	8	9	10	10	9	8	8	8	8
16	8	9	15	15	11	8	8	8	8
17	8	9	16	16	12	16	16	16	16
18	10	11	16	16	12	18	18	18	18
19	16	17	18	18	18	18	18	20	18
20	18	19	20	20	19	20	20	20	20
21	18	19	20	20	20	20	20	20	20
22	18	19	20	20	20	20	20	20	20
23	18	19	20	20	20	20	20	20	20

Table B.14: *VEIL* benchmark - WCRT for all tested policies.

		Direct				Indirect			
msg	no error	Max Priority	Same Priority	Same DMP	Server EDF	Max Priority	Same Priority	Same DMP	Server EDF
1	1	2	2	2	2	1	1	1	1
2	1	2	2	2	2	2	1	1	2
3	1	2	2	2	2	2	1	1	2
4	1	3	3	3	3	2	1	1	1
5	2	4	4	4	4	4	2	2	2
6	2	4	4	4	4	4	4	4	4
7	2	5	6	6	6	4	4	4	4
8	4	6	8	8	8	4	4	4	4
9	4	5	8	8	8	6	6	6	6
10	4	7	8	8	8	8	8	8	8
11	6	9	10	10	10	10	8	8	8
12	7	9	8	8	10	12	10	10	10
13	8	9	10	12	12	12	10	10	10
14	8	11	12	12	12	14	12	12	12
15	8	11	12	12	11	14	12	12	12
16	8	9	12	12	10	15	14	14	14
17	10	11	12	12	12	16	16	16	16
18	12	15	16	16	14	16	16	16	16
19	14	16	16	16	18	16	16	16	18

Table B.15: *PSA* benchmark - Server average response time

Policy/Message	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
MaxPriority	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
SamePriority	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.31	2.00	2.00	2.38	2.49
SameDMP	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.31	2.00	2.00	2.38	2.49
ServerEDF	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	2.00	2.00	2.01
Policy/Message	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	
MaxPriority	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
SamePriority	2.65	2.52	2.49	2.43	4.35	4.40	5.39	3.82	2.26	2.21	2.67	
SameDMP	2.65	2.52	2.49	2.43	4.35	4.42	5.39	3.82	2.26	2.21	2.67	
ServerEDF	2.00	2.01	1.91	1.58	2.78	2.61	2.70	2.13	2.26	2.22	2.67	

Table B.16: *VEIL* benchmark - Server average response time

Policy/Message	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
MaxPriority	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
SamePriority	1.00	1.00	1.00	1.00	2.00	2.00	2.00	2.01	2.01	2.00
SameDMP	1.00	1.00	1.00	1.00	2.00	2.00	2.00	2.01	2.02	2.01
ServerEDF	1.00	1.00	1.00	1.00	1.00	1.00	2.00	2.01	2.02	2.01
Policy/Message	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	
MaxPriority	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	
SamePriority	2.00	1.53	2.00	2.00	2.00	2.00	2.00	2.00	2.00	
SameDMP	2.00	1.53	2.02	2.00	2.00	2.00	2.00	2.00	2.00	
ServerEDF	2.00	1.53	2.02	2.00	1.97	2.00	2.00	2.00	2.00	

B.3 Recovery Method with Multiple copy retransmission

Table B.17 with *Updated_SAE* benchmark, LSW = 55.1% and and *RepLevel*={4, 3, 2, 1}, the messages 31, 32 and 33 have their WCRT with Indirect Interference.

Table B.18 presents values for *PSA* benchmark, with Aggressive environment, LSW = 28% and *RepLevel*={3, 3, 2, 1}.

Table B.19 is for the *VEIL* benchmark, with Aggressive environment, *RepLevel*={3, 2, 2 , 1} and LSW = 23.8%

Table B.17: *Updated_SAE* benchmark - Response time with all interference patterns and *RepLevel*={4, 3, 2, 1}.

	Indirect Interference								Direct Interference								
msg	4-4-4-4	4-4-6-0	4-6-4-0	4-6-0-0	6-4-4-0	6-6-0-0	6-4-0-0	4-0-0-0	4-4-4-0	4-6-0-0	6-4-0-0	6-0-0-0	IND	DIR	WCRT	no error	Deadline
1	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
2	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
3	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
4	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
5	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
6	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
7	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
8	1	1	1	1	1	2	1	1	2	2	2	2	2	2	2	1	2
9	1	1	1	1	2	2	2	1	2	2	3	3	2	3	3	1	3
10	1	1	1	1	2	2	2	1	2	2	3	3	2	3	3	1	3
11	1	1	1	2	2	2	2	1	2	2	3	3	2	3	3	1	3
12	2	2	2	2	2	2	2	2	3	3	3	3	2	3	3	1	3
13	2	2	2	2	2	2	2	2	3	3	3	3	2	3	3	1	3
14	2	2	2	2	2	2	2	2	3	3	3	3	2	3	3	1	3
15	2	2	2	2	2	2	2	2	3	3	3	3	2	3	3	1	3
16	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	1	3
17	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	1	4
18	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	2	4
19	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	4
20	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	4
21	3	3	3	3	3	3	3	2	4	4	4	3	3	4	4	2	4
22	3	4	4	3	3	3	3	2	4	4	4	3	4	4	4	2	4
23	3	4	4	3	4	3	3	2	4	4	4	3	4	4	4	2	5
24	3	4	4	3	4	3	3	2	4	4	4	3	4	4	4	2	5
25	3	4	4	3	4	3	3	2	4	4	4	4	4	4	4	2	5
26	4	4	4	3	4	4	3	2	5	4	4	4	4	5	5	2	5
27	4	4	4	3	4	4	3	2	5	4	4	4	4	5	5	2	5
28	5	4	4	4	4	4	3	3	5	5	4	4	5	5	5	2	5
29	5	4	4	4	4	4	4	3	5	5	5	4	5	5	5	2	5
30	5	4	4	4	4	4	4	3	5	5	5	4	5	5	5	2	20
31	6	5	5	4	5	4	4	3	5	5	5	4	6	5	6	2	40
32	6	5	5	4	5	4	4	3	5	5	5	4	6	5	6	2	40
33	6	5	5	4	5	4	4	3	5	5	5	4	6	5	6	2	40
34	6	6	6	4	5	5	4	3	6	5	5	4	6	6	6	3	400
35	6	6	6	4	6	5	4	3	6	5	5	5	6	6	6	3	400
36	6	6	6	4	6	5	4	3	6	5	5	5	6	6	6	3	400

Table B.18: *PSA* benchmark - Response time with all interference patterns .

	Indirect Interference								Direct Interference								
msg	3-3-3-3	3-3-6-0	3-6-3-0	3-6-0-0	6-3-3-0	6-6-0-0	6-3-0-0	4-0-0-0	3-3-3-0	3-6-0-0	6-3-0-0	6-0-0-0	IND	DIR	WCRT	no error	Deadline
1	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
2	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
3	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
4	1	1	1	1	1	2	1	1	2	2	2	2	2	2	2	1	2
5	1	1	1	1	2	2	2	1	2	2	3	3	2	3	3	1	3
6	1	1	1	1	2	2	2	1	2	2	3	3	2	3	3	1	3
7	1	1	1	1	2	2	2	1	2	2	3	3	2	3	3	1	4
8	1	1	1	1	2	2	2	2	2	2	3	3	2	3	3	1	4
9	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	1	4
10	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	1	8
11	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	1	10
12	2	2	2	2	2	3	2	2	3	3	3	3	3	3	3	1	10
13	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	10
14	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	10
15	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	10
16	2	3	3	3	3	3	3	2	3	4	4	3	3	4	4	2	20
17	3	4	4	3	4	4	3	2	4	4	4	4	4	4	4	2	20
18	3	4	4	3	4	4	3	2	4	4	4	4	4	4	4	2	20
19	3	4	4	3	4	4	3	3	4	4	4	4	4	4	4	2	20
20	4	4	4	4	4	4	4	3	5	5	5	4	4	5	5	2	20
21	4	4	4	4	4	4	4	3	5	5	5	4	4	5	5	2	30
22	4	4	4	4	4	4	4	3	5	5	5	4	4	5	5	2	30
23	4	4	4	4	4	4	4	3	5	5	5	4	4	5	5	2	40

Table B.19: *VEIL* benchmark - response time with all interference patterns.

	Indirect Interference								Direct Interference								
msg	**3-3-3-3**	**3-3-4-0**	**3-4-3-0**	**3-6-0-0**	**4-3-3-0**	**4-4-0-0**	**6-3-0-0**	**4-0-0-0**	**3-3-3-0**	**3-4-0-0**	**4-3-0-0**	**6-0-0-0**	**IND**	**DIR**	**WCRT**	**no error**	**Deadline**
1	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
2	1	1	1	1	1	1	1	1	2	2	2	2	1	2	2	1	2
3	1	1	1	1	1	1	2	1	2	2	2	2	2	2	2	1	2
4	1	1	1	1	1	1	2	1	2	2	2	3	2	3	3	1	4
5	1	1	1	1	1	1	2	1	2	2	2	3	2	3	3	1	4
6	1	1	1	1	1	2	2	1	2	2	2	3	2	3	3	1	4
7	1	1	1	1	2	2	2	2	2	2	3	3	2	3	3	1	10
8	2	2	2	2	2	2	2	2	3	3	3	3	2	3	3	1	10
9	2	2	2	2	2	2	2	2	3	3	3	3	2	3	3	1	20
10	2	2	2	3	2	2	2	2	3	3	3	3	3	3	3	1	20
11	2	2	2	3	2	3	3	2	3	3	3	3	3	3	3	1	20
12	2	2	2	3	2	3	3	2	3	3	3	3	3	3	3	2	50
13	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	100
14	2	2	3	3	3	3	3	2	3	4	4	3	3	4	4	2	100
15	3	3	3	3	3	3	3	2	4	4	4	3	3	4	4	2	200
16	3	3	3	3	3	3	3	2	4	4	4	4	3	4	4	2	200
17	3	3	3	3	3	3	3	2	4	4	4	4	4	4	4	2	200
18	3	4	4	3	4	3	3	2	4	4	4	4	4	4	4	2	200
19	4	4	4	4	4	3	4	3	5	4	4	4	4	5	5	2	200

Table B.20: Comparing analytic WCRT with the one observed in simulations for the *PSA* message set with LSW = 28.0% of LEC, considering an Aggressive environment.

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Design	2	2	2	2	3	3	3	3	3	3	3	3
Simulation	2	2	2	2	2	2	2	2	2	2	2	2
msg	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	
Design	4	4	4	4	4	4	4	5	5	5	5	
Simulation	3	3	3	3	3	3	3	4	3	4	4	

Table B.21: Comparing analytic WCRT with the one observed in simulations for the *VEIL* message set with LSW = 23.8% of LEC, considering an Aggressive environment.

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Design	2	2	2	3	3	3	3	3	3	3
Simulation	2	2	2	2	2	2	2	2	2	2
msg	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	
Design	3	3	4	4	4	4	4	4	5	
Simulation	2	2	2	3	3	3	3	3	3	

B.3.1 *Controlled Retransmission* vs *Automatic Retransmission* - Average and WCRT

Table B.22: Comparing average response time of for the *PSA* benchmark with LSW = 28.0% of LEC, considering an Aggressive environment - *Controlled Retransmission* vs *Automatic Retransmission*

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Automatic	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.3333	1.3333	2.0000	1.5000	1.6667
Controlled	1.0002	1.0002	1.0002	1.0003	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002
msg	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	
Automatic	2.0000	2.0833	2.3333	2.8333	3.3333	4.0000	4.0000	4.0000	3.0000	3.5000	4.6667	
Controlled	1.0003	1.0836	1.3337	1.6669	2.0003	2.0003	2.0003	2.0003	1.5002	1.5002	2.0002	

Table B.23: Comparing WCRT of the *PSA* benchmark with LSW = 28.0% of LEC, considering an Aggressive environment - *Controlled Retransmission* vs *Automatic Retransmission*

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Automatic	1	1	1	1	1	1	1	2	2	2	2	2
Controlled	2	2	2	2	2	3	3	2	2	3	3	2
msg	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	
Automatic	2	3	3	3	4	4	4	4	4	6	6	
Controlled	2	3	3	3	4	4	3	4	3	3	3	

Table B.24: Comparing Average RT of the *VEIL* benchmark with LSW = 23.8% of LEC, considering an Aggressive environment - *Controlled Retransmission* vs *Automatic Retransmission*

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Automatic	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.5000	2.0000	2.0000
Controlled	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0002	1.0003
msg	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	
Automatic	2.0000	1.5000	3.0000	3.0000	3.0000	3.0000	4.0000	4.0000	4.0000	
Controlled	1.0003	1.0002	1.0002	2.0002	2.0001	2.0002	2.0002	2.0004	2.0004	

Table B.25: Comparing WCRT of the *VEIL* benchmark with LSW = 23.8% of LEC, considering an Aggressive environment - *Controlled Retransmission* vs *Automatic Retransmission*

msg	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Automatic	1	1	1	1	1	1	1	2	2	2
Controlled	2	2	2	2	2	2	2	2	2	2

msg	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
Automatic	2	2	3	3	3	3	4	4	4
Controlled	3	2	2	3	3	3	3	3	3

Appendix C

Resolving IMO Scenarios in the Master Node

The CAN controller uses different rules for the transmitter and the receivers in error detection of the last bits of the CAN frame, namely in the EOF field, that may lead to inconsistent scenarios [RVA+98].

The proposed method relies on the fact that the Master node shares the same view of every message transmission state with errors or error free - with all the nodes. As previously stated this assumption may be violated in very particular cases that need to be thoroughly analyzed. The generic scenario is depicted in Figure C.1.

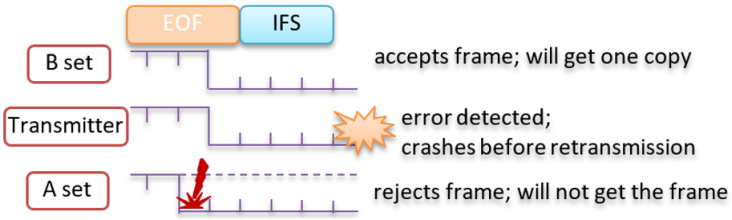

Figure C.1: Inconsistent Message Omission scenario.

When the synchronous messages are transmitted the Master is a receiver and we assume that he has a correct and consistent message state view with all the nodes, if the error happens previous to the last but one bit in the EOF field. If the error is on the last but one bit, then one or more nodes (set A) may detect a dominant value in this bit, so they reject the message and transmit an error flag. The other receiving nodes (set B) view the error message as an overload flag and accept the message. The transmitter detects a dominant signal on the last bit and transmits an error flag, but as the automatic retransmission is not active, the message is not queued for retransmission by the transmitter, so some of the nodes (set A) do not get the message and other nodes (set B) will get it. This scenario is defined as an Inconsistent Message Omission (IMO) [RVA+98] and constitutes a serious impairment to obtain a high-reliability transmission system. From the Master point of view, three different scenarios may arise:

1. The Master node has the same view as the nodes in set A - in this case the message is detected by the Master node as faulty and is recovered by the mechanism described in this thesis and the nodes in set B will receive a second copy of the message, being this an Inconsistent Message Duplicate - IMD [RVA+98] . This inconsistency is not critic, since these nodes always get the message and can choose the replica to use;

2. The Master node has the same view as the nodes in set B - the Master accepts the message and considers it correct, so no further action is taken (regarding this message). This will effectively lead to an IMO and the nodes in set A will not get the message. This is a serious impairment to obtain the intended reliability, since one IMO per hour is expected to happen in each mission of one hour (numbers recalculated based on the values present in [RVA+98]), with a bus running at 1 Mbps and subject to an Aggressive environment with BER equal to 2.6· and 10^{-7};

3. Only the Master has the view presented for set A and all the other nodes behavior is the same one as the set B nodes, so all the nodes will get the message. As the Master considers the message in error, it will be retriggered for retransmission, so all the nodes will get a duplicate of it in the next EC (we have again an IMD). This situation is similar to the first scenario, but in this case the retransmission would not be necessary, since all the nodes already possess the message.

From the presented inconsistent scenarios, only the number two must be resolved if a highly reliable transmission system is sought. This can be done in the following way, as illustrated in Figure C.2: looking at the CAN specification [BG91] we know that if an error occurred in the last but one bit, of at least one receiver, then an error frame is sent and then from the last bit of the EOF field position at least six more bits (due to the sent error flag) should have a dominant value. Then the Master must read individually these bits in order to decide if an error occurred or not. If one of these bits have dominant value then the Master considers that an error occurred and the message should be put in the recovery queue. In this way we transform a possible IMO in an IMD, which does not compromise the attainment of a high-reliability system.

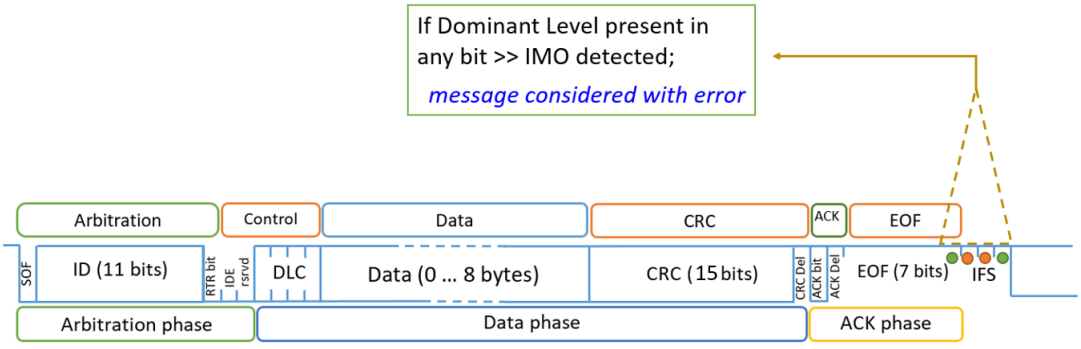

Figure C.2: Proposed solution.

Nevertheless, the proposed solution can fail if an error occur in all the 4 last bits of the frame. In this case the Master reads recessive values when in the bus a dominant level is present. In this case the Master will consider the message as correct and some nodes may have discard it (similar to scenario 2). The probability of occurrence of this inconsistent scenario in an Aggressive environment is, for the Master and considering single bit errors, equal to (2.6· and 10^{-7})^4 which leads to a value less than 10^{-9} IMOs per hour (an acceptable value even for safety critical systems). This solution needs a specific implementation in hardware, e.g. FPGA, for the Master node only, in order to be able access individual bits of the CAN frame (namely the last four ones, LB + 3 of the IFS) and to perform the necessary checking actions of these bits, as previously described.

Appendix D

Acronyms List

GENERIC

ADAS	Advanced Driver Assistance Systems
AFDX	Avionics Full-DupleX Switched Ethernet
AVB	Audio/Video Bridging
BACnet	Building Automation Control net
BER	Bit-Error Rate
CAN	Controller Area Network
CA	Collision Avoidance
CLP	Constraint Logic Programming
COTS	Commercial Off-the-Shelf
CPS	Cyber-Physical Systems
CR	Collision Resolution
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
DES	Distributed Embedded Systems
DM	Deadline-Monotonic
DS	Deferrable Server
DS	Dynamic Segment
ECU	Electronic Control Unit
EDF	Earliest-Deadline First
EMC	Electromagnetic Compatibility
EMI	Electro-magnetic Interference
ES	Embedded Systems
FTDMA	Flexible TDMA

FTT	Flexible Time-Triggered
HaRTES	Hard Real-Time Ethernet Switching
LLF	Least-Laxity First
IoT	Internet of Things
IP	Internet Protocol
ITS	Intelligent Transportation System
ISO	International Organization for Standardization
LVDS	Low-Voltage Differential Signalling
LIN	Loca Interconnect Network
MILP	Mixed Integer Linear Programming
MAC	Medium Access Control
MOST	Media Oriented Systems Transport
MS	minislot
MT	macrotick
NES	Networked Embedded System
NIT	Network Idle Time
OBD	On-Board Diagnostics
OEM	Original Equipment Manufacturer
PDU	Protocol Data Unit
QoS	Quality-of-Service
RF	Radio Frequency
RM	Rate-Monotonic
RTS	Real-Time Systems
SAE	Society of Automotive Engineers
SS	Sporadic Server
SS	Static Segment
STP	Shielded TP (cable)
TCAN	Timely CAN
TDMA	Time Division Multiple Access
TP	Twisted Pair (cable)
TT	Time-Triggered
TTA	Time-Triggered Architecture
TTCAN	Time-Triggered CAN

TTP	Time-Triggered Protocol
TTP/A	Time-Triggered Protocol, SAE Class A
TTP/C	Time-Triggered Protocol, SAE Class C
UTP	Unshielded TP (cable)
UTSP	Unshielded Twitsed Single Pair
VAN	Vehicle Area Network
VANET	Vehicle Ad-hoc Network
WCRT	Worst Case Response Time
WorldFIP	Factory Instrumentation Protocol
WSN	Wireless Sensor Networks

FTT-related

AM	Asynchronous Message
AMS	Asynchronous Messaging System
ART	Asynchronous Requirements Table
AW	Asynchronous Window
EC	Elementary Cycle
FTT-SE	Flexible Time-Triggered communication over Switched Ethernet
LAW	Length of Asynchronous Window
LEC	Length of Elementary Cycle
LSW	Length of Synchronous Window
LTM	Length of Trigger Message
SM	Synchronous Message
SMS	Synchronous Messaging System
SRDB	System Requirements DataBase
SRT	Synchronous Requirements Table
SW	Synchronous Window
TM	Trigger Message