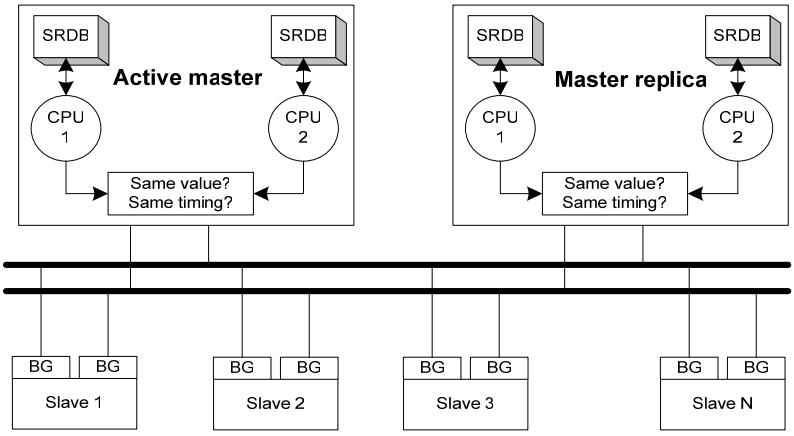
Universidade de Aveiro Departamento de Electrónica e Telecomunicações
2005

Joaquim José de Castro Ferreira

Tolerância a Falhas em Sistemas de Comunicação de Tempo-Real Flexíveis

Fault-Tolerance in Flexible Real-Time Communication Systems




Universidade de Aveiro Departamento de Electrónica e Telecomunicações
2005

Joaquim José de Castro Ferreira

Tolerância a Falhas em Sistemas de Comunicação de Tempo-Real Flexíveis

Fault-Tolerance in Flexible Real-Time Communication Systems

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Engenharia Informática, realizada sob a orientação científica de Luís Miguel Pinho de Almeida, Professor Auxiliar do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro e co-orientação de José Alberto Gouveia Fonseca, Professor Associado do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro.

Dissertation submitted to the University of Aveiro in the fulfilment of the requirements for the degree of Doutor em Engenharia Informática, under the supervision of Luís Miguel Pinho de Almeida, Professor Auxiliar at the Departamento de Electrónica e Telecomunicações of the University of Aveiro and co-supervision of José Alberto Gouveia Fonseca, Professor Associado at the Departamento de Electrónica e Telecomunicações of the University of Aveiro.

O júri / The Jury

Presidente / President	Prof. Doutor António Francisco Carrelhas Cachapuz Professor Catedrático da Universidade de Aveiro
Vogais / Examiners committee	Prof. Doutor Luís Miguel Pinho de Almeida Professor Auxiliar da Universidade de Aveiro (orientador)
	Prof. Doutor José Alberto Gouveia Fonseca Professor Associado da Universidade de Aveiro (co-orientador)
	Prof. Doutor Paulo Jorge Esteves Veríssimo Professor Catedrático da Faculdade de Ciências da Universidade de Lisboa
	Prof. Doutor Juan Ricardo Pimentel-Flores Full Professor da Universidade de Kettering, Estados Unidos da América
	Prof. Doutor Francisco Manuel Madureira e Castro Vasques de Carvalho Professor Associado da Faculdade de Engenharia da Universidade do Porto
	Prof. Doutor Ernesto Fernando Ventura Martins Professor Auxiliar da Universidade de Aveiro

agradecimentos

O trabalho realizado no âmbito desta dissertação contou com a colaboração, directa e indirecta, de diversas pessoas. Quero aqui e a todas elas expressar o meu sincero agradecimento. Contudo e devido ao seu especial envolvimento, gostaria de particularizar os seguintes agradecimentos.

A Luís Miguel Pinho de Almeida, Professor da Universidade de Aveiro e meu orientador, a quem expresso o mais profundo reconhecimento pelo empenho, disponibilidade, amizade e estímulo demonstrado na orientação deste trabalho, numa primeira fase na qualidade de co-orientador e posteriormente como orientador científico principal.
A José Alberto Fonseca, Professor da Universidade de Aveiro e meu co-orientador, por me ter incentivado a enfrentar este desafio e, principalmente, pelo modo empenhado, amigo e disponível, com que assumiu a supervisão deste trabalho, quer como orientador científico principal numa primeira fase dos trabalhos, quer mais tarde como co-orientador.
A Paulo Pedreiras, não só pelas enriquecedoras e profícuas discussões técnicas e científicas, mas principalmente pela amizade demonstrada ao longo destes anos.
A Ernesto Martins, da Universidade de Aveiro, pelas mais valias, pessoais e científicas, resultantes da sua colaboração neste trabalho.
A Arnaldo Oliveira, da Universidade de Aveiro, pela amizade, disponibilidade e valiosa colaboração em algumas etapas deste trabalho.
A Pedro Fonseca, da Universidade de Aveiro, pela amizade demonstrada ao longo destes anos e pela colaboração em algumas etapas deste trabalho.
A Guillermo Rodríguez-Navas, Julían Proenza, Juan Rigo do Departamento de Matemática e Informática da Universidade das Ilhas Baleares, pela amabilidade e amizade com que me receberam na sua instituição e pela colaboração neste trabalho.
Ao Engenheiro Rui Matos da empresa J. R. Matos, SA, por ter facultado o acesso à sua empresa para a realização de alguns testes.
A todos os meus colegas e amigos, dentro e fora do Departamento de Engenharia Informática e de Tecnologias da Informação da ESTCB, do Laboratório de Sistemas Electrónicos do IEETA no seio do qual desenvolvi este trabalho e do Departamento de Electrónica e Telecomunicações da UA, pela amizade, companheirismo e encorajamento manifestado ao longo destes anos e que assim me ajudaram a desenvolver esta tese. Gostaria particularmente de agradecer àqueles com quem diariamente convivi: Francisco Borges, José Vieira, Tullio Facchinetti, Manuel Barranco, Mário Calha, Valter Silva, Frederico Santos e Ricardo Marau.
E, acima de tudo, à Natália, pelo encorajamento e sacrifícios suportados ao longo destes anos e à Inês, ao Nuno e ao Miguel por não terem desistido de perguntar se o livro já estava escrito.

resumo

Nas últimas décadas, os sistemas embutidos distribuídos, têm sido usados em variados domínios de aplicação, desde o controlo de processos industriais até ao controlo de aviões e automóveis, sendo expectável que esta tendência se mantenha e até se intensifique durante os próximos anos.

Os requisitos de confiabilidade de algumas destas aplicações são extremamente importantes, visto que o não cumprimento de serviços de uma forma previsível e pontual pode causar graves danos económicos ou até pôr em risco vidas humanas.

A adopção das melhores práticas de projecto no desenvolvimento destes sistemas não elimina, por si só, a ocorrência de falhas causadas pelo comportamento não determinístico do ambiente onde o sistema embutido distribuído operará. Desta forma, é necessário incluir mecanismos de tolerância a falhas que impeçam que eventuais falhas possam comprometer todo o sistema.

Contudo, para serem eficazes, os mecanismos de tolerância a falhas necessitam ter conhecimento *a priori* do comportamento correcto do sistema de modo a poderem ser capazes de distinguir os modos correctos de funcionamento dos incorrectos.

Tradicionalmente, quando se projectam mecanismos de tolerância a falhas, o conhecimento *a priori* significa que todos os possíveis modos de funcionamento são conhecidos na fase de projecto, não os podendo adaptar nem fazer evoluir durante a operação do sistema. Como consequência, os sistemas projectados de acordo com este princípio ou são completamente estáticos ou permitem apenas um pequeno número de modos de operação.

Contudo, é desejável que os sistemas disponham de alguma flexibilidade de modo a suportarem a evolução dos requisitos durante a fase de operação, simplificar a manutenção e reparação, bem como melhorar a eficiência usando apenas os recursos do sistema que são efectivamente necessários em cada instante. Além disto, esta eficiência pode ter um impacto positivo no custo do sistema, em virtude deste poder disponibilizar mais funcionalidades com o mesmo custo ou a mesma funcionalidade a um menor custo.

Porém, flexibilidade e confiabilidade têm sido encarados como conceitos conflituais.

Isto deve-se ao facto de flexibilidade implicar a capacidade de permitir a evolução dos requisitos que, por sua vez, podem levar a cenários de operação imprevisíveis e possivelmente inseguros. Desta fora, é comummente aceite que apenas um sistema completamente estático pode ser tornado confiável, o que significa que todos os aspectos operacionais têm de ser completamente definidos durante a fase de projecto.

Num sentido lato, esta constatação é verdadeira. Contudo, se os modos como o sistema se adapta a requisitos evolutivos puderem ser restringidos e controlados, então talvez seja possível garantir a confiabilidade permanente apesar das alterações aos requisitos durante a fase de operação.

A tese suportada por esta dissertação defende que é possível flexibilizar um sistema, dentro de limites bem definidos, sem comprometer a sua confiabilidade e propõe alguns mecanismos que permitem a construção de sistemas de segurança crítica baseados no protocolo *Controller Area Network* (CAN). Mais concretamente, o foco principal deste trabalho incide sobre o

protocolo *Flexible Time-Triggered* CAN (FTT-CAN), que foi especialmente desenvolvido para disponibilizar um grande nível de flexibilidade operacional combinando, não só as vantagens dos paradigmas de transmissão de mensagens baseados em eventos e em tempo, mas também a flexibilidade associada ao escalonamento dinâmico do tráfego cuja transmissão é despoletada apenas pela evolução do tempo.

Este facto condiciona e torna mais complexo o desenvolvimento de mecanismos de tolerância a falhas para FTT-CAN do que para outros protocolos como por exemplo, TTCAN ou FlexRay, nos quais existe um conhecimento estático, antecipado e comum a todos os nodos, do escalonamento de mensagens cuja transmissão é despoletada pela evolução do tempo.

Contudo, e apesar desta complexidade adicional, este trabalho demonstra que é possível construir mecanismos de tolerância a falhas para FTT-CAN preservando a sua flexibilidade operacional.

É também defendido nesta dissertação que um sistema baseado no protocolo FTT-CAN e equipado com os mecanismos de tolerância a falhas propostos é passível de ser usado em aplicações de segurança crítica.

Esta afirmação é suportada, no âmbito do protocolo FTT-CAN, através da definição de uma arquitectura tolerante a falhas integrando nodos com modos de falha tipo falha-silêncio e nodos mestre replicados.

Os vários problemas resultantes da replicação dos nodos mestre são, também eles, analisados e várias soluções são propostas para os obviar. Concretamente, é proposto um protocolo que garante a consistência das estruturas de dados replicadas a quando da sua actualização e um outro protocolo que permite a transferência dessas estruturas de dados para um nodo mestre que se encontre não sincronizado com os restantes depois de inicializado ou reinicializado de modo assíncrono.

Além disto, esta dissertação também discute o projecto de nodos FTT-CAN que exibam um modo de falha do tipo falha-silêncio e propõe duas soluções baseadas em componentes de hardware localizados no interface de rede de cada nodo, para resolver este problema. Uma das soluções propostas baseia-se em *bus guardians* que permitem a imposição de comportamento falha-silêncio nos nodos escravos e suportam o escalonamento dinâmico de tráfego na rede. A outra solução baseia-se num interface de rede que arbitra o acesso de dois microprocessadores ao barramento. Este interface permite que a replicação interna de um nodo seja efectuada de forma transparente e assegura um comportamento falha-silêncio quer no domínio temporal quer no domínio do valor ao permitir transmissões do nodo apenas quando ambas as réplicas coincidam no conteúdo das mensagens e nos instantes de transmissão. Esta última solução está mais adaptada para ser usada nos nodos mestre, contudo também poderá ser usada nos nodos escravo, sempre que tal se revele fundamental.

abstract

Distributed embedded systems (DES) have been widely used in the last few decades in several application fields, ranging from industrial process control to avionics and automotive systems. In fact, it is expectable that this trend will continue over the years to come.

In some of these application domains 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.

The adoption of the best practices in the design of distributed embedded systems does not fully avoid the occurrence of faults, arising from the non-deterministic behavior of the environment where each particular DES operates. Thus, fault-tolerance mechanisms need to be included in the DES to prevent possible faults leading to system failure.

To be effective, fault-tolerance mechanisms require an *a priori* knowledge of the correct system behavior to be capable of distinguishing them from the erroneous ones.

Traditionally, when designing fault-tolerance mechanisms, the *a priori* knowledge means that all possible operational modes are known at system design time and cannot adapt nor evolve during runtime. As a consequence, systems designed according to this principle are either fully static or allow a small number of operational modes only. Flexibility, however, is a desired property in a system in order to support evolving requirements, simplify maintenance and repair, and improve the efficiency in using system resources by using only the resources that are effectively required at each instant. This efficiency might impact positively on the system cost because with the same resources one can add more functionality or one can offer the same functionality with fewer resources.

However, flexibility and dependability are often regarded as conflicting concepts. This is so because flexibility implies the ability to deal with evolving requirements that, in turn, can lead to unpredictable and possibly unsafe operating scenarios. Therefore, it is commonly accepted that only a fully static system can be made dependable, meaning that all operating conditions are completely defined at pre-runtime.

In the broad sense and assuming unbounded flexibility this assessment is true, but if one restricts and controls the ways the system could adapt to evolving requirements, then it might be possible to enforce continuous dependability.

This thesis claims that it is possible to provide a bounded degree of flexibility without compromising dependability and proposes some mechanisms to build safety-critical systems based on the Controller Area Network (CAN).

In particular, the main focus of this work is the Flexible Time-Triggered CAN protocol (FTT-CAN), which was specifically developed to provide such high level of operational flexibility, not only combining the advantages of time- and event-triggered paradigms but also providing flexibility to the time-triggered traffic. This fact makes the development of fault-tolerant mechanisms more complex in FTT-CAN than in other protocols, such as TTCAN or FlexRay, in which there is *a priori* static common knowledge of the time-triggered message schedule shared by all nodes. Nevertheless, as it is demonstrated in this work, it is possible to build fault-tolerant mechanisms for FTT-CAN that preserve its

high level of operational flexibility, particularly concerning the time-triggered traffic. With such mechanisms it is argued that FTT-CAN is suitable for safety-critical applications, too.
This claim was validated in the scope of the FTT-CAN protocol by presenting a fault-tolerant system architecture with replicated masters and fail-silent nodes. The specific problems and mechanisms related with master replication, particularly a protocol to enforce consistency during updates of replicated data structures and another protocol to transfer these data structures to an unsynchronized node upon asynchronous startup or restart, are also addressed.
Moreover, this thesis also discusses the implementations of fail-silence in FTT-CAN nodes and proposes two solutions, both based on hardware components that are attached to the node network interface. One solution relies on bus guardians that allow enforcing fail-silence in the time domain. These bus guardians are adapted to support dynamic traffic scheduling and are fit for use in FTT-CAN slave nodes, only. The other solution relies on a special network interface, with duplicated microprocessor interface, that supports internal replication of the node, transparently. In this case, fail-silence can be assured both in the time and value domain since transmissions are carried out only if both internal nodes agree on the transmission instant and message contents. This solution is well adapted for use in the masters but it can also be used, if desired, in slave nodes.

apoios

Este trabalho foi apoiado pelas seguintes instituições:

Escola Superior de Tecnologia de Castelo Branco, que me dispensou de serviço docente durante três anos e que financiou a minha participação em várias conferências internacionais.

Ministério da Educação e ao FSE no âmbito do III Quadro Comunitário de Apoio, através do programa PRODEP III, que financiou parcialmente a minha dispensa de serviço docente, bem como a minha participação em várias conferências internacionais onde foram apresentados resultados parciais obtidos no âmbito desta tese.

Unidade de Investigação IEETA da Universidade de Aveiro, que apoiou financeiramente a minha participação em várias conferências internacionais para apresentação de resultados parciais obtidos no âmbito desta tese.

À Natália,
à Inês, ao Nuno e ao Miguel,
aos meus pais e irmão,
a todos os familiares e amigos.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The problem

Distributed embedded systems (DES) have been widely used in the last few decades in several application fields, ranging from industrial process control to avionics and automotive systems. In fact, it is expectable that this trend will continue over the years to come. In some of these application domains 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 [Kop97].

The adoption of the best practices in the design of distributed embedded systems does not fully avoid the occurrence of faults, arising from the non-deterministic behavior of the environment where each particular DES operates. Thus, fault-tolerance mechanisms need to be included in the DES to prevent possible faults leading to system failure. To be effective, fault-tolerance mechanisms require an *a priori* knowledge of the correct system behavior to be capable of distinguishing them from the erroneous ones. Traditionally, when designing fault-tolerance mechanisms, the *a priori* knowledge means that all possible operational modes are known at system design time and cannot adapt nor evolve during runtime. As a consequence, systems designed according to this principle are either fully static or allow a small number of operational modes only. Flexibility, however, is a desired property in a system in order to support evolving requirements, simplify maintenance and repair, and improve the efficiency in using system resources by using only the resources that are effectively required at each instant. This efficiency might impact positively on the system cost because with the same resources one can add more functionality or one can offer the same functionality with fewer resources.

However, flexibility and dependability are often regarded as conflicting concepts [Kop97]. This is so because flexibility implies the ability to deal with evolving requirements that, in turn, can lead to unpredictable and possibly unsafe operating scenarios. Therefore, it is commonly accepted that only a fully static system can be made dependable [RTC00], meaning that all operating conditions are completely defined at pre-runtime. In the broad sense and assuming

unbounded flexibility this assessment is true, but if one restricts and controls the ways the system could adapt to evolving requirements, then it might be possible to enforce continuous dependability.

The issue of real-time systems that need to adapt their behavior according to changes in external or internal factors has been addressed by the real-time community, specifically in the areas of feedback scheduling [LSTS02] and value-based scheduling [BPB+00][PBA03]. Feedback scheduling is a technique derived from the control theory that satisfies both transient and steady state performance specifications of real-time systems. Feedback scheduling algorithms have been used mostly in soft real-time applications working on open and unpredictable environments [SLST99]. Value-based scheduling is a decision problem involving the choice of a collection of services to execute so that the best possible outcome is achieved [BPB+00]. This technique is specially attractive for systems operating in unpredictable and unbounded environments, where some decisions are postponed until after the delivery or deployment of the system [PBA03]. Leaving some decisions as late as possible allows the behavior of the system to be better tailored according to the dynamic runtime conditions of the environment. According to Prasad *et al.* [PBA03], the motivation for dynamic schedules with run-time decisions is based on the inefficient resource usage and non-graceful degradation that are typical of static schedules. The inefficient resource usage results from the inherent pessimism of predicting resource requirements, while the non-graceful degradation is a consequence of the inflexible behavior of static schedules to failures and overloads.

Allowing run-time decisions involves identifying the extra services that need to be supported when spare resources are available, or identifying the services that need to be sacrificed when resources are scarce. However, postponing these decisions until run-time is worthwhile only if [PBA03]:

- A less pessimistic resource usage is obtained.
- There is sufficient run-time knowledge that allows better decisions to be made concerning the services to sacrifice or support.
- The overhead associated with run-time decisions does not outweighs the potential benefits.

Making reliable run-time decisions requires the definition of safety boundaries that cannot be crossed in any circumstances. In this way, run-time decisions may be taken provided they do not violate the safety boundary. The Simplex architecture [SRG94] supports the dependable evolution of real-time systems that use Commercial Off-The-Shelf (COTS) components. In Simplex, upgrades are supported by grouping a set of analytically redundant components (i.e., that satisfy the same abstract specification) into a subsystem module. Each module contains a safety component, a baseline component, and an optional new component. A module manager monitors the behavior of the new component and, if it behaves correctly, replaces the baseline

component with the new one. The run-time decisions in the Simplex architecture case are just switching back and forth from the baseline component to the new component (upgrade) in case the upgrade component tries to lead the system to an unsafe operating region defined and policed by the safety component.

The alternative functionality inherent to the value-based scheduling is closely related with graceful degradation [SK04] and focuses on tolerating failures not only with a redundant backup, but possibly also by relying on another component or subsystem that provides an alternative function. The system is in a degraded operating mode, but the degradation is a change in functionality rather than a loss of performance. Eventually, as resources are lost, system performance will degrade and some system services may be stopped to provide resources for other services that are mission-critical. Although there are some techniques [She03] to reduce the number of possible system configurations, that grow exponentially with the number of components (both hardware and software), alternative functionality is still hard to implement, specially in scarce resource distributed embedded systems.

Also, in the recent years real-time systems are used in more versatile and, often, dynamic scenarios. For example, some anticipate [SSM+] that future military systems should be self-adaptive, self-reflective and network-centric. Given the technological advance that cutting-edge military systems exhibit when compared with corresponding civil systems, it is expectable that some of these concepts and technologies will be also adopted in other systems, within a few years. According to [SSM+], some of the most challenging computing and communication requirements for new and planned combat systems can be characterized as follows:

- Multiple quality of service (QoS) properties must be satisfied in real-time.
- Different levels of service are appropriate under different configurations, environmental conditions, and costs.
- The levels of service in one dimension must be coordinated with and/or traded off against the levels of service in other dimensions to meet mission needs, e.g., the security and dependability of message transmission must be traded off against latency and predictability, and
- The need for autonomous and time-critical application behavior necessitates a flexible distributed system infrastructure that can adapt robustly to dynamic changes in mission requirements and environmental conditions.

The self-adaptive [LSZ+01] requirement expresses the ability to modify, either statically or dynamically, the system's functional and QoS-related properties. In a statically way by, e.g., minimizing hardware and software infrastructure dependencies or dynamically, by optimizing system responses to evolving environments or requirements, such as changing component interconnections, power-levels, CPU/network bandwidth, latency/jitter, and dependability needs.

The self-reflective [BCC+99] requirement goes a step further in providing the means for examining system's capabilities while the system is running, enabling automated adjustment for optimizing those capabilities. In this way, the self-reflective requirement supports more advanced adaptive behavior, i.e., the necessary adaptations can be performed autonomously based on conditions within the system or in the system's environment.

The wormhole metaphor [Ver03] has recently been applied to dependable adaptive real-time applications [MSCV04] running in wireless unstructured environments. The wormhole metaphor addresses the issues posed by uncertainty and proposes some guiding principles to the construction of distributed systems. It assumes that uncertainty is not uniform nor permanent across all system components, i.e., some parts are more predictable than others. It also promotes a proactive behavior in achieving predictability by making predictability occur whenever needed. In this way, the more predictable parts of the systems can be seen as wormholes since they will execute certain tasks faster or reliably than apparently possible in the other parts of the system. One of the possible instantiations of the wormhole concept, in the real-time context, is the timeliness property that takes the form of the Timely Computing Base (TCB) timeliness wormhole [VC02]. A system equipped with a TCB provides a set of dependable services (timely execution, duration measurement and timing failure detection) that are supported by a specialized architecture in which the TCB is a comparably small part of the whole system.

Systems based in the wormhole metaphor are adaptive in the sense that they handle the uncertainty of surrounding environment by adapting the QoS in a dependable way. That is, applications are notified by the low level dependable TCB services of timeliness violations and react by gracefully degrading the QoS and possibly leading the system to a fail-safe state, as reported in the cooperating cars demonstrator described by Martins *et al.* [MSCV04].

Dependable adaptation in distributed embedded real-time systems is usually the responsibility of the middleware [RWS01]. However, the middleware requires the support of the lower communication infrastructure to deliver flexible, yet dependable services to the application.

Most of the previous solutions, notably the ones arising from the real-time community, do not address the case of dependable distributed embedded systems. The problem becomes more complex when one considers flexible distributed embedded systems, where the issue of inter-node coordination arises. In such systems the real-time communications infrastructure plays a central role, since it must provide a set of services capable of efficiently supporting the distributed system requirements. Traditional real-time communication solutions adopted in safety-critical systems are fully static and offline defined in terms of messaging structure and are, thus, clearly unsuited for online adaptable distributed embedded systems. In this context, it is necessary to bridge the gap between the existing real-time scheduling techniques that already provide flexibility and the safety-critical real-time communication protocols that are mostly inflexible with a reduced ability to support online adaptation.

1.2 The thesis

The thesis supported by the present dissertation argues that:

> *It is possible to provide a high degree of operational flexibility, specifically online adaptation capabilities, in distributed embedded systems without compromising dependability. The FTT-CAN protocol can be used to achieve that purpose in the specific case of CAN based safety-critical systems.*

1.3 Contributions

The main contributions presented in this dissertation, including the proposed system architecture, mechanisms and components, are targeted to systems built on the Flexible Time-Triggered (FTT) [Ped03] paradigm and specifically the FTT-CAN protocol [APF02]. However, most of these contributions are equally applicable to dynamic master-slave architectures in general. The major contributions of this dissertation are summarized next.

1.3.1 Experimental assessment of CAN bit error rate

Most of the work on fault-tolerance in Controller Area Network (CAN) [ISO93] makes use of the bit error rate (BER) parameter of the CAN bus. However, there was no published data regarding this important parameter, and so some fault-tolerant mechanisms for CAN systems were based on quite pessimistic BER assumptions and not on real experimental data. This was the main drive for designing suitable test equipment capable of measuring CAN bit error rate in several utilization scenarios, with particular electromagnetic interference patterns.

From experimental data analysis, it was conjectured, for the considered environments, that in native CAN the occurrence of inconsistent message omissions has a lower probability than it was previously assumed. In fact it is below the 10^{-9} threshold usually accepted for safety-critical applications [Kop97]. However, the probability of inconsistent message duplicates (messages are eventually delivered but they could be out of order) in CAN is still high enough to be taken into account. This last finding, shows that one cannot neglect the occurrence of inconsistent message omissions in FTT-CAN because they are proportional to the number of inconsistent message duplicates in CAN.

The results obtained in the experiments do not pretend to be universally applicable, since they largely depend on the considered interference pattern that, in a limit scenario, could corrupt all legitimate bus traffic.

1.3.2 Architecture to achieve fault-tolerance in FTT-CAN

FTT-CAN impairments to dependability, namely the single point of failure formed by the master node and the fail uncontrolled nature of current FTT-CAN nodes, were identified and

discussed. Based on this, a general FTT-CAN architecture capable of delivering the desired level of operational flexibility without compromising safety was proposed. The proposed architecture considers the use of replicated components both masters and slaves, and enforces components to fail in a silent way.

The master replication scheme adds up some new problems, related with consensus and synchronization, that needed to be considered. When replicating the master, one needs to assure that all the instances of the data structures are coherent and that all traffic schedulers in all masters remain synchronized and coherent in every operational scenarios, i.e. they generate the same schedules synchronously. It is also important to refer that the proposed architecture may also include replicated transmission paths in the communication system.

1.3.3 Mechanisms to handle message omissions in FTT-CAN

Possible electromagnetic interference or other source of errors may cause a fault in message transmission with the correspondent message omission. Depending on the omission location within the elementary cycle (EC), several schemes capable of recovering from such situations were proposed. If the omission occurs in the trigger message transmission it is possible to retransmit it during the trigger message transmission window (typically until the middle of the EC) in order to remove the omission. If a synchronous message is omitted, this can be detected by the absence of answer to the trigger message in the respective EC. In this case, the master may use fault-tolerant scheduling techniques to try to recover the missing message, e.g. by accounting with time for possible retransmissions and rescheduling the message again within the deadline, if possible. The detection of missing synchronous messages can be also used to implement a membership service for slave nodes. In what concerns masters, a specific membership service is implemented based on a polling mechanism. An omission of an asynchronous message could be removed by retransmitting the omitted message according to CAN rules. These schemes are only valid in case of transient interferences that cause sporadic message omissions. In order to tolerate permanent slave node failure a replication scheme was also proposed.

1.3.4 Master replica synchronization after asynchronous start/restart

Upon an asynchronous startup or restart of a replicated master, its data structures will not be consistent nor synchronized with the ones from the active master. A master's synchronization mechanism was proposed to address this issue. It is based on the assumption that masters are fail-silent. Thus, as long as there is an output, it is correct. This implies that the current primary master is correct as long as it continues issuing messages to the slaves.

Replicated masters compare their internally generated EC-schedules with the ones transmitted by the primary master and when a difference is detected a synchronization request is issued. This request causes the transmission of the current data structures from the primary

to the requesting backup master as well as the synchronization of the schedulers, i.e. the set of instantaneous relative phasing. This mechanism is also used to support the failure of the active master and to replace it.

1.3.5 Protocol for consistent updates of FTT-CAN masters's data structures

Besides the need for a master's synchronization protocol able to transfer the active master data structures to other backup masters, there is also a need for a protocol able to enforce consensus among masters in case of an asynchronous request to update the master data structures, e.g., for changing the sampling rate of a given sensor reading of a particular slave node.

A protocol was proposed to handle master data structures update requests. This protocol takes advantage of some specific properties of CAN and FTT-CAN in order to reduce the protocol complexity as well as the computation and communication overheads. It is a semi-active protocol in the sense that all requests are processed in parallel by every master replica. However, in order to eliminate inconsistencies between masters, the active master is prioritized to the other masters. The active master rules all the process, assuming the role of the protocol *leader*, while the backup masters assume the role of *followers*. Possible local inconsistencies arising from lack of an atomic broadcast protocol are consistently cleared during the protocol execution by the active master in a *leader-followers* approach.

This protocol was also partially validated using a PROMELA model and the SPIN model checker for the case of a system with four nodes (three masters and one slave).

1.3.6 Enforcement of fail silence behavior in FTT-CAN nodes

A dual-processor CAN controller interface was presented to enforce fail-silence behavior at the master nodes. This custom interface enables the master node internal replication of the data structures and traffic schedulers in two different CPUs and compares both CPU outputs in terms of value and timing. In this way, master nodes are only allowed to transmit messages if both CPU outputs are identical and produced within a narrow time window.

Slaves also require fail-silence behavior and although one could adopt the same mechanism used in master nodes, that would be expensive. Thus, slave nodes fail-silence enforcement both in time and value domain should only be adopted in special cases where the slave node information (value **and** timing) is absolutely essential. In other cases, limiting slave nodes ability to transmit uncontrollably will suffice. This corresponds to enforce fail-silence behavior in the time domain only. An unconventional type of bus guardians was proposed to solve this problem.

The bus guardians are unconventional in the sense that they preserve FTT-CAN flexibility with respect to the traffic scheduling, in contrast with other completely static approaches (e.g., bus guardians of TTP/C and FlexRay).

From the slave's perspective, a schedule is valid only within the scope of an elementary cycle, thus the bus guardian policing a node only needs to be aware of the node schedule in a EC by EC basis. In this way the bus guardian decodes every trigger message contents and blocks any unscheduled transmission from the node.

1.4 Organization of the dissertation

In order to support the thesis previously stated, this dissertation is organized in the following way:

Chapter 2 – Presents background information concerning real-time communication in shared media and dependability in distributed systems. Particularly, it focuses on media access control policies, which have a direct impact on the timeliness of the communications, and on dependability topics such as distributed consensus, fail-silence failure mode, replica determinism, membership and fault modes.

Chapter 3 – This Chapter focuses on some relevant topics related with flexible and dependable real-time communication, including a discussion of flexibility *versus* dependability in several communication protocols and architectures, namely Controller Area Network (CAN) and CAN based protocols (TTCAN, FlexCAN, TCAN and FTT-CAN), as well as TTP/C, FlexRay and ARINC-629.

Chapter 4 – Discusses the impairments to dependability of CAN and FTT-CAN both at network level and at node level. The probability of transmission faults causing inconsistencies is analyzed based on the channel bit error rate. Since the existing values for CAN bit error rate were high and based in assumptions, it was decided to experimentally access CAN bit error rate. Experimental results have shown that CAN bit error rate is much lower than previously assumed. After discussing FTT-CAN impairments to dependability and having experimentally assessed CAN bit error rate, the fault hypothesis was defined. This Chapter concludes by presenting a general architecture able to enforce fault tolerance in FTT-CAN while delivering the desired level of operational flexibility under a controlled fashion so that dependability is continuously assured.

Chapter 5 – Discusses the impact of transient interferences that cause sporadic message omissions and presents several possible schemes to confine the impact of errors in the synchronous FTT-CAN traffic and to recover from those errors. A slave replication scheme capable of replacing faulty slave nodes is also presented.

Chapter 6 – This Chapter presents two different approaches to enforce fail silent behavior both in the master and in the slave nodes. Fail silence in the slave nodes is enforced using either dynamic bus guardians or internal replication and temporized agreement.

The latter mechanism enforce fail silent both in the time and in the value domain and it was designed to be used primarily on the master nodes. Notice however that internal replication and temporized agreement can also be adopted at the slave nodes whenever needed. Dynamic bus guardians in their turn are to be used on the slave nodes only, since they cannot be adopted in the master nodes because of the causal relation between the master node computed schedule and the bus guardian operation.

Chapter 7 Sets the conclusion of the dissertation and points out several directions for future work.

Chapter 2

Concepts of dependable real-time communication

2.1 Introduction

As it was referred in the previous Chapter, dependable embedded systems are becoming pervasive in many domains, e.g. in automotive vehicles, in avionics, in building automation, in factory automation, etc. These systems are usually distributed and rely on a fieldbus network to interconnect sensors, actuators and controllers in a reliable and timely way.

Traditionally, most of the distributed systems used in safety-critical applications are *federated* [Rus01], i.e. each function has its own fault-tolerant embedded control system with low connectivity (both in bandwidth and in criticality) to other functions. This facilitates the implementation of very strict frontiers between systems supporting different functions. In this way, faults are easily confined since the failure of one function has small impact on the continued operation of others because resources are not shared. However, the federated approach exhibits obvious drawbacks resulting from its excessive use of resources: each function needs its own computer system (which is generally replicated for fault tolerance), with all the attendant costs of acquisition, space, power, weight, cooling, installation, and maintenance. This is why there is a trend in recent applications towards *integrated* solutions, where some resources are shared across different functions. In such systems, the danger of fault propagation from one function to another is higher. One example of such integrated solutions is the Integrated Modular Avionics (IMA) that has emerged as a design concept to challenge the federated architecture [ARI91][Rus99]. Either federated or integrated, it is commonly accepted today that distributed systems outperform, in many ways, centralized systems [Bro03].

In a distributed system architecture the network must provide the communication services between the distributed components (nodes). This requires the use of an adequate protocol able to control the information flow and to provide some guarantees concerning timeliness, atomicity and dependability.

Timeliness guarantees are usually provided by real-time computing techniques. There are many definitions for real-time computing [Veg96], but probably the most widely adopted is the one from Stankovic and Ramamritham [SR89]:

> "*A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced.*"

This definition, however, does not fully address the distributed nature of some real-time systems. In fact, in these systems processes may share resources over different interconnected computing nodes and the system overall performance is influenced not only by the data processing but also by the communication delays.

There are two main classes of real-time systems: hard and soft real-time. Their distinction is based on the possible consequences from not meeting the time constraints, expressed as **deadlines**, i.e., the latest instants in time at which the results must be produced. Kopetz [Kop97] presents definitions for hard and soft real-time systems based on the types of deadlines that the system should meet. If a result has utility even after the deadline has passed, the deadline is classified as **soft**, otherwise it is **firm**. If a catastrophe, either in terms of human injury or assets loss, could result from missing a firm deadline then such deadline is classified as **hard**. It is important to note that some types of temporal constraints cannot be fully expressed in the form of a simple deadline. This is the case, for example, of causality relationships.

In real-time communications, the transmission of a message on a bus is equivalent to a non-preemptive process running in a CPU and both a bus and a CPU may be regarded as shared resources that need to be scheduled between concurrent messages or processes.

Despite this analogy, there is a fundamental distinction between real-time communications on a bus and real-time computing on a CPU: *knowledge* [HM90]. While in a single CPU computing system the scheduling decisions are made knowing the current state of all processes, the tasks that a distributed system is required to coordinate are based in*distributed knowledge*, i.e. the knowledge is distributed among the nodes of the system and it is not necessarily common to all nodes. The execution of simultaneous actions by a group of members of a distributed system requires common knowledge. Agreement, e.g. on a distributed scheduling decision, is an example of a simultaneous action in a distributed system and it requires specific support from the network.

Achieving common knowledge requires the execution of coordinated actions, but such actions cannot be guaranteed in many real distributed systems [HM90]. Only weaker variants of common knowledge that are guaranteed to be performed within a bounded amount of time, are attainable in practical cases. This problem is harder when coordinated actions need to be executed in bounded time, as in the case of real-time distributed systems.

Controlling the access to the shared resource (bus) is a pre-requisite to provide consensus and membership services, that, in turn, contribute for achieving common knowledge. The

rest of this Chapter presents an overview of medium access control policies and a survey of some relevant topics of dependability in the context of real-time communications, such as distributed consensus, fail-silence failure mode, replica determinism, membership and fault models.

2.2 Medium access control

In the specific case of bus based real-time communications, the bus is a shared medium between all nodes and it is the basis for providing low level common knowledge among the nodes. This must be carried out within a bounded time and thus, it is imperative that the access to the bus is also bounded in time, which implies that the medium access control (MAC) protocols must be deterministic. The MAC protocols determine the order of network access by contending nodes and, in the case of real-time communications, ensure that all nodes have the right to access the bus within a bounded time window.

According to Thomesse [Tho98], there are two main classes of MAC protocols: controlled and uncontrolled access (Figure 2.1). In the former there is a distributed knowledge in the network concerning the access rights to the bus, either based in the time or in explicit commands of a bus master, that prevents the nodes to transmit messages simultaneously. Two sub categories are normally considered within this group: centralized and distributed control. In the latter category, uncontrolled access, there is no distributed knowledge concerning the bus access, so every node may attempt to transmit at any given instant and possible collisions are detected and handled to prioritize access.

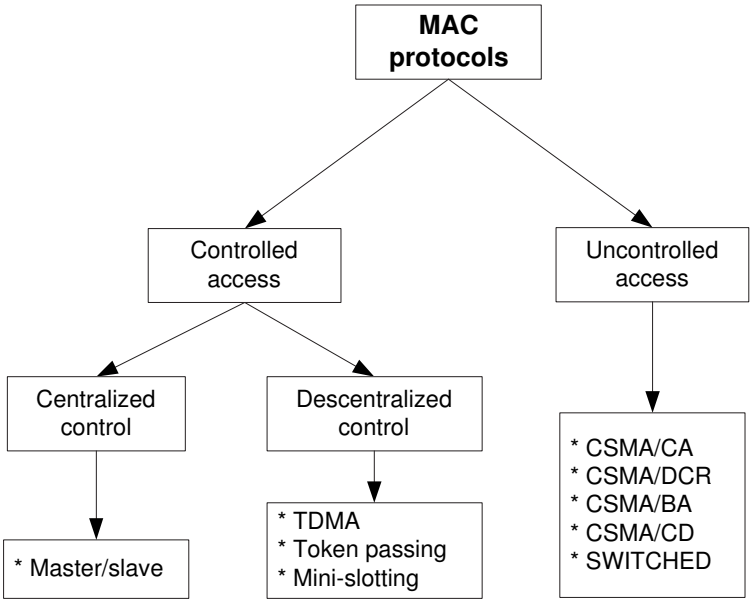


Figure 2.1: A taxonomy of MAC protocols (adapted from [Alm04]).

2.2.1 Centralized Control: Master-Slave

In a master-slave access scheme, the slave nodes access to the bus is granted by the master via a control message. Master messages act as synchronization points that slaves can use to synchronize their operation. A particular case of master-slave system is the master-multislave access control scheme [APF02] adopted in the FTT protocols, in which a single master message triggers the transmission of several slave messages reducing thus the overhead imposed by master control messages.

In a master-slave network, the traffic is scheduled by the master node only, which enables the use of any scheduling algorithm, either online or offline. This approach allows the use of quite simple slave nodes since the main processing is concentrated in the master node. These systems are normally well suited to handle periodic traffic. Aperiodic traffic is handled by some kind of pooling.

From the dependability point of view, the master is a single point of failure and for high reliability requirements it must be replicated.

There are many examples of master-slave systems such as the MIL-STD1553B [Hav86], PROFIBUS [IEC00] (between each master and the associated slaves), Ethernet Powerlink [Gro03b] and Bluetooth [Gro03a] (within each *piconet*).

The Master-Slave model can be combined with the Producer(s)-Consumer(s) model resulting in the Producer(s)-Distributor-Consumer(s) model proposed by Thomesse [Tho93] and adopted by WorldFIP [IEC00].

2.2.2 Distributed Control: Token-Passing

In a token-passing network, a token is circulated among the nodes according to a virtual ring. A given node can only access the bus when holding the token. Broadcast buses using this access control protocol are known as token-buses. Timely behavior is achieved limiting the time that a node can hold the token continuously before sending it to the next node.

There are two important parameters that control the timeliness of these networks, the Target Token Rotation Time and the Real Token Rotation Time. The former is a network parameter specified at system start-up. The latter is the time taken by the token in its last rotation. A node will be able to transmit for an interval corresponding to the difference between the first parameter minus the second, if positive. In any case, any node can always transmit at least a certain amount of traffic while holding the token, to prevent starvation.

Although this type of networks seems adequate to situations where the main processing resources are distributed over several nodes, it also exhibits some drawbacks. Firstly, variations in the token rotation time contribute to increased jitter of periodic traffic. Secondly, the number of nodes on the virtual ring increases the token rotation time and consequently limits the highest possible transmission rate for each message stream. Thirdly, the bus bandwidth is, under high load, equally distributed among all masters regardless of the applications require-

ments and fourthly, the token retention or loss by a node is an impairment to dependability.

PROFIBUS uses a timed-token protocol to control the access of a set of master nodes to the bus. Notice that PROFIBUS includes nodes of two sorts, masters and slaves, but the token is circulated among master nodes only. A master can communicate with the remaining masters while holding the token or with its associated slaves (master-slave).

2.2.3 Distributed Control: Virtual Token-Passing

Another token-based access control protocol is the virtual token-passing where the order by which nodes access the bus is determined by the respective address. Each node has an access counter which is always incremented simultaneously in all nodes after each successful transaction or after a timeout. A node is allowed to access the bus for a single transaction when its access counter contains a value equal to its own address. When the access counters pass beyond the number of nodes in the system, which is a network configuration parameter specified at start-up, the respective values are reset to one.

This protocol is more robust than real-token passing because it can easily handle the token loss situation (non-responding nodes).

P-Net [CEN96] protocol and Virtual Token-Passing Ethernet (VTPE) [CFP03] are examples of this type of protocols. This is sometimes referred to as mini-slotting [PBG99] despite the absence of a cyclic synchronization message.

2.2.4 Hybrid Control: Centralized Token-Passing

There is, yet, another token-based MAC protocol known as centralized token-passing. This is used in Foundation Fieldbus [IEC00] and combines centrally scheduled access with traditional token passing. In Foundation Fieldbus each link[1] has several *Link Masters*, LMs, one of which performs the role of the *Link Active Scheduler*, LAS. The LAS has a schedule describing the communication activity that must be executed at predefined instants in time. This communication activity can be of two types, either simple exchanges that follow the Producer-Distributor-Consumer model in which the LAS issues a *Compel Data* token to force a given producer node (known as publisher) to publish the respective data while the respective consumer nodes (known as subscribers) simultaneously collect it, or more complex sequences of exchanges which are carried out by another LM during a given pre-fixed amount of time. In this case the LAS issues an *Execute Sequence* token that allows the respective LM to perform the required sequence of exchanges. At the end, the LM must return the token to the LAS.

During the periods in which there is no scheduled communication activity, the LAS circulates a *Pass Token* through all the LMs specified in an internal list in a similar way as in

[1] A link is the logical medium by which Foundation Fieldbus devices are interconnected. It is composed of one or more physical segments interconnected by bus repeaters or couplers. All of the devices on a link share a common schedule which is administered by that link's current LAS.

traditional token passing. However, the duration of the token holding periods by each LM is controlled by the LAS in order to guarantee the timeliness of the scheduled traffic. The token passing mechanism uses a protocol similar to the one of PROFIBUS that relies on the difference between the real token (Pass Token) rotation time and the target rotation time parameter to control the amount of exchanges that each LM can perform while holding the token. The amount of traffic generated by each mechanism is bounded in order to allocate a minimum bandwidth to each one.

2.2.5 Distributed Control: Flexible Time Division Multiple Access or mini-slotting

This type of bus access scheme is very similar to virtual token-passing, but here the nodes are cyclically synchronized by a master that generates the clock base for all nodes and for the application software. In between synchronization pulses, all nodes can send messages according to the ascending identifier node sequence (an identifier may only be used by a single node). Upon receiving the synchronization pulse, all nodes start the slot counters. The slot counters begin counting at zero and count up to the highest identifier value for which a transmission request is present, the corresponding message is then transmitted and all the slot counters stop at the current value for the duration of the transmission. Once the transmission is complete, the slot counters begin counting upwards again.

This mechanism is purely time-controlled, and allows the deterministic transmission of a specific number of high-priority messages in every communication cycle even when the bus capacity is fully used. It also permits the flexible assignment of remaining bandwidth to low-priority messages.

This type of bus access scheme is adopted in byteflight [PBG99][M. 00] and in the aperiodic phase of ARINC-629 [ARI90].

2.2.6 Distributed Control: Time Division Multiple Access

The other form of distributed controlled access is the Time Division Multiple Access (TDMA). TDMA is based on the partitioning of global system time into exclusive time windows, during which only one node can access the bus.

The use of a TDMA scheme implies using a **clock synchronization** algorithm to provide common knowledge about time. Every node knows the instants when it should perform some actions and it also knows the instants of other nodes actions.

The widths of the time windows for each node to access the bus can be tailored in order to adequately support periodic traffic with low jitter.

Examples of TDMA based protocols are the Time-Triggered Protocol - TTP [TTT02][KG94], TTCAN [ISO00] and the time-triggered part of the FlexRay [Con04b].

2.2.7 Uncontrolled Access: Carrier Sense Multiple Access

Concerning uncontrolled access protocols, there is no external control signal, either explicit (e.g. token) or implicit (e.g. time) to instruct a node when to transmit. The arbitration is performed based on the bus status and on local information, only. These are generally known as Carrier Sense Multiple Access (CSMA) protocols according to which a node wishing to transmit listens to the bus and starts transmission only upon silence detection (carrier sense feature). Yet, collisions may occur since it is possible that several nodes detect silence on the bus at the same time and start transmitting almost simultaneously (multiple access feature). There are several types of CSMA protocols that differ on what is done upon a collision detection or on what is done to prevent collisions at all. In buses based on CSMA the arbitration required to decide which of the conflicting nodes is going to transmit next is completely decentralized and independent. This means that there is no centralized information concerning the present system configuration (e.g. number of nodes and communication relationships among them) and that the arbitration is carried out the same way whatever the configuration is. This fact grants CSMA-based systems a high level of flexibility, for example, making it possible from a functional point-of-view to connect or disconnect nodes during normal on-line operation without the need for system-wide knowledge.

2.2.8 Uncontrolled Access: CSMA-CD

The ability of a CSMA protocol to handle time-constrained communication depends on what is done to resolve collisions. For example, in CSMA-CD (collision detection) the nodes involved in a collision stop the transmission and try it again later, after a certain random time interval. This is the case of the well known Ethernet bus protocol (IEEE 802.3) in the original shared mode. Notice that chained collisions are possible until a node is granted access to the bus or the message is dropped. This becomes critical during heavy traffic loads and does not allow the determination of a practical upper bound to the latency that messages may suffer. Hence, the original Ethernet is not well suited to real-time communication except possibly for very low bandwidth utilization levels.

This situation has evolved in the last decade with the introduction of Ethernet switches that allow preventing collisions and thus open the possibility for deterministic behavior.

2.2.9 Uncontrolled Access: CSMA-BA

A different approach is used in the CAN fieldbus where the physical layer guarantees that collisions among statically prioritized messages are non-destructive. This is a sort of collision in which the resulting logic state of the bus is known and it is equal to the content of the message with highest priority involved in the collision. This scheme performs a bitwise arbitration (BA) in which the node transmitting the highest priority message gains immediate access to the bus. Nodes transmitting lower priority messages stop and contend again for the access to the bus as

soon as the current transmission terminates and so on until all messages are transmitted. This is a deterministic MAC protocol that allows, for statically defined message sets, to calculate *a priori* the maximum latencies that messages can suffer [TBW95]. This access method is also referred by different authors as CSMA-CA (Collision Avoidance), CSMA-DCR (Deterministic Collision Resolution) and CSMA-PCR (Priority Collision Resolution) [Alm99].

2.2.10 Uncontrolled Access: P-Persistent CSMA-CA

Another example of a CSMA-based fieldbus is the LONWORKS. This fieldbus uses yet another variant called *p-persistent* CSMA-CA (collision avoidance) [Cor99]. According to this scheme the nodes wanting to access the bus do not attempt to do it immediately after silence detection, as in normal CSMA-CD. On the contrary, upon silence detection each node wishing to access the bus waits for a random time, uniformly distributed with probability p over a predefined slotted time interval. This randomization of the access delays effectively reduces the probability of a collision resulting in improved performance for medium to high loads when compared with normal CSMA-CD. Nevertheless, *p-persistent* CSMA-CA still suffers from the unwanted thrashing effect, i.e. a reduction in the network throughput after a certain load level due to excessive number of collisions (chained collisions) and it adds an extra initial access delay even when the medium is free. This mechanism is also typical in wireless radio communications such as the IEEE 802.11 protocol.

In the LONWORKS protocol, the bus access control may also use an adaptive version of the *p-persistent* CSMA, called *predictive p-persistent* CSMA, in the course of acknowledgement transactions. The difference is that when collisions happen, the randomization interval is enlarged thus further reducing the probability of chained collisions. The protocol uses the positive acknowledgement of message transmissions to decide on whether to enlarge or reduce the randomization interval. When there are few or no collisions that interval is reduced to a minimum. On the other hand, under heavy traffic load, an initial increase in the number of collisions causes that interval to be enlarged which has an opposing effect of reducing those collisions. This is like a negative feedback control system where an equilibrium is reached that maintains the level of collisions approximately constant, independently of the traffic load. The result is a considerable improvement in the network performance for heavy traffic loads, avoiding the undesirable effect of thrashing. This only works with acknowledged transmissions.

2.3 Dependability and Real-time Communication

Dependability, safety and reliability and some other related words are often used to represent several similar concepts. In 1992, Laprie [Lap92] proposed a terminology that is widely used and will also be adopted in this thesis whenever possible. That terminology was enhanced in 2001 by Avizienis, Laprie and Randell [AAR01]. According to this enhanced terminology:

> "***dependability*** *of a computing system is the ability to deliver service that can justifiably be trusted. The* ***service*** *delivered by a system is its behavior as it is perceived by its user(s); a* ***user*** *is another system (physical, human) that interacts with the former at the* ***service interface****. The* ***function*** *of a system is what the system is intended for, and is described by the system specification.*".

The previous definition of dependability is rather subjective and may vary depending on the specific application and the users. Still according to [AAR01], the dependability concept integrates several attributes:

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

The **Security** concept from Laprie's initial taxonomy [Lap92] is represented in the revised classification as the concurrent existence of: availability (for authorized users only), confidentiality and integrity (with *improper* meaning *unauthorized*).

These attributes may be emphasized to a greater or lesser extent [AAR01] depending on the specific application: availability is always required, although to a varying degree, whereas reliability, safety, confidentiality may or may not be required.

In the context of this thesis, the first four attributes are the most important. Despite the fact that attributes related with security are becoming increasingly relevant, the possibility of malicious interference with embedded wired communications holds little concern since vehicles, robots and industrial machinery have yet closed networks that are not easy to tamper.

Availability, reliability, maintainability, integrity and safety are attributes of utmost importance in any distributed embedded system and, therefore, these attributes must be present in the communication sub-system, too.

After defining dependability and describing its attributes it is important to quantify dependability in terms of reliability and availability. The first remark concerning this issue is that the extent to which a system possesses the attributes of dependability should be interpreted in a **probabilistic** sense and not in an absolute and deterministic sense. This is due to the unavoidable presence or occurrence of faults that make systems never totally available, reliable, safe, or secure. Basically, fault scenarios are not deterministic by nature.

Knowing that dependability is based on correct delivery of services, a failure occurs when a service is not delivered according to the specification. System specification is a rather difficult task and the chance of making incorrect specifications is high [Lev00]. The use of formal specification methods that enable an early detection of design errors or incomplete specification details is a way to attenuate the impact of incorrect specifications.

Closely related with formal specification are the formal verification techniques, such as the model checking, by which [CGP99]:

> *"a desired behavioral property of a reactive system is verified over a given system (the model) through exhaustive enumeration of all the states reachable by the system and the behaviors that traverse through them."*

Applying model checking to a design consists of three main tasks. The first task is the modeling of the system in a formalism accepted by a model checking tool. The second task is the specification of the properties that the system should satisfy. The third task is the verification of these properties over the model using a model checking tool which automatically determines if the property holds for the model. The result of the verification is either 'yes', if the system satisfies the property specified, or a counterexample that shows a trace to the state where the property is not valid.

Reliability can be described as the probability of failure during a given time interval. According to the United States Federal Aviation Authority this probability is defined as [FAA88]:

> *Extremely improbable failure conditions are those so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type.(p. 14) (...) Extremely improbable failure conditions are those having a probability of the order of* 10^{-9} *or less.*(p. 15)

Achieving a probability of failure less than 10^{-9} is the goal of any safety-critical system, not only in the aviation industry. However it is difficult to guarantee and even infeasible to measure this level of reliability in a software based system [BF93] using statistical methods. To achieve such a low probability of failure, errors must be confined and faults have to be tolerated. A large range of mechanisms/techniques are available to give some reliability guarantees, however a detailed description of them is out of the scope of this thesis.

There are basically two main approaches to dependable systems design [Lat03]. One approach is probabilistic, a design is created and analyzed to prove that the probability of failure is low enough. To reason about these systems, accurate probability estimates are required. Unfortunately, probabilities are difficult to estimate with the degree of precision necessary for safety-critical systems. The second approach is to define a set of guarantees for the system, and relegate probabilities to the assumptions. Guarantees are proven to hold as long as a specified set of assumptions hold. While easier to reason about, a designer must

now examine the probability that the assumptions will hold. Common assumptions include: limits on the number of faults that can occur within a certain time interval, non-partitionable network, time limits on when a node may reintegrate, and the assumption that a majority of non-faulty nodes exists.

Clearly, these assumptions are not expected to hold for every imaginable fault scenario. The designer must show that the probability of pernicious faults occurring is *reasonably low*, and the dependability of the system is still acceptable. Unfortunately, this is the same problem that the probabilistic approach has, *reasonably low* as referred before, means a failure rate in the order of 10^{-9} failures/hour. A comprehensive fault model for fail-operational safety-critical systems will never meet all the assumptions. Stated otherwise, the dependability requirement is so high that odd or rarely occurring faults cannot be ignored. One additional complication is that safety analysis may depend on criticality. Faults or fault combinations with severe consequences or low controllability must be tolerated regardless of the probability. An example is the requirement of no single point of failure, regardless of the probability of that failure [MIS95].

2.3.1 Fault management

When producing a safety critical system much effort must be put into making the faults affect the system as little as possible. There are some techniques for doing this, aiming at different aspects namely: fault avoidance, fault removal, fault detection and fault tolerance.

Fault avoidance – Fault avoidance techniques aim to prevent faults from entering the system in the first place. This is the primary aim of the entire design process. Techniques as formal specification and model checking are some of the best practices to avoid design faults.

Fault removal – Fault removal is a set of techniques where one tries to find all faults introduced into the system during the design phase. This is done before the system is taken into use. Extensive testing of both hardware and software are part of fault removal methods.

Fault detection – The fault detection methods are applied to systems in use, to try to detect faults before they cause errors, minimizing their effects on the system. A system that uses fault detection algorithms is more suitable for safety critical systems because they continuously try to find faults in the system. These methods include, among others, functionality testing that checks whether the hardware still performs according to its specification. Information redundancy, on the other hand, aims to expose errors in the data by using CRC, checksums and error correcting codes.

Fault tolerance – Many techniques for fault tolerance are dependant on the existence of fault detection techniques. Fault tolerance is a property associated with the ability of

a system to accommodate possible faults without changing the behavior of the system. There are many techniques for fault tolerance including both hardware and software redundancy.

In the specific case of real-time communication the mechanisms to achieve fault-tolerance are not just concentrated in the network nodes to guarantee their dependability via voting schemes, N-version programming, hardware redundancy, and virtually all other general purpose techniques. The faults originated (or propagated) in the communication channel may compromise the ability of the distributed system to achieve common knowledge. A faulty message originated in the bus (e.g., an undetected bit error) or transmitted by a faulty node may be propagated to all the nodes causing the whole system to collapse. Notable examples of these latter faults are timing or value failures in a node, non-atomic broadcast, replica non-determinism and electromagnetic induced errors in the channel.

2.3.2 Distributed consensus

Consensus [PSL80] is having a group of n processes in a distributed system agreeing on a value. A consensus protocol is an algorithm that produces such an agreement. Each process in a consensus protocol has, as part of its initial state, an input from some specified range, and must eventually decide on some output from the same range. Deciding on an output is irrevocable; though a process that has decided may continue to participate in the protocol, it cannot change its decision value.

The importance of the consensus problem derives from its omnipresence in the area of distributed systems. Indeed, consensus is at the basis of solutions to achieve synchronization, reliable communication, atomic commitment, consistency control, resource allocation, replicated file systems, sensor reading, etc.

In order to solve the consensus problem the protocol has to satisfy the following safety and progress properties [CT96]:

Termination –Every correct process eventually decides some value.

Uniform integrity – Every process decides at most once.

Agreement – No two correct processes decide differently.

Uniform validity – If a process decides v, then v was proposed by some process.

Consensus in the presence of faults is difficult to attain. Systems with different levels of synchrony or different kinds of failures require different algorithms. At one extreme, a system can be totally asynchronous, in that no assumptions can be made about the relative speeds of the processes or the communication medium. At the other extreme, a system can be totally synchronous where we can assume upper bounds on processing and communication delays.

Usually, two kinds of failures are considered. Fail-stop failures cause a process to die at any time and stop participating in the algorithm. Byzantine failures are those where a process sends incorrect information, possibly according to a malevolent plan.

Consensus with fail-stop failures

In 1985 Fischer *et al.* [FLP85] proved the impossibility of achieving a deterministic solution for the consensus problem in an asynchronous distributed system with just one faulty process (usually known as the FLP impossibility result).

Considering a completely asynchronous system, with no assumptions about the relative speeds of the processes or the speed of communication, it is not possible to check if a process has failed if there is no reference to a clock to implement some kind of time-out mechanism. Assuming also reliable communication and that at most one process may fail-stop at any time, there is no algorithm which can guarantee consensus on a binary value in finite time.

The reason for this inconvenient result is based on the fact that it cannot be decided whether a process has died or if it is just very slow in sending its message (i.e. it is impossible to distinguish between a process which is arbitrarily delayed and one which is indefinitely delayed).

This result also means that totally asynchronous systems can never have any kind of fault-tolerance, since they cannot even handle the most benign of faults under the best conditions. To achieve fault-tolerance in asynchronous systems requires making some assumptions about the system or about the kinds of faults which can be handled. In real systems, this is usually done by assuming an upper bound in communication and processor speed, and considering a process faulty if it does not respond within a bounded time.

There are basically three ways to circumvent the FLP impossibility and achieve fault-tolerance in asynchronous systems:

Change the definition of *consensus* – Instead of requiring consensus in finite time, achieve consensus in finite time with probability 1. That is, there is a chance that the algorithm will be indefinitely delayed, but the probability of this happening is 0 [BT85]. This is usually know as ***randomization***. An excellent survey on work using randomization can be found in [Asp03].

Partial synchrony – Relax the definition of *asynchronous* and allow some level of synchrony. Three kinds of asynchrony in the system described by Fisher*et al.* [FLP85] may be identified: process asynchrony, communication asynchrony and message order asynchrony. Fisher *et al.* proved that making the processes synchronous only is not enough. But making either the communication or message order synchronous is enough alone. The previous results are valid as long as each processor can perform an *atomic step*, consisting of receiving a message, performing some computation, and sending messages to

other processes. If an arbitrary delay is allowed in the middle of the operation, then no consensus algorithm is possible with only communication synchrony.

Failure detectors – Since it is impossible, in an asynchronous system, to detect the death of a process, and to distinguish a dead process from a merely slow one, adding failure detectors would solve the consensus problem without relaxing either the consensus definition or the assumptions concerning the system synchrony.

A failure detector [CT91] is a module that keeps a list of processes which it thinks have crashed, and regularly probes each process to update its list. Since the failure detector cannot be sure of a process death any more than any other process can, it should be assumed that it will not only make mistakes, but will make an infinite number of mistakes. The weakest properties that this module must have are: all fail-stop processes are eventually detected and any correct process which is on the list of failed processes should eventually be taken off the list.

This concept can be implemented in practice by having the failure detector probing each process regularly. Every unresponsive process **P** is placed on the list and a broadcast message is sent to all processes (including **P**) announcing its death. If **P** has not crashed, then it will eventually refute its death announcement. Chandra and Toueg [CT91] show that this weak and unreliable model of failure detectors allows the consensus problem to be solved.

Consensus with Byzantine failures

Solving consensus in environments where processes can exhibit Byzantine behavior is notably difficult [BHRT03]. The Byzantine Generals Problem, introduced by Lamport *et al.* [LSP82], is a model for the consensus problem in the light of faulty processes which send false messages. Faults induced by faulty processes sending faulty messages according to some malevolent plan (also known as asymmetric or arbitrary faults), are the most serious impairment to distributed consensus. Any algorithm able to achieve consensus in the presence of Byzantine faults will be also able to handle arbitrary faults.

According to the Byzantine Generals Problem, a Byzantine commanding general, who has surrounded the enemy with his armies each led by a lieutenant general, wishes to organize a concerted plan of action, i.e., to attack or to retreat. However, the Byzantine corps of general has been infiltrated by traitors. Despite this, the loyal Byzantine lieutenant generals must all reach the same conclusion either to attack or to retreat by sending messages back and forth among themselves. Moreover, their conclusion must agree with the commanding general's order. An algorithm which completes this problem successfully is said to reach Byzantine agreement. The solution to this problem yields a different answer for oral and for written messages.

Oral messages – If one-third or more of the generals are traitors, then no consensus is possible, i.e., $3f + 1$ processing elements are needed to tolerate f Byzantine faults.

Written messages – Consensus is always possible since when a general is relaying information he cannot change a message, because messages are signed and any change would be detected by the receiving general.

In a distributed system the Byzantine generals are replaced by processing elements. The algorithm for solving the Byzantine Generals Problem with n processing elements requires $f + 1$ rounds to complete. Fischer and Lynch [FL82] showed that at least $f + 1$ rounds are needed for all deterministic solutions to the Byzantine Generals Problem, however, the message size grows exponentially at each round ($O^{(nf+1)}$) [PSL80]. Over the years a number of more efficient algorithms with respect to the number of messages required to reach agreement, have been proposed. Barborak *et al.* [BDM93] presents a good survey of this issue.

The solutions based on *"oral messages"*, assume that the group of processors is completely connected to allow for private communication between any pair of processors.

Byzantine agreement becomes much simpler if messages are authenticated or signed [PSL80] [LSP82]. A message is authenticated if:

1. a message signed by a fault-free processing element cannot be forged
2. any corruption of the message is detectable and
3. signature can be authenticated by any other processing element

This, obviously, limits the malevolent plans of a faulty processor. In this situation, there is no limit on the number of faulty processors that are tolerable, and the network no longer requires private communication channels between processing elements.

It emerges from the previous results that Byzantine agreement solutions manipulate three independent resources: processing elements, rounds and message size. Although it is possible to create procedures that are optimal in some of these respects, no algorithm optimized in all three categories has been found [BDM93].

A number of possible solutions for solving the consensus problem in environments where processes can exhibit Byzantine behavior have been presented over the years, however they are out of the scope of this dissertation. The reader should refer for Malkhi and Reiter [MR97], Kihlstrom *et al.* [KMMS97], Doudou and Schiper [DS98], Baldoni *et al.* [BHRT03] and Aguilera *et al.* [ADGFT04] for additional details.

Distributed consensus is one key issue in fault-tolerant distributed systems and there are several algorithms that can be used to solve this problem. Choosing the right algorithm greatly depends on the type of distributed system on which the protocol will run and the assumptions that may be done concerning the expected faults the system is suppose to circumvent (fault hypothesis).

2.3.3 Fault-tolerant broadcasts

As it was explained previously, distributed consensus is a paradigm that simplifies the task of designing fault-tolerant distributed applications, since consensus allows processes to reach a common decision that depends on their initial inputs, despite the occurrence of possible failures. Theoretical research on fault-tolerant distributed computing has largely focused on consensus, while applied research has focused on reliable broadcast and its variants [HT94].

Given their wide applicability, fault-tolerant broadcasts and consensus have been extensively studied, resulting in a voluminous literature. However, according to Hadzilacos and Toueg [HT94] this extensive literature is not distinguished for its coherence and the close relationship among these problems is often obfuscated. In fact, total order broadcast and consensus are equivalent problems, i.e., if there exists an algorithm that solves one problem, then it can be transformed to solve the other problem. Dolev *et al* [DDS87] have shown that total order broadcast can be transformed into consensus, and Chandra and Toueg [CT96] have shown that consensus can be transformed into total order broadcast.

Fault-tolerant broadcasts are communication primitives that facilitate the development of fault-tolerant applications. The weakest among these is *reliable broadcast* allowing processes to broadcast messages such that all processes agree on the set of messages they deliver, despite failures. Stronger variants of reliable broadcast impose additional requirements on the order in which messages are delivered (e.g., processes may have to deliver all messages in the same order).

Reliable broadcast is the weakest type of fault-tolerant broadcast and it guarantees three properties:

1. All correct processes agree on the set of messages they deliver.

2. All messages broadcast by correct processes are delivered.

3. No spurious messages are ever delivered.

For some applications, these properties may be sufficient however, reliable broadcast imposes no restriction concerning the order in which the messages are delivered. However in some applications message delivery order is important, thus, a collection of stronger types of broadcasts, differing in the guarantees they provide on message delivery order, will be described based in the definitions/taxonomy of Hadzilacos and Toueg [HT94].

FIFO broadcast is a reliable broadcast that guarantees that messages broadcast by the same sender are delivered in the order they were broadcast. *Causal broadcast*, requires messages to be delivered according to the causal precedence relation, i.e., if the broadcast of m causally precedes the broadcast of m' then m must be delivered before m'. If two messages are not causally related, however, different processes can deliver them in different orders. *Atomic broadcast* prevents this undesirable behavior by requiring processes to deliver all messages in

the same order. Finally, *FIFO atomic broadcast* combines the requirements of FIFO broadcast and atomic broadcast, and causal atomic broadcast combines the requirements of causal broadcast and atomic broadcast.

The subsequent definitions of the various types of broadcast assume the occurrence of benign failures only, in order to simplify the definitions and also to strengthen the properties of broadcasts in ways that are important in practice. After presenting the definitions of the various types of broadcasts, the modifications necessary to consider arbitrary failures (Byzantine) will be presented.

Reliable broadcast

Reliable broadcast [HT94] requires that all correct processes deliver the same set of messages (Agreement), and that this set includes all the messages broadcast by correct processes (Validity) but no spurious messages (Integrity).

More formally, reliable broadcast is defined in terms of two primitives: `broadcast` and `deliver`. When a process p invokes `broadcast` with a message m as a parameter, then p *broadcasts* m. It is assumed that m is taken from a set M of possible messages. When a process q returns from the execution of `deliver` with message m as the returned value, then q *delivers* m.

Since every process can broadcast several messages, it is important to univocally identify the message's sender, and to distinguish between the different messages broadcast by a particular sender. Thus, every message m includes the identity of its sender, denoted *sender(m)*, and a sequence number, denoted *seq#(m)*. If *sender(m)* $= p$ and *seq#(m)* $= i$ then m is the i*th* message broadcast by p.

Reliable broadcast is a broadcast that satisfies the following three properties:

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Agreement*: If a correct process delivers a message m, then all correct processes eventually deliver m.
- *Integrity*: For any message m, every correct process delivers m at most once, and only if m was previously broadcast by *sender(m)*.

Validity together with agreement ensures that a message broadcasted by a correct process is delivered by all correct processes. However, if the sender of a message m is faulty, the specification of reliable broadcast allows two possible outcomes: either m is delivered by all correct processes or by none. For example, if a process p crashes immediately after invoking `broadcast`(m), correct processes will never be aware of p's intention to broadcast m, and thus cannot deliver anything. On the other hand, if p invokes `broadcast`(m) and fails during the execution of this primitive after having sent enough information about m, then correct processes may be able to deliver m.

FIFO broadcast

FIFO broadcast aims to contextualize each message to avoid message misinterpretation. In this way, a message should not be delivered by a process that does not know its context. In some applications, the context of a message m consists of the messages previously broadcast by the sender of m. For example [HT94], in an airline reservation system, the context of a message canceling a reservation consists of the message that previously established that reservation: the cancelation message should not be delivered at a site that has not yet "*seen*" the reservation message. Such applications require the semantics of FIFO broadcast, a reliable broadcast that satisfies the following requirement on message delivery:

- *FIFO order*: If a process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.

Causal broadcast

FIFO order is adequate when the context of a message m consists only of the messages that the sender of m broadcast before m. A message m, however, may also depend on messages that the sender of m effectively *delivered* before broadcasting m. In this case, the message delivery order guaranteed by FIFO broadcast is not enough. An example of this [HT94] is a network news application where users distribute their articles with FIFO broadcast. The following undesirable scenario could occur:

1. User A broadcasts an article;
2. User B, at a different site, delivers that article and broadcasts a response that can only be understood by a user who has already seen the original article;
3. User C delivers B's response before delivering the original article from A and so misinterprets the response.

Causal Broadcast intensifies FIFO broadcast by preventing the previous problem generalizing the notion of a message *depending* on another one, and ensuring that a message is not delivered until all the messages it depends on have been delivered. This more general notion of dependence is captured with the causal precedence relation on message broadcasts and deliveries. Given a causal precedence relation, causal broadcast is defined as a reliable broadcast that satisfies:

- *Causal order*: If the broadcast of a message m causally precedes the broadcast of a message m', then no correct process delivers m' unless it has previously delivered m.

Atomic broadcast

If the broadcasts of two messages are not related by causal precedence, causal broadcast does not impose any requirement on the order they can be delivered. In particular, two correct processes may deliver them in different orders. This disagreement on message delivery order is undesirable and potentially catastrophic in some applications. For example [HT94], consider a replicated database with two copies of a bank account X residing at different sites. Initially, X has a value of 100€. A user deposits 20€ triggering a broadcast of "*add 20€ to X*" to the two copies of X. At the same time, at a different site, the bank initiates a broadcast of "*add 10% interest to X*". Because these two broadcasts are not causally related, causal broadcast allows the two copies of X to deliver these update messages in different orders. This results in the two copies of X having different values, creating an inconsistency in the database.

To prevent such problems, atomic broadcast requires that all correct processes deliver all messages in the same order. This total order on message delivery ensures that all correct processes have the same *view* of the system, allowing them to act consistently without any additional communication. Formally, an atomic broadcast is a reliable broadcast that satisfies the following requirement:

- *Total order*: If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

The agreement and total order requirements of atomic broadcast imply that correct processes eventually deliver the same sequence of messages.

FIFO atomic broadcast

Atomic broadcast does not require that messages be delivered in FIFO order. For example, atomic broadcast allows the following scenario: a process suffers a transient failure during the broadcast of a message m, and then broadcasts m', and correct processes only deliver m'. Thus, atomic broadcast is not stronger than FIFO broadcast. Therefore, there is the need to define FIFO atomic broadcast which is a reliable broadcast that satisfies both FIFO order and total order. FIFO atomic broadcast is stronger than both atomic broadcast and FIFO broadcast.

Causal atomic broadcast

FIFO atomic broadcast does not require messages to be delivered in causal order. Considering again the earlier network news example, and supposing that FIFO atomic broadcast is used to disseminate articles. The following undesirable scenario is possible. Faulty user A broadcasts an article and faulty user B, who is the only one to deliver that message, broadcasts a response and then immediately crashes (before delivering its own response). Correct user

C delivers the response, although it never delivers the original article. Thus, FIFO atomic broadcast does not necessarily satisfy causal order.

Causal atomic broadcast is, then, a reliable broadcast that satisfies both causal order and total order. Causal atomic broadcast is stronger than both FIFO atomic Broadcast and causal broadcast. This type of broadcast is the key mechanism of the State Machine approach to fault-tolerance [Sch90].

The relations among these six types of broadcasts, in terms of their order properties, is illustrated in Figure 2.2.

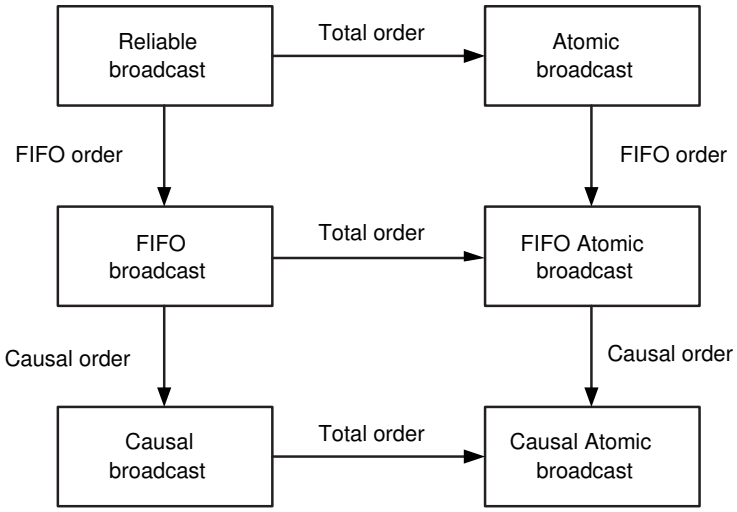


Figure 2.2: Relationship among the broadcast primitives (adapted from [HT94]).

Timed broadcasts

Real-time applications require that if a message is delivered at all processes, then it is delivered within a bounded time after it was broadcast. This property is called Δ-*Timeliness*. As usual, in a distributed system elapsed time can be interpreted in two different ways: real-time as measured by an external observer, or local time, as measured by the local clocks of the processes. This originates two different ways of defining the Δ-*Timeliness* property. The one corresponding to real-time is:

- *Real-time* Δ-*Timeliness*: There is a known constant Δ such that if a message m is broadcast at real-time t, then no correct process delivers m after real-time $t + \Delta$.

On the other hand, the definition of Δ-*Timeliness* in terms of local clocks bounds the difference between the local broadcasting time and the local delivery time. To formally specify such a bound, it is assumed that each message m contains a *timestamp ts(m)* denoting the local time at which m was broadcast according to the sender's clock. That is, if a process p

wishes to broadcast a message m when its local clock shows c, then p tags m with *ts(m)=c*. The definition of Δ-*Timeliness* that corresponds to local time is:

- *Local-time* Δ-*Timeliness*: There is a known constant Δ such that no correct process p delivers a message m after local time *ts(m)*+ Δ on p's clock.

A broadcast that satisfies either version of the Δ-*Timeliness* property is called a timed broadcast. For example, timed reliable broadcast is a reliable broadcast that satisfies local- or real-time Δ-*Timeliness*. When referring to a timed broadcast, one must explicitly state which of the two timeliness properties is assumed. The parameter Δ is called the latency of the timed broadcast.

Uniformity

The agreement, integrity, order, and Δ-Timeliness properties of the broadcasts defined so far place no restrictions on the messages delivered by faulty processes. Since up until now, only benign failures have been considered, such restrictions are desirable and achievable. For example, the agreement property states that if a correct process delivers a message m, then all correct processes eventually deliver m. This requirement allows a faulty process to deliver a message that is never delivered by the correct processes. This behavior is undesirable in many applications, such as atomic commitment in distributed databases [BHG86][zBT93] and can be avoided if the failures are benign. For such failures, the agreement property can be strengthened to:

- *Uniform agreement*: If a process (whether correct or faulty) delivers a message m, then all correct processes eventually deliver m.

Likewise, the other properties of broadcasts can be strengthened to include the case of messages delivered by faulty processes.

Broadcast specifications for arbitrary failures

The broadcast specifications presented so far were based in the assumption that only benign failures could occur. In order to support arbitrary failures, some modifications to these specifications are required. As it was previously discussed (beginning of section2.3.3), processes are allowed to broadcast and deliver any message $m \in M$ and each message must include some fields, such as a sender's ID, *sender(m)*, a sequence number, *seq#(m)* and possibly a timestamp, *ts(m)*. In a system with arbitrary failures, it cannot be assumed that messages broadcast by processes that commit arbitrary failures belong to M since they may not have the appropriate fields and so it is expectable that correct processes ignore such messages. With this latter assumption, a correct process can always extract *sender(m)*, *seq#(m)* and when appropriate, *ts(m)* from any message m that it delivers. Notice that a process p that

commits arbitrary failures may broadcast a message m with $sender(m) \neq p$ or with the wrong sequence number, or with a totally arbitrary timestamp.

Consider the case of reliable broadcast with arbitrary failures. The definitions of Validity and Agreement only refer to messages broadcast and delivered by correct processes, and thus, are not changed. There are, however, some problems with the definition of Integrity, which needs to be redefined for the case of arbitrary failures, as follows:

- *Integrity*: For any message m, every correct process delivers m at most once, and if *sender(m)* is correct then m was previously broadcast by *sender(m)*.

Considering now the case of FIFO broadcast, the benign failure version of FIFO order imposes an order on the delivery of messages broadcast by a process p that may be faulty. However, if p commits arbitrary failures, such an order is not meaningful. Thus, in the case of arbitrary failures, the order requirement is weakened by restricting its application only to messages broadcast by correct processes:

- *FIFO order*: If a correct process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.

According to [HT94], the definition of causal broadcast in the presence of arbitrary failures is of questionable utility, since the context of a message broadcast by a correct process, i.e., its *causal past*, may include the delivery of a message from a process that committed arbitrary failures. Thus, causal broadcast with arbitrary failures is usually not considered.

In what concerns atomic broadcast with arbitrary failures, the definition of reliable broadcast in that case has already been provided and the definition of total order refers only to deliveries by correct processes. Hence the definition of atomic broadcast remains unchanged, for the case of arbitrary failures.

Finally, each version of Δ-*Timeliness* is considered. The definition of Local-time Δ-*Timeliness* refers only to actions of correct processes, and remains unchanged. Real-time Δ-*Timeliness* however, refers to the real-time at which a message is broadcast, which is an ambiguous definition since the sender of that message may be subject to arbitrary failures. This problem is circumvented by restricting the requirement to messages broadcast by correct processes only:

- *Real-time* Δ-*Timeliness*: There is a known constant Δ such that if a message m is broadcast by a correct process at real time t, then no correct process delivers m after real-time $t + \Delta$.

Inconsistency and contamination

The problem of contamination comes from the observation that, even with the strongest specification, total order broadcast does not prevent a faulty process p from reaching an

inconsistent state. This is a serious problem because p can broadcast a message, in total order, based on this inconsistent state, and thus contaminate correct processes [GT91][HT94].

Considering an application where processes communicate via fault-tolerant broadcasts (Figure 2.3) and assuming that only benign failures may occur, the current state of every process (whether correct or faulty) depends on the messages that it has delivered so far. This state, and the application protocol that the process executes, determines whether it should broadcast a message, and if so, the contents of that message.

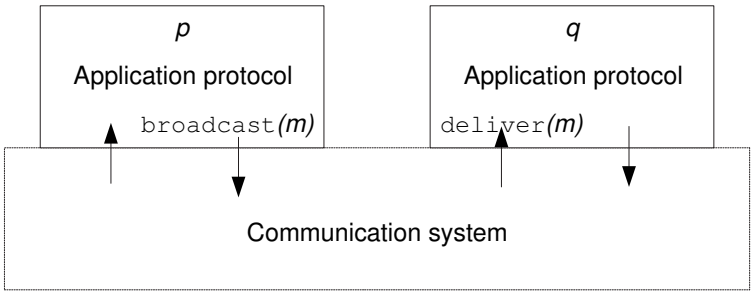


Figure 2.3: Application protocol using broadcasts (adapted from [HT94]).

Figure 2.4 illustrates an example where an incorrect process contaminates the correct processes. Process *p3* delivers messages *m1* and *m3*, but not *m2*. So, its state is inconsistent when it multicasts *m4* to the other processes before crashing. The correct processes *p1* and *p2* deliver *m4*, thus getting contaminated by the inconsistent state of *p3*.

If a process executes correctly prior to crashing, it is intuitive to assume that all message deliveries before the crash are consistent with the rest of the system, however as the previous example shows, this is not always true.

Inconsistency can be eliminated assuming benign failures, however this calls for a new class of algorithms to implement *reliable* broadcast. Intuitively, a process can prevent its contamination by refusing to deliver messages from processes whose previous deliveries are

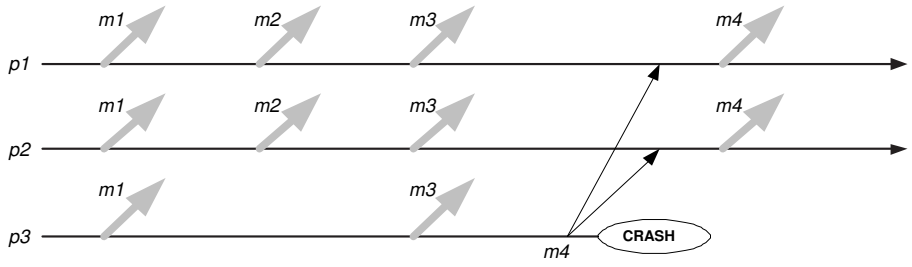


Figure 2.4: Contamination of correct processes *p1* and *p2* by a message *m4* based on an inconsistent state (*p3* delivered *m3* but not *m2*) (adapted from [DSU03]).

not compatible with its own. The amount of information that each message should carry, so that every process can determine whether it is safe to deliver it, depends on the type of broadcast considered and on the failure assumptions. Preventing inconsistency is, however, more difficult and costly because it requires techniques that allow a faulty process to detect whether it is about to make a message delivery error, and, if so, to immediately stop (i.e., to implement a fail-silent failure mode). A precise definition of inconsistency and contamination with respect to broadcasts can be found in [GT91].

It is important to remark that, with arbitrary failures, neither inconsistency nor contamination can be prevented. This is because the state of a faulty process may be inconsistent even if it delivers all messages correctly. This process may then contaminate the rest of the system by broadcasting an erroneous message that seems correct to every other process.

2.3.4 Fail-silence failure mode

As it was referred in the previous section, to prevent inconsistencies and contamination, it is desirable that a faulty process detects whether it is about to make a message delivery error and, in that case, immediately stop its action, i.e., to implement a *fail-silent* failure mode. The very same behavior is also desirable in nodes of a distributed system, to avoid inconsistencies and their propagation to other nodes. With fail-silence behavior, an error inside a node cannot affect other nodes and thus each node becomes a separate fault confinement region (FCR).

The use of fail silent nodes also reduces the complexity of designing fault-tolerant systems, since only $k+1$ replicas are needed to tolerate k failures of a functional unit while, for fail uncontrolled replicas $3k+1$ are required.

According to [Tem98], a node is considered to be fail-silent if it:

- sends correct messages at specified points in time that can be verified as being correct by all non-faulty receivers;
- sends corrupted messages at specified points in time that can be identified as being corrupt by all non-faulty receivers. These messages are discarded.
- sends no messages at all;

This definition of fail-silence describes the behavior of a node both in the time and in value domains. However, stronger constraints are made to the temporal behavior of a fail-silent node than to the behavior in the value domain. This is so because fail-silence in the value domain is not compromised as long as the messages sent are detectably corrupted, while fail-silence in the time domain is violated whenever any deviation from the specified instant, at which the messages can be sent, occurs.

The alternatives to enforce fail silent behavior may be generically divided in two main groups; the ones that result from adding redundancy to each node and ones that rely on behavioral error detection techniques [Tem98].

Using replicated processing within a node with output comparison or voting calls for the use of mechanisms to keep the replicas perfectly synchronized and to avoid replicas to diverge due, e.g. to asynchronous events. Synchronization at processor instruction level is the most obvious way to achieve replica synchronism, driving identical processors with the same clock source and evaluating their outputs (either comparing or voting) at critical instants, e.g. every bus access. Special care must be taken with asynchronous events that must be delivered to the processors so that all perceive the same event at the same point of their instruction streams.

Over the years many systems were designed based in double-processor fail silent nodes such as Sequoia [Ber88] and Stratus [WB91]. However, these systems do have some drawbacks [BES+96]. First of all the processors must exhibit the same deterministic behavior every clock cycle and don't care states are not allowed so that they produce identical outputs. Secondly, the use of special purpose hardware as comparators or voters, reliable clock sources and asynchronous event handlers greatly increases the design complexity. Finally, due to their operation in lock step, a transient fault could affect both processors in the same way, making the node susceptible to common mode failures. An alternative approach to eliminate the hardware level complexity of the solutions referred above is to transfer the replica synchronism to a higher level (process or task level) using software protocols over a set of standard processors operating independently of each other in a node. Task synchronization approaches were used in SIFT [Wen78] and in Voltan [SES+92].

Behavioral error detection mechanisms, either in software or in hardware, are another alternative for enforcing fail-silence behavior. Mechanisms such as checksums, watchdog timers and processor monitoring, are usually implemented using COTS components. Error detection latency is the major bottleneck of these systems since the error detection mechanisms are only able to detect errors a relatively long time after they occur.

Bus guardians, which are autonomous devices with respect to the node network controller and host processor, also implement behavioral error detection mechanisms that contribute to reduce possible residual fail silent violations resulting from the error detection latency by enforcing an adequate timing in the node transmissions.

Bus guardians

The purpose of the bus guardians is to increase the probability that nodes in a cluster will face faults covered by the fault hypothesis only. This is usually accomplished by placing a guardian at the component's network interface(s) and making it control the appearance of the respective component at the interface acting, thus, as a failure mode converter [KS03]. As a result, the failure modes of the component are, at the interface to other components, replaced by the failure modes of the guardian.

Figure 2.5 illustrates a generic setup of a guardian protecting a fault containment region. Basically the guardian mimics the behavior of a component interface that is compliant to the fault hypothesis. At its output interface the guardian will reproduce the input received

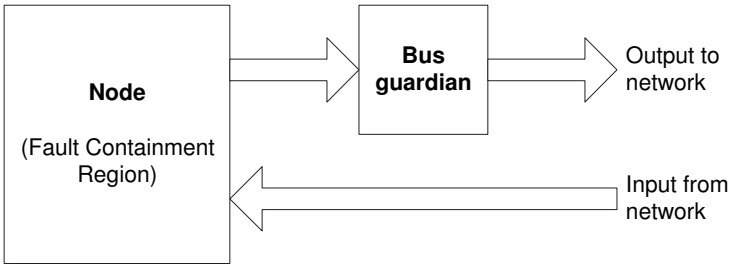


Figure 2.5: Generic bus guardian.

from the attached unit if this input complies to some specified rules (e.g., the correct timing). Otherwise the guardian will exhibit a predefined behavior, compliant with the fault hypothesis (e.g., by blocking the network access to its attached unit). In this sense the bus guardian filters all transmissions to the network and disables network access from faulty nodes that otherwise could occupy scheduled time of other nodes. In a limit situation, the *babbling idiot* failure mode, in which a faulty interface transmits constantly, all legitimate network traffic could be disrupted if the bus guardian was not mediating the node's access to the network.

To be effective in the message filtering, the bus guardian must fail independently with respect to the interfaces it monitors, i.e. it must belong to a separate FCU (Fault Confinement Unit). Thus, it needs to have its own copy of the traffic schedule and an independent knowledge of the time [Rus01].

The bus guardian may be physically located as part of the node computer (e.g., by means of self-checking mechanisms) or external to the node computer. While, in general, the first approach is more cost-efficient, the fault coverage of the second approach is higher but it is also more expensive because it requires both a separate oscillator and power supply.

Recently, in some star topology configurations of TTP/C [BKS03], and as an option in FlexRay [Con04b], the bus guardianship functionality can be moved to a central hub. The obvious advantages of a central bus guardian are the price factor and the possibility to isolate a faulty branch from the network. The centralized bus guardian is a fully independent fault containment unit but it is at the same time a single point of failure. The duplication of the central star hub may overcome this issue, however details about this feature on both TTP/C and FlexRay are not available yet.

This recent trend towards star topologies is somewhat conflicting with one of the most important benefits of fieldbuses: the reduced wiring harness. However, it favors dependability because it becomes possible to isolate a defective branch from the network.

2.3.5 Replica determinism

A common approach to building fault-tolerant distributed systems is to replicate subsystems (the servers) that fail independently. The objective is to give other subsystems (the

clients) the illusion of service that is provided by a single server. There are two main groups of replication protocols and derivatives to enforce consistency between replicas in distributed systems: active and passive replication.

In active replication, also called state machine approach [Sch90], every replica, in parallel, receives and processes the same sequence of client requests and sends back the reply. Consistency is enforced because when fed with the same inputs in the same order (usually using atomic broadcast [WPS+00]), replicas will produce the same output. The requests are handled independently but must be processed in a deterministic way. This method is simple and possible node failures are transparent to the clients because other replicas also process the requests. However the determinism constraint may be difficult to enforce (e.g. in a multi-threaded node).

In passive replication, also known as primary backup [BMST93], clients send their requests to the primary replica that is responsible for processing them and returning the responses back to the clients. The backup replicas only interact with the primary and apply the respective updates. No determinism constraint is necessary but special care must be put on the mechanisms that enforce agreement between primary and backups (usually a membership service). A failure in the primary before sending the reply to the client cannot be masked by passive replication. In this case the client will time-out and, after having identified the new primary, re-issue the request. This significantly increases response time, making this protocol unsuitable for some time sensitive applications.

Two variants of these replication protocols are semi-active and semi-passive replication. In the former, the replicas do not need to process client requests in a deterministic way. Each time replicas need to make non-deterministic choices, one of them (leader) makes the decision and informs the others (followers). In semi-passive replication [DSS98], the client sends its request to all replicas and every replica sends a response back to the client. Thus, the client does not need to know the identity of the primary, neither the client needs to have time-outs to detect the crash of the primary. Therefore the effect of failures is completely masked to the client. Semi-passive replication is fully based on failure detectors and thus it does not require a membership service.

Replication and real-time

Most of the work on synchronous and asynchronous replication protocols has been mostly focused on applications for which real-time behavior was not a fundamental requirement. However, real-time applications operate under strict timing and dependability constraints. Hence, the problem of server replication poses additional challenges in a real-time environment.

Several experimental projects have addressed the problem of replication in real-time systems, e.g., the Time-Triggered Protocol (TTP) [KG94], RTCast [ASJS96], the real-time publisher/subscriber model [RGS95], the Window-Consistent Replication Service [MRJ97] and the Real-Time Primary-Backup (RTPB) [ZJ98].

TTP uses a time-triggered scheme to provide predictable immediate message delivery, membership service, and redundancy management in fault-tolerant real-time systems. The design of TTP is simplified by assuming that messages sent are either received by all correct destinations or no destination at all (which is reasonable for the redundant bus used in TTP). Also the architecture is based on the assumption that worst-case load is determined*a priori* at design time and the system response to external events is cyclic at pre-defined time intervals.

RTCast is a lightweight fault-tolerant multicast and membership service for real-time process groups which exchange periodic and aperiodic messages. The service supports bounded time message transport, atomicity and order for multicasts within a group of communicating processes in the presence of processor crashes and communication failures. It guarantees agreement on membership among the communicating processes, and ensures that membership changes resulting from joining or departing processors are atomic and ordered. Both TTP and RTCast are based on active replication.

The publisher/subscriber model for distributed real-time systems, presented by Rajkumar *et al* [RGS95], provides a simple user interface for publishing messages on a logical*channel* and for subscribing to selected channels as needed by each application. In the absence of faults each message sent by a publisher on a channel should be received by all subscribers. The abstraction hides a portable, analyzable, scalable and efficient mechanism for group communication. It does not, however, attempt to guarantee atomicity and order in the presence of failures, which may compromise consistency.

The Window-Consistent Replication Service, presented by Mehra*et al* [MRJ97], adopts a passive replication scheme in the scope of process control systems, where upon the primary backup failure the system switches to the backup node within a few hundred milliseconds. In that time there can be hundreds of updates to the data repository. This makes it impractical, and perhaps impossible, to update the backup synchronously each time the primary repository changes. The proposed alternative exploits the data semantics by allowing the backup to maintain an older copy of the data that resides on the primary. The application may have distinct tolerances for the staleness of different data objects. With sufficiently recent data, the backup can safely supplant a failed primary; the backup can then reconstruct a consistent system state by extrapolating from previous values and new sensor readings. However, the system must ensure that the difference between the primary and the backup data is bounded within a predefined time window. Data objects may have distinct tolerances in how far the backup can lag behind before the object state becomes stale. This protocol bounds the distance between the primary and the backup such that consistency is not compromised, while minimizing the overhead in exchanging messages between the primary and its backup.

Roughly speaking, this protocol works as follows: a client application registers a data object declaring the consistency requirements for the data in terms of a time window. The primary selectively transmits to the backup, as opposed to sending an update every time an object changes, bounding both resource utilization and data inconsistency. The primary

ensures that each backup site maintains a version of the object that was valid on the primary within the preceding time window by *scheduling* these update messages.

Finally, RTPB replication scheme builds upon the previous Window-Consistent replication protocol by proposing a more general temporal consistency model and an inter-object temporal consistency.

2.3.6 Membership

A *membership service* is a service used in a distributed system to maintain information about which sites are functioning and which have failed at any given time [HS95]. The membership problem is a fundamental problem of distributed computing like clock synchronization and atomic broadcast, in the sense that once solved, it allows easy solutions to other important problems encountered when designing fault-tolerant distributed applications.

A group membership protocol manages the formation and maintenance of a set of processes called a *group*. For example, a group may be a set of processes that are cooperating towards a common task, e.g., the primary and backup servers of a database. In general, a process may leave a group because it failed, it voluntarily requested to leave, or it is expelled by other members of the group. Similarly, a process may join a group; for example, it may have been selected to replace a process that has recently left the group. A group membership protocol must manage such dynamic changes in some coherent way.

There are numerous group membership protocols proposed in the literature for synchronous systems, following the initial work of Cristian [Cri91]. However, this problem cannot be solved in asynchronous systems with crash failures [CHTCB96].

2.3.7 Faults and fault models

A fault in the channel is one event that affects any element of the physical layer of the network (cable, connectors, transceiver's circuitry, etc.). This fault can have either internal or external origin with respect to its location in the system.

Internal faults represent the malfunctioning/damaged parts of a system that induce errors. These faults occur due to physical defects during manufacture or due to component aging. Most internal faults are likely to be permanent or intermittent because the effects of physical defects (e.g., broken, short, or loose connections) tend to persist or cycle between active and inactive states.

External faults, on the other hand, result from environmental interferences or disruptions, such as electromagnetic perturbation, radiation, temperature, or vibration. These external faults can be transient because disruptive environmental conditions, e.g.,*electromagnetic interference* (EMI) are temporary and may cause functional error modes without actual component damaging.

Techniques to minimize both internal and external induced faults, such as cable screening,

differential signaling and robust connectors have been widely used. However, these techniques are not perfect and do not fully prevent faults to occur. The number and the impact of these faults depends on the cabling harness, the environment and on a range of other possible factors. The inherently stochastic nature of the sources of EMI, as lightning, radar, mobile phones, voltage switching and so on, makes it very difficult to measure or predict the effects of EMI. In fact, EMI induced faults cannot be predicted with any accuracy [EE00] in an unstructured environment. Nevertheless, statistics representing the effect of the most common sources of EMI can be collected, helping to better characterize and anticipate the effect of EMI (and other sources) induced faults in the system. The same reasoning can be applied to the sources of internal faults. However, to be analyzable a bus system requires a model of the expected faults, otherwise no dependability guarantees could be expected. This calls for the definition of some assumptions concerning the number and type of faults, i.e., the fault model.

There have been several attempts to model faults in bus systems. Tindell *et al.* [TBW95] presented a **deterministic** fault model for the CAN bus that enables calculating an upper bound for frame response times on CAN in the presence of faults. Tindell's error model is based in the assumption that the number of errors can be upper bounded during a given time period and faults were treated as sporadic single-bit faults with a minimum separation between faults. Tindell's model is simple and it is useful to compute the bandwidth overhead required to meet the deadlines, however it fails to mimic the nature of real faults. As a result the frame response time overhead penalty is too high and not necessarily justified in practical cases [BBRN04].

Some of these shortcomings were removed by Punnekkat *et al.* [PHN00] by providing a more general fault model which can deal with interference caused by several specific sporadic sources. The framework considers the most predominant forms of interference in a given application (as mobile phones and radars) and assumes that specific patterns of interference can be derived by collecting error statistics. The patterns are characterized by an initial burst of faults and then a sporadic distribution of faults with a known minimum inter-arrival time. It is a form of **bounded** model. The main advantage of this approach is the ability to model specific interference patterns. However, as all bounded fault models, it is difficult to have absolute confidence in the accuracy of the model and it can easily lead to a poor bandwidth utilization.

Rufino *et al.* [RVA+98] proposed an alternative way to model faults in CAN, not considering the faults at the physical layer directly but instead modeling them at a higher level. This fault model defines an omission degree assumption that at most n retransmissions of that frame are required to deliver a frame. This model has the following properties:

- Bounded Omission Degree: in a known time interval, omission failures may occur in at most k transmissions.

- Bounded Inaccessibility: in a known time interval, the network may be inaccessible at

most i times, with a total duration of at most T_{ina}.

This model is simpler than the previous and also requires error statistics to tune its parameters.

The three frameworks previously presented use models based on either a minimum inter-arrival time between faults or on a bounded omission degree. Therefore they assume that the overhead of faults that can occur is **bounded** and hence it is possible to compute worst case response time analysis in a deterministic way.

However, as referred before, faults are random and cannot be reliably characterized by a bounded model. This leads to two problems [Bro03]:

- In order to have confidence that the analysis covers the worst case fault conditions, the worst case overhead due to faults must be set very high (to guarantee all deadlines); this results in so much spare bandwidth being reserved that there is very little available for normal messages.
- At run time, there is no guarantee that the faults will actually conform to the assumptions used for analysis.

Some authors [KS94][NYQS00][BPSW99a], however, realized that faults are a random phenomenon and in essence, a random phenomenon tends to better obey probabilistic laws rather than deterministic ones. In this way, faults are modeled as a random pulse sequence with an exponential distribution of inter-arrival times, forming a Poisson distribution [KS94]. According to this **probabilistic** model, there is neither a guarantee on the minimum separation between faults, nor on the number of faults within an interval. Faults are assumed to occur following a Poisson distribution (assuming independent faults), so in any given interval, there is a non-zero probability of any number of faults occurring. In such**unbounded** model, a worst-case response time analysis is not possible if one wants to have absolute confidence in the results.

Navet *et al.* [NYQS00] adopted an unbounded model for CAN, that considers not only the frequency of faults, but the duration of the faults. Both the frequency and gravity are considered to follow Poisson distributions, which allows the overhead of the faults to be considered as a generalized Poisson distribution. The probabilistic response time analysis for CAN, proposed by Navet *et al.*, considers only the number of faults necessary to cause a failure independently of how close they must occur. The computation of the response time analysis is made in two steps:

1. Initial analysis, using the scheduling analysis of Tindell to calculate the maximum number of faults K_i that can be tolerated for each message before the deadline is reached, considering that each fault generates a known maximum overhead which extends the response time.

2. Once the maximum number of faults (K_i) and the worst case response time that this would generate are obtained, they are used in the second stage of the analysis with the fault model to find the probability that a message may miss its deadline.

In this way, the probability of any given frame missing its deadline is known, and it is called Worst Case Deadline Failure Probability (WCDFP). Since the fault model assumed by Navet is a generalized Poisson process, the WCDFP can be analytically calculated fromK_i.

Navet's model analysis is more complex than the previous, the analysis includes a number of sources of pessimism in the estimation of the WCDFP [BBRN02] and it also requires the value of three parameters that depend on the environment in which the system is used. These parameters should be tuned during the design phase of the system based on real error statistics. In this way, Navet's model strongly relies on real CAN error statistics which are seldom available in the literature. In fact, up to our best knowledge, there are no public CAN error statistics available apart from our recent work [FOFF04], which will be presented later on.

In his work, Broster *et al.* [BBRN02] highlighted several sources of pessimism in Navet's work and presented a new analysis that provides a probability distribution of worst case response times under a random arrival (Poisson) fault model, to compute the probability of deadline failures.

The general approach is to produce a probability tree of feasible scenarios starting from a critical instant and then traversing the tree to calculate a distribution of response times. Branch pruning, based on a threshold parameter for insignificant probabilities, limits the size of the tree. This approach however, is rather computationally intensive and does not necessarily cover all the search space. A more efficient analysis was presented in [BBRN04].

The previously referred fault models and analysis of the probability of deadline failures are targeted to CAN. One key feature of CAN is that if a fault occurs during transmission of a frame, then the frame is automatically queued for retransmission. This can generally be used to provide an assured delivery service but can also be the cause of uncertainty in delivery timing since the number and distribution of faults determines the time at which a frame is received. In contrast, other buses based on a time-triggered paradigm (e.g., ARINC-629, TTP/C and FlexRay) do not respond to corrupted frames other than disregarding them. These protocols leave all the concerns of absent data to the application. The result of this is that the timeliness of the bus is preserved, but the number of messages that are lost is directly proportional to the number and distribution of faults that occur.

Summarizing, with bounded models it is possible to precisely compute response times, thus guaranteing message delivering timeliness, at the cost of a bandwidth overhead penalty necessary to cover fault scenarios assumptions. However, this does not guarantee that, at run time, faults will actually comply with the considered fault model. In contrast, unbounded fault models do not allow a precise response time analysis computation and the analysis itself

is more complex than in the case of bounded models. However, these models are more realistic and may lead to a reduced bandwidth overhead.

2.4 Conclusion

This Chapter presented some background information concerning real-time communication in shared media and dependability in distributed systems. The concepts addressed in this Chapter are the basis to understand the remainder of this dissertation.

In the case of bus based real-time communications, the bus is a shared medium between all nodes and inter-node communication must be carried out within a bounded time and, thus, it is imperative that the access to the bus is also bounded in time, which implies that the medium access control (MAC) protocols must be deterministic.

A survey of the two main classes of MAC protocols, controlled and uncontrolled access, was presented. In the former there is a distributed knowledge in the network concerning the access rights to the bus, either based in the time or in explicit commands of a bus master, that prevents the nodes to transmit messages simultaneously. Two sub categories are normally considered within this group: centralized(Master-Slave) and distributed control (Token-Passing, Virtual Token-Passing, FTDMA, TDMA). In the latter category, uncontrolled access (CSMA, CSMA-CD, CSMA-BA and P-Persistent CSMA), there is no distributed knowledge concerning the bus access, so every node may attempt to transmit at any given instant and possible collisions are detected and handled to prioritize access.

This Chapter has also addressed, at an introductory level, the issues of dependability in the context of real-time communications. Particularly it focused on the basic dependability and fault management concepts and presented a brief overview of some relevant topics and results in the area of distributed consensus and fault-tolerant broadcasts. Subsequently, the fail-silence failure mode was discussed together with the bus guardian concept, and the issues of replication in the real-time context. Finally this Chapter concluded with a discussion of several fault models.

Chapter 3

Flexibility and safety of some bus protocols

3.1 Introduction

Generically, the term *flexibility* is associated with the ability to adapt to new circunstances/events. It can, obviously, be applied in many different contexts and thus, when talking about flexibility, it is important to define exactly to which context one is referring to. In the context of communication systems one can refer to aspects such as the ability to use different physical media, topologies, bit encoding, the ability to support live insertion/removal of nodes, the capacity to support on-line mode changes as well as fast download of new application software, and the ability to support dynamic communication requirements. Basically, the more options the system supports, the more flexible it is. From the previous listing one can distinguish two main classes of flexibility attributes: pre-run time (e.g., physical media, topology, bit-encoding and mode-changes) and dynamic (e.g., live insertion/removal of nodes and dynamic communication requirements). Pre-run time flexibility attributes enlarge the system designer's solutions space [BA04] and may even contribute to optimized designs in terms of dependability and timeliness.

This is not the case of dynamic flexibility attributes that deal with evolving requirements, thus possibly leading to unpredictable and possibly unsafe operating scenarios, if adequate precautions are not taken. This is the reason why flexibility and safety have been considered conflicting concepts [Kop97] and, therefore, there is a widespread belief that critical safety implies a fully static system, meaning that all operating conditions must be completely defined at pre-run time favoring the design of fault tolerance mechanisms and the certification of the resulting systems. This is emphasized in the aviation industry standard DO-255 [RTC00], which states the following requirement for airborne computer systems:

"The ACR [Avionics Computer Resource] shall include internal hardware and soft-

ware management methods as necessary to ensure that time, space and I/O allocations are deterministic and static."

The issue of dynamic flexibility attributes becomes even more sensitive when the system is distributed, with several nodes exchanging messages over a communication network. For example, failures in message delivery or in remote nodes, even temporary, make it more difficult to assure a coherent notion of the global system state. Consequently, if on-line changes are allowed in the operating parameters of the system, some nodes may not perceive these changes equally, thus leading to inconsistency errors.

Furthermore, if the operating parameters can change on-line, unboundedly and uncontrolled, it is not possible to use *a priori* knowledge to distinguish correct system states from incorrect ones. The use of *a priori* knowledge is of capital importance in safety-critical applications to distinguish between what is correct and what is wrong. *A priori* knowledge is maximized when a system is fully static (or a system that supports a number of pre-defined modes), since once the system is designed it will be always possible to anticipate what will happen in a response to a given event.

However, it is also commonly accepted that flexibility is a desired property in a system in order to support evolving requirements, simplify maintenance and repair, and improve the efficiency in using system resources. This last aspect is particularly related to operational flexibility and it basically corresponds to use only the resources that are effectively required at each instant. This efficiency might impact positively on the system cost because with the same resources one can add more functionality or one can offer the same functionality with fewer resources. This is particularly interesting to cost-sensitive industries, such as the automotive industry. Also, flexibility can be used to achieve a better behavior under exceptional circumstances such as faults and overloads [Le 92], favoring dependability.

A system supporting dynamic flexibility attributes requires a*quasi-static* notion of *a priori* knowledge, i.e., a time interval during which system's attributes are fixed. Transitions between these time intervals, in case of a request to change system's parameters, require filtering the requests in order to accept only those that conform to a previously defined set of possibilities so that the continued safe and timely behavior of the system is guaranteed.

Therefore, despite its possible difficulty, it seems worth exploring ways to add operational flexibility without jeopardizing system safety.

Flexible and dependable real-time communication is the key issue of this thesis. This Chapter presents an overview of relevant topics in this field, including a discussion of flexibility *versus* dependability in several communication protocols and architectures, namely Controller Area Network (CAN) and CAN based protocols, TTP/C, FlexRay and ARINC-629.

3.2 CAN and CAN related protocols

Controller Area Network (CAN) is a popular and very well-known bus system, both in academia and in industry, initially targeted to automotive applications as a single digital bus to replace the wiring harnesses that were growing in complexity, weight and cost with the advent of new electrical and electronic appliances in vehicles.

The widespread and successful use of CAN in the automotive industry, the low cost associated with high volume production of controllers and CAN's inherent technical merit, have driven to CAN adoption in other application domains such as: industrial communications, medical equipment, machine tool, robotics and in distributed embedded systems in general.

The large installed base of CAN nodes (over 10^9 according to [iA]) with low failure rates over almost two decades, led to the use of CAN in some critical applications such as Anti-locking Brake Systems (ABS) and Electronic Stability Program (ESP) in cars. In parallel with the wide dissemination of CAN in industry, the academia also devoted a large effort to CAN analysis and research, making CAN one the must studied fieldbuses.

The widespread use of CAN in safety-critical applications is still, however, an open issue with some arguing that it is not suitable [Kop98] while others argue that it may be adopted in some applications if some precautions are adopted [Bro03][PF04]. The ones arguing against the use of CAN come mostly from the dependable systems community where the dogma of fixed time-triggered communication schedules prevails. It is clear that the *a priori* knowledge of the communication schedule favors error detection and simplifies the fault-tolerance mechanisms and the certification process, however the ones in favor of CAN use in safety-critical applications argue that CAN inherent flexibility can help reacting to transient overloads (e.g., due to electromagnetic interference), also that CAN may accommodate some kind of time-triggered behavior and finally that CAN has proved, over the years, its merits with low failure rates.

In an effort to adapt CAN to the requirements of safety-critical applications (e.g., X-by-wire), an ISO task force defined a protocol for a time-triggered transmission of CAN messages (TT-CAN). Because the CAN protocol remains unchanged, it is possible to transmit both time-triggered and event-triggered messages via the same physical bus system. According to some authors [BBRN04][RP03][FOFF04], the efforts to make a CAN extension more suitable for safety-critical applications, were not fully successful. In fact, according to these authors, native CAN is more robust than TTCAN.

This thesis argues that it is possible to provide a bounded degree of flexibility without compromising dependability and proposes some mechanisms to building CAN based safety-critical systems. The main focus of this work is the Flexible Time-Triggered CAN protocol: a protocol that combines the predictability of time-triggered systems, favoring the design of fault-tolerance mechanisms, and the flexibility of CAN, favoring the adaptation to evolving conditions.

In this context, a description of some relevant details of CAN is required to give the reader sufficient knowledge of the protocol to understand subsequent Chapters. However, it is beyond the objectives of this thesis to present a deep description of the CAN protocol. Specification documents [ISO93][BOS91] should be referred for a clearer and detailed description.

Over the years, several protocols based in CAN were presented. These protocols take advantage of some CAN properties and try to improve some known CAN drawbacks. An overview of CAN and some relevant CAN related protocols (TCAN, FlexCAN, FTT-CAN and TTCAN) will be present next.

3.2.1 CAN

CAN protocol was introduced in the mid eighties by Robert Bosch GmbH [BOS91] and it was internationally standardized in 1993 as ISO 11898-1 [ISO93]. CAN provides two layers of the stack of the Open Systems Interconnection (OSI) reference model: the physical layer the data link layer and optionally an additional application layer (not standardized). Notice that CAN physical layer was not defined in Bosch original specification, only the data link layer was defined. However, the CAN ISO specification filled this gap and the physical layer was then fully specified. This collapsed OSI model is common to most of the fieldbuses, where application services access the data link layer directly. Some higher-level ISO stack protocols, e.g. CAN Kingdom [Fre95] and CANOpen [CiA02] are also defined and used in some applications.

CAN is a message-oriented transmission protocol, i.e., it defines message contents rather than nodes and node addresses. Every message has an associated message identifier, which is unique within the whole network, defining both the content and the priority of the message.

Network topology

CAN network topology is bus based. Replicated busses are not referred in CAN standard, however it is possible to implement them [RVA99a].

Over the years, some star topologies for CAN have been proposed [Ruc94][IXX05][BRNPA04]. Cena [CDV01] also proposed a passive star for CAN, however it does not comply with CAN standard and thus it will not be considered further. Solution presented in [IXX05], is based in passive star network topology and rely on the use of a central element, the star coupler, to concentrate all the incoming signals. The result of this coupling is then broadcast to the nodes. The solution presented by Rucks [Ruc94] is based on an active star coupler capable of receiving the incoming signals from the nodes bit by bit, implements a logical AND, and retransmits the result to all nodes. None of these solutions is capable of disconnecting a defecting branch and so, from the dependability point of view they only tolerate spatial proximity faults.

Recently, Barranco *et al.* [BRNPA04] proposed a promising solution based in an active hub that is able to isolate defective nodes from the network. This active star is compliant with

CAN standard and allows detecting a variety of faults (stuck-at node fault, shorted medium fault, medium partition fault and bit-flipping fault) that will cause the faulty node to be isolated.

Physical Layer

The CAN ISO standard, incorporates the original Bosch specifications [BOS91] as well as part of the physical layer, the physical signaling, the bit timing and synchronization. There are a small number of other CAN physical layer specifications including:

ISO 11898-2 High Speed – ISO 11898-2 is the most used physical layer standard for CAN networks. In this standard the data rate is defined up to 1 Mbit/s with a theoretically possible bus length of 40 m at 1 Mbit/s.

ISO 11898-3 Fault-Tolerant – This standard defines data rates up to 125 kbit/s with the maximum bus length depending on the data rate used and the bus load. Up to 32 nodes per network are specified. The fault-tolerant transceivers support the complete error management including the detection of bus errors and automatic switching to asymmetrical signal transmission.

SAE J2411 Single Wire – An unshielded single wire is defined as the bus medium and the communication takes place with a nominal data rate of 33,3 kbit/s The standard defines up to 32 nodes per network. The main application area of this standard is in comfort electronics networks in vehicles.

ISO 11992 Point-to-Point – This standard defines a point-to-point connection for use mainly in vehicles with trailers. The nominal data rate is 125 kbit/s with a maximum bus line length of 40 m.

The most popular CAN physical layer protocol, available in most of the CAN transceivers, is the one defined in the ISO 11898-2 standard [ISO93]. So, this physical layer protocol will be the one adopted throughout this thesis.

The CAN physical layer provides a 2-state medium: the *dominant* and the *recessive*. Whenever two nodes simultaneously transmit bits of opposite value, then all nodes should read dominant. Usually the dominant state is associated with the binary value 0 and recessive with the binary value 1.

The physical layer has a number of built-in fault-tolerant features. CAN provides resilience against a variety of physical faults such as one open wire, the short-circuit of the two signal wires, or even one of the signal wires shorted to ground or power. Notice that not all CAN controllers are able to implement these features, by switching from differential signaling to single line signalling, at higher bus speeds. The CAN differential electrical signaling mode is very resistant to electromagnetic interference (EMI) since interference will tend to affect each

side of a differential signal almost equally. However, the differential electrical signalling does not fully prevent electromagnetic interference to affect the signal on the bus in such a way that one or more nodes on the bus will simultaneously read a different bit value from that which was transmitted. A node detecting the error (possibly the transmitter) will invalidate the message by transmitting an error frame. The number and the nature of EMI induced transmission faults and the ability of the physical layer to prevent them depends on several factors as the cables type and length, the number of nodes, the transceiver type, the EMI shielding, etc. Also, as it was referred in the previous Chapter, the intrinsic nature of the EMI phenomena makes it very hard to predict and model.

Data Link Layer

The data link layer of CAN includes the services and protocols required to assure a correct transfer of information from one node to another.

There are four types of message on CAN: data frames, error frames, remote transmission request frames and overload frames. The latter two types of messages are rarely used in real application and, thus, will not be described further.

The CAN protocol supports two message frame formats, the only essential difference being in the length of the identifier. The CAN *base frame* format supports a length of 11 bits for the identifier (formerly known as CAN 2.0 A), and the CAN *extended frame* format supports a length of 29 bits for the identifier (formerly known as CAN 2.0 B). The CAN base format is sufficient for most applications and will be the format adopted in this dissertation.

Data on the bus is sent in data frames which consist of up to 8 bytes of data plus a header and a footer. The frame is structured as a number of fields, as depicted in Figure3.1.

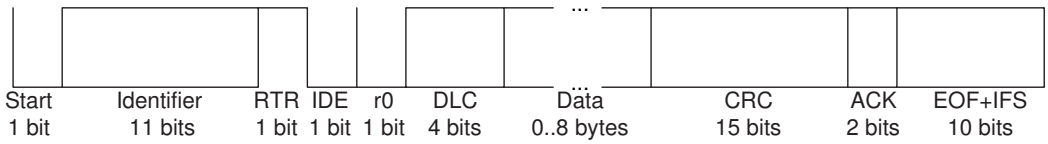


Figure 3.1: CAN base frame format.

A CAN base frame message begins with the start bit called *Start Of Frame* (SOF). This bit is followed by the *arbitration field* which consist of the identifier and the *Remote Transmission Request* (RTR) bit used to distinguish between the data frame and the data request frame called remote frame. The following *control field* contains the *IDentifier Extension* (IDE) bit to distinguish between the CAN base frame and the CAN extended frame, as well as the Data Length Code (DLC) used to indicate the number of following data bytes in the *data field*. If the message is used as a remote frame, the DLC contains the number of requested data bytes. The data field that follows is able to hold up to 8 bytes. The integrity of the frame is

guaranteed by the following *Cyclic Redundant Check* (CRC) sum. The *ACKnowledge* (ACK) field comprises the ACK slot and the ACK delimiter. The bit in the ACK slot is sent as a recessive bit and is overwritten as a dominant bit by those receivers which have at this time received the data correctly. Correct messages are acknowledged by the receivers regardless of the result of the acceptance test. The end of the message is indicated by *End Of Frame* (EOF). The *Intermission Frame Space* (IFS) is the minimum time in equivalent number of bits separating consecutive messages. Unless another station starts transmitting, the bus remains idle after this.

Associated with every CAN message there is a unique message identifier that defines its content and also the priority of the message. Bus access conflicts are resolved by a non-destructive bitwise arbitration scheme where the identifiers of the involved messages are observed bit-by-bit by all nodes, in accordance with the wired-AND mechanism, by which the dominant state overwrites the recessive state. All those nodes with recessive transmission and dominant observation lose the competition for bus access. The nodes that lost the arbitration automatically become receivers of the message with the highest priority and do not re-attempt transmission until the bus is available again. In this way, transmission requests are handled in order of their importance for the system as a whole.

Detecting and signalling errors

A CAN node does not acknowledge message reception, instead it signals errors immediately as they occur. For error detection the CAN protocol implements three mechanisms at the message level:

Cyclic Redundancy Check – This mechanism accounts for message corruption. The transmitter node computes the CRC and transmits it within the message. The receiver decodes the message and recomputes the CRC and if they do not match, there has been a CRC error.

Frame check – This mechanism detects message format violations, i.e., it checks each field against the fixed format and the frame size.

Acknowledge errors – Since the receivers of a message must issue an acknowledgement bit in the ACK field, if the transmitter does not receive an acknowledgement an ACK error is indicated, thus allowing a node to detect isolation from the network.

Besides error detection at the message level, the CAN protocol also implements mechanisms for error detection at the bit level:

Transmission monitoring – Each node transmitting a message also monitors the bus level to be able to detect differences between the bit sent and the bit received. This mechanism allows distinguishing global errors from errors local to the transmitter only.

Bit stuffing – CAN uses a non return to zero codification to prevent nodes from loosing synchronization by receiving long sequences of recessive or dominant bits. A supplementary bit is inserted by the transmitter into the bitstream after five consecutive equal bits. The stuff bits are removed by the receivers that also detect violations of the bit stuffing rule.

If at least one station detects any error, it will start transmitting an error frame in the next bit aborting the current message transmission. This prevents other stations from accepting the message and thus ensures the consistency of data throughout the network. After transmission of an erroneous message that has been aborted, the sender automatically re-attempts transmission (automatic retransmission). During the new arbitration process all nodes compete for bus access and the one with the higher priority message will win arbitration. Thus the message affected by the error could be delayed. There is, however, a special case where consistency throughout the network is compromised. This issue, will be detailed in Chapter 4.

To prevent a faulty CAN controller to abort all transmissions, including the correct ones, the CAN protocol provides a mechanism to distinguish sporadic errors from permanent errors and local failures at the station. This is done by statistical assessment of station error situations with the aim of recognizing a station's own defects and possibly entering an operation mode in which the rest of the CAN network is not negatively affected. This may go as far as the station switching itself off (bus-off state). This behavior will be described in Chapter4.

Bus guardianship

CAN protocol specification does not mention using bus guardianship functionality to protect the network against a *babbling idiot* node. As it will be detailed in Chapter 4, the CAN built-in fault containment mechanisms, capable of limiting the impact or even isolating from the network faulty nodes (error passive, bus-off), do not perform adequately in some scenarios or when they do, the time to bus-off could be too high for some applications.

This motivated the development of independent bus guardian for CAN by Tindell and Hansson [TH95] and by Broster and Burns [BB03]. Tindell and Hansson suggested a specialized hardware driver to provide a robust defence against software failures leading to*babbling idiot* failure. Broster and Burns proposed an external bus guardian (a completely independent node on the bus) capable of detecting most of*babbling* failures. Although the proposed guardian cannot detect all *babbling* failures, the worst case *babbling* that cannot be detected by a guardian is identifiable. Also, the worst case response time for any frame taking into account the additional overhead from any undetected*babbling* is only slightly larger than the worst case response time if *babbling* cannot occur. Broster and Burns bus guardian operation is based in the minimum message inter transmission time of a message stream. If that condition is violated the faulty node is isolated from the bus.

Operational flexibility

The content-oriented addressing scheme adopted in CAN makes a CAN based system very flexible in terms of live insertion/removal of nodes. This can be done without making any hardware or software modifications, as long as the new nodes are purely receivers at the instant of insertion/removal.

Despite not being so predictable as a time-trigger communication bus, the inherent flexibility of an event-triggered communication bus such as CAN provides some advantages. It allows, for example, very efficient error correction mechanisms to tolerate channel faults such as electromagnetic interference which causes corruption of bits on the bus. In fact, Broster*et al.* have shown [BBRN02] that the probability of timely delivery of messages in CAN with faults is higher than in TTCAN.

3.2.2 TTCAN

Time-Triggered Communication on CAN (TTCAN) [ISO01] is an extension of CAN, introducing time-triggered operation based on a high precision network-wide time base. The time-triggered communication is built on top of the unchanged standard CAN protocol, allowing a software implementation of TTCAN, using existing CAN controllers (level 1 only).

There are two possible levels in TTCAN, level 1 and level 2. Level 1 only provides time triggered operation using local time (Cycle_Time). Level 2 requires a hardware implementation and provides increased synchronization quality, global time and external clock synchronization. As native CAN, TTCAN is limited to a maximum data rate of 1 Mbit/s, with typical data efficiency below 50%.

Network Topology

TTCAN network topology is bus based. Replicated busses are not referred in TTCAN standard, however it is possible to implement them [MFH+02].

Message Transmission

TTCAN adopts a Time-division Multiple Access (TDMA) bus access scheme. The TDMA bandwidth allocation scheme divides the timeline into time slots, or time windows. Network nodes are assigned different slots to access the bus. The sequence of time slots allocated to nodes repeats according to a basic cycle. Several basic cycles are grouped together in a matrix cycle. All basic cycles have the same length, but can differ in their structure. When a matrix cycle finishes the transmission scheme starts over by repeating the matrix cycle (Figure3.2). The matrix cycle defines a message transmission schedule. However, a TTCAN node does not need to know the whole system matrix, it only needs information of the messages it will send and receive.

Since TTCAN is built on top of native CAN, some time windows may be reserved for several event messages. In such windows, it is possible that more than one transmitter may compete for the bus access right. During these slots the arbitration mechanism of the CAN protocol is used to prioritize the competing messages and to grant access to the higher priority one. In this sense, the medium access mechanism in TTCAN can be described as TDMA with Carrier Sense Multiple Access with Bitwise Arbitration (CSMA-BA) in some pre-defined time slots. This feature makes the TTCAN protocol as flexile as CAN during the arbitration windows, without compromising the overall system timeliness, i.e. the event-triggered messages do not interfere with the time-triggered ones.

The TDMA cycle starts with the transmission of a reference message from a time master. The reference messages are regular CAN messages with a special and known *a priori* identifier and are used to synchronize and calibrate the time bases of all nodes according to the time master's time base, providing a global time for the network.

TTCAN level 1 guarantees the time triggered operation of CAN based on the reference message of a time master. Fault-tolerance of that functionality is established by redundant time masters, the so called potential time masters. Level 2 establishes a globally synchronized time base and a continuous drift correction among the CAN controllers.

There are three types of time windows (Figure 3.2):

- **Exclusive time windows** – for periodic messages that are transmitted without competition for the CAN bus. No other message can be scheduled in the same window. The automatic retransmission, upon error, is not allowed.

- **Arbitrating time windows** – for event triggered messages, where several event-triggered messages may share the same time window and bus conflicts are resolved by the native CAN arbitration. Two or more consecutive arbitrating time windows can be merged. Message transmission can only be started if there is sufficient time remaining for the message to fit in. Automatic retransmission of CAN messages is disabled (except for merged arbitrating windows).

- **Free time windows** – reserved for future extensions of the network. A transmission schedule could reserve time windows for future use, either for new nodes or to assign existing nodes more bandwidth. Notice that a free time window can not be assigned to a message unless it is previously converted to either an exclusive or an arbitrating time window.

Bus Guardianship

The TTCAN standard draft claims that each TTCAN controller provides error detection and fail silent behavior. This assumption seems excessive concerning the fail silent behavior.

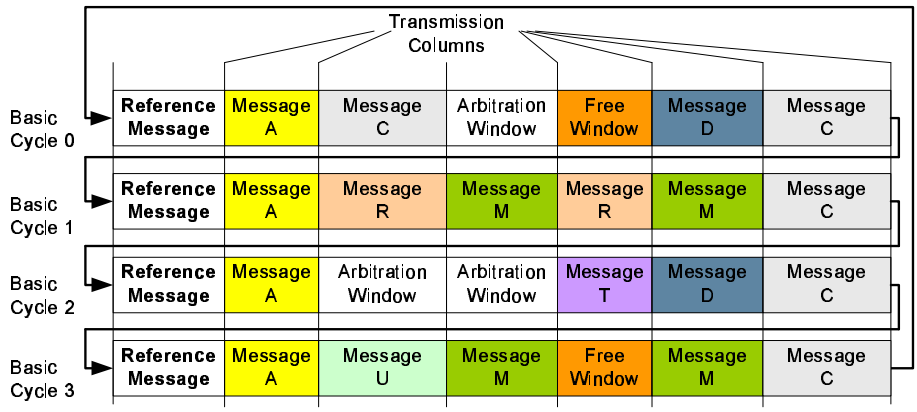


Figure 3.2: TTCAN system matrix, where several basic cycles build the matrix cycle (adapted from [FMD+00]).

Firstly, nothing is said about the fail silence nature (temporal domain, value domain or both), secondly, no bus guardians are used to provide fail silence behavior in the temporal domain. In fact, the following statement is made at Bosch's TTCAN homepage [Bos04]:

> *"Dedicated bus guardians are not needed in TTCAN nodes, bus conflicts between nodes are prevented by CAN's non-destructive bitwise arbitration mechanism and by CAN's fault confinement (error-passive, bus-off)!*

It seems clear that, without bus guardians, a TTCAN node may always, due to e.g. an application error, a corruption of the system matrix memory or to a fault on the local oscillator, transmit a message in an exclusive time window belonging to another node, violating thus, the fail silent assumption.

Nevertheless, errors are detected by the failure handling mechanisms of TTCAN and eventually all the transmissions from the faulty node will be disabled, leading to a fail silent behavior, but only after the occurrence of several errors (exact number depends on the error nature).

Clock Synchronization

Clock synchronization in a TTCAN network is provided by a time master. The time master establishes its own local time as global time by transmitting the reference message. To compensate for slightly different clock drifts in the TTCAN nodes and, to provide a consistent view of the global time, the nodes perform a drift compensation operation.

A unique time master would be a single point of failure, thus TTCAN provides a mechanism for time masters redundancy and replacement whenever the current time master fails to send a reference message. In this case, the CAN bus remains idle and any of the potential time

masters will try to transmit a reference message after a certain amount of time. In case two potential time masters try to send a reference message at the same time, the native CAN bit arbitration mechanism ensures that only the one with the highest priority wins. When a failed time master reconnects to the system with active time triggered communication, it waits until it is synchronized to the network before it may try to become time master again.

Error Detection

The strategy adopted, concerning fault confinement, is very similar to the one followed by CAN (ISO 11898-1), i.e., error passive and bus-off.

Since CAN failures are already considered in ISO 11898-1 standard (data link layer), TTCAN considers mainly scheduling errors (e.g. absence of a message) and each TTCAN controller only provides error detection. Active fault confinement is left to a higher layer or to the application.

Operational Flexibility

TTCAN may work under several operational modes: configuration mode, CAN communication, time-triggered communication or event synchronized time-triggered communication. However, operating modes may only change via configuration mode. The system matrix configuration may be read and written by the application during initialization, but it is locked during time-triggered communication, i.e., scheduling tables are locally implemented in every node and must be configured before system start-up. As a result the operation flexibility of TTCAN is quite limited, since the bus scheduling is static and defined at pre run-time.

3.2.3 FTT-CAN

The central proposition supported by this dissertation takes advantage of FTT-CAN to implement devices and mechanisms capable of providing bounded flexibility without compromising dependability.

The basis for the FTT-CAN protocol (Flexible Time-Triggered communication on CAN) has been first presented in [AFF98]. Basically, the protocol makes use of the dual-phase elementary cycle concept in order to combine time and event-triggered communication with temporal isolation. Moreover, the time-triggered traffic is scheduled on-line and centrally in a particular node called master. This feature facilitates the on-line admission control of dynamic requests for periodic communication because the respective requirements are held centrally in just one local database. With on-line admission control, the protocol supports the time-triggered traffic in a flexible way, under guaranteed timeliness. Furthermore, there is yet another feature that clearly distinguishes this protocol from other proposals concerning time-triggered communication on CAN [PD95][ISO01] that is the exploitation of its native distributed arbitration mechanism. In fact, the protocol relies on a relaxed master-slave

medium access control in which the same master message triggers the transmission of messages in several slaves simultaneously (master/multi-slave). The eventual collisions between slave's messages are handled by the native distributed arbitration of CAN.

The protocol also takes advantage of the native arbitration to handle event-triggered traffic in the same way as the original CAN protocol does. Particularly, there is no need for the master to poll the slaves for pending event-triggered requests. Slaves with pending requests may try to transmit immediately, as in normal CAN, but just within the respective phase of each elementary cycle. This scheme, similar to the arbitration windows in TTCAN, allows a very efficient combination of time and event-triggered traffic, particularly resulting in low communication overhead and shorter response times.

In FTT-CAN the bus time is slotted in consecutive Elementary Cycles (ECs) with fixed duration. All nodes are synchronized at the start of each EC by the reception of a particular message known as EC trigger message, which is sent by the master node.

Within each EC the protocol defines two consecutive windows, asynchronous and synchronous, that correspond to two separate phases (see figure 3.3). The former one is used to convey event-triggered traffic, herein called asynchronous because the respective transmission requests can be issued at any instant. The latter one is used to convey time-triggered traffic, herein called synchronous because its transmission occurs synchronously with the ECs. The synchronous window of the n^{th} EC has a duration that is set according to the traffic that is scheduled for it. The schedule for each EC is conveyed by the respective EC trigger message (see figure 3.4). Since this window is placed at the end of the EC, its starting instant is variable and it is also encoded in the respective EC trigger message. The protocol allows establishing a maximum duration for the synchronous windows and correspondingly a maximum bandwidth for that type of traffic. Consequently, a minimum bandwidth can be guaranteed for the asynchronous traffic.

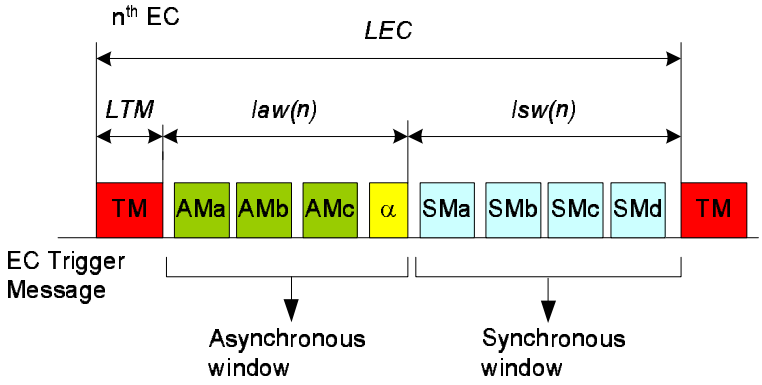


Figure 3.3: The Elementary Cycle (EC) in FTT-CAN.

In order to maintain the temporal properties of the synchronous traffic, such as com-

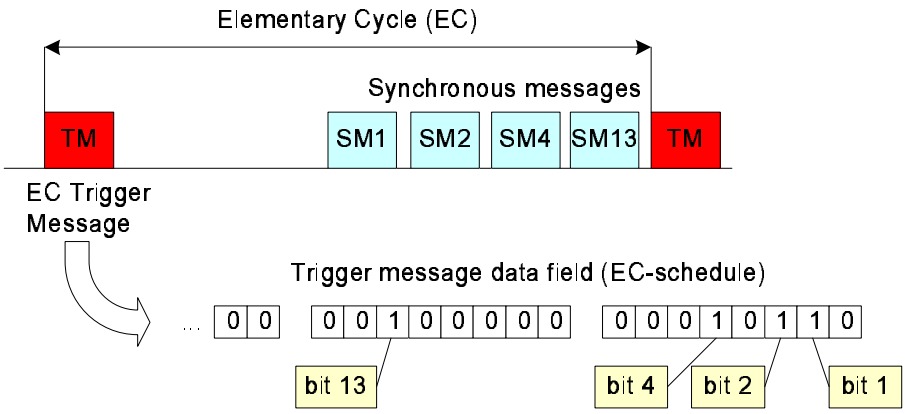


Figure 3.4: Master/multislave access control. Slaves produce synchronous messages according to an elementary-cycle schedule conveyed by the trigger message. If the x data bit is 1, then message x is produced in this EC; if it is 0, then message x is not produced.

posability with respect to the temporal behavior, it must be protected from the potential interference of asynchronous requests. Thus, a strict temporal isolation between both phases is enforced by preventing the start of transmissions that could not complete within the respective window. This is achieved by removing from the network controller transmission buffer any pending request that cannot be served up to completion within that interval, keeping it in the transmission queue. As a consequence, a short amount of idle-time may appear at the end of the asynchronous window (α in Figure 3.3). At the end of the synchronous window, another short amount of idle-time may appear. In this case, it is due to variations in the stuff-bits used in the physical encoding of CAN messages.

The communication requirements are held in a database located in the master node [Ped03], the System Requirements Database (SRDB). The SRDB holds the properties of each of the message streams to be conveyed by the system, both real-time and non-real-time, as well as a set of operational parameters related to system configuration and status. This information is stored in a set of three tables.

Synchronous Requirements Table (SRT) – This table contains the description of the periodic streams of messages. The relevant parameters are an identification, the data length in bytes, the corresponding maximum transmission time, the period, the deadline, a value that reflects the importance of the stream in the application, the attributes that indicate the operations that are acceptable over this stream and a list, or a range, of possible values for any specified parameter. The attributes allow limiting the level of flexibility in a per stream basis. They may specify that no changes are accepted for a given stream, or that the stream may be removed from the SRT, or that the period may assume a value in a given range, or one value from a list of possible values. On the other

hand, they may also specify that a stream is unconstrained.

Asynchronous Requirements – The Asynchronous Requirements component is composed by the reunion of two tables, the Asynchronous Requirements Table (ART) and the Non-Real-Time Requirements Table (NRT). The ART is used to store the properties of the asynchronous messages conveyed by the system that, despite being asynchronous, may or may not have timeliness requirements. For example alarm messages usually have hard timeliness requirements while messages used to perform remote diagnosis or configuration frequently do not have such timeliness constraints. The asynchronous messages are scheduled according to a best-effort policy, based on fixed priorities. The NRT stores the size of the longest non real-time message produced by each node, as required to enforce the temporal isolation between synchronous and asynchronous traffic.

Configuration and Status Record (SCSR) – This record stores all system configuration data, i.e., the bus transmission speed, duration of the elementary cycle, minimum amount of bandwidth allocated to asynchronous traffic, etc.

Based on the SRT, an on-line scheduler builds the synchronous schedules for each EC. These schedules are then inserted in the data area of the respective EC trigger message (see figure 3.4) and broadcast with it. Due to the on-line nature of the scheduling function, changes performed in the SRT at run-time will be reflected in the bus traffic within a bounded delay, resulting in a flexible behavior.

From an operational point of view, two different solutions have been used to implement the scheduler. One is the planning-scheduler [APF99], a software-based implementation that allows reducing the processing overhead of on-line scheduling. This technique consists on building a static schedule table for a given period of time into the future called plan and rebuilding that table on-line at the end of each plan. The plan duration is not correlated with the messages periods and thus the memory requirements to hold a plan table are bounded and known *a priori*. The planning scheduler is particularly well suited to systems with low computational capacity nodes (e.g. based on simple 8-bit microcontrollers). A negative feature of this technique is its lower responsiveness to changes in the communication requirements, when compared to normal on-line scheduling, arising from the static nature of each plan table. Notice that changes in the SRT, which holds those requirements, are taken into account from plan to plan, only. However, when the planning scheduler is used in the scope of FTT-CAN, the limitation on system responsiveness can be substantially reduced by using asynchronous messages to enforce the changes in communication requirements, temporarily, until they are handled by the planning scheduler [PAF00].

The second solution that has been developed to implement the scheduling function in FTT-CAN makes use of FPGA-based scheduling co-processors. This solution provides, at a higher hardware cost, the extra computational capacity required to execute both the scheduling policy

on-line as well as an adequate schedulability analysis. For example, the co-processor described in [MF01] scans the SRT and creates a new EC schedule every EC. Moreover, it is also capable of executing several schedulability tests in that interval. The result of this solution is a high degree of flexibility and responsiveness, plus a residual computational overhead, only, in the master processor.

3.2.4 Some emerging CAN based protocols

Several CAN based higher layer protocols have emerged over the years. Some of them are widely used in industry, e.g., CANopen, CAN Kingdom and DeviceNet. However, due to space limitations those protocols will not be presented and discussed in this work. Recently, two other CAN based protocols targeted to safety-critical systems have also been proposed: TCAN and FlexCAN. Since the scope of these new protocols is somewhat related with the main topic of this work, they will be presented in the next sections.

TCAN

The Timely-CAN (TCAN) [Bro03] protocol was developed based on the observation that if the message transmitter knows, when it starts to transmit a message, that it will arrive to the receivers after the deadline, then it is better to drop such a message (Figure3.5). Aborting a message in these circumstances can be considered a form of real-time error confinement, because no further message retransmissions will be attempted and the interference with other network traffic is bounded. According to this scheme, any messages that arrive will arrive in time. The bandwidth released by not transmitting messages that would arrive late can be used to *absorb* delays and hence help to provide timely delivery of other frames.

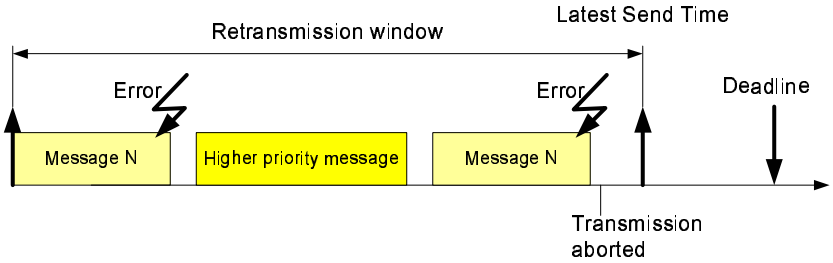


Figure 3.5: Typical TCAN message transmission scenario.

Another factor that has contributed for the development of TCAN was the unpredictable nature of faults that makes impossible the definition of off-line guarantees of timely behavior, even if an analysis based on fault models is used. In these cases, it is argued that losing a message should not be considered fatal and the system should continue to function as normally as possible without the message. Thus, the TCAN protocol relaxes the requirement for message delivery by assuming that some messages will not be delivered at all.

Each TCAN frame is associated with a delivery threshold time, termed the latest send time (LST), which is known *a priori* by all nodes. In this context, the role of message transmitters is to ensure that a frame is not transmitted after the threshold while the role of the receivers is to wait for receiving the frame only until the threshold. This, requires a common time base and thus this is also a time-triggered scheme with a relaxed definition of transmission instant, which can take place within a predefined window.

The TCAN protocol lies somewhere between CAN and TTCAN, since it takes advantage of CAN native robustness by allowing message retransmission within a specified window only, providing, thus, the timing predictability of TTCAN. The duration of the transmission window is determined by response time analysis.

Nevertheless, the TCAN protocol alone does not provide the kind of operational flexibility that is pursued in this work, i.e., the ability of changing the communication requirements online without compromising dependability. Implementing operational flexibility in TCAN requires a mechanism for controlling the changes of the communication requirements and to propagate those changes to all nodes.

FlexCAN and SafeCAN

FlexCAN is a recently proposed architecture [PF04] targeted for highly dependable safety-critical systems. FlexCAN is built upon a specialized protocol termed SafeCAN [PK04] that deals with error detection and management of replicated components, both buses and nodes. The FlexCAN architecture has been validated in a steer-by-wire system [Pim04].

FlexCAN provides a set of software components, called safeware, that enable end users to write safe applications without needing to design all the application from scratch. In this way, end users deal with simplified models because the details of the underlying protocol regarding hardware reconfiguration, replicated component management, fault and error detection and fault confinement, are hidden from them by SafeCAN.

There is no global clock in FlexCAN, message synchronization is done on an application basis. That is, any data source, e.g., sensors, controllers and actuators in a control loop, can be selected to produce time-triggered messages and all other network nodes should synchronize with the message sent by the data source. Thus, FlexCAN uses an event-triggered communication protocol (CAN) and a time-triggered synchronization protocol at the application layer.

FlexCAN deals mainly with the replication of communication channels and nodes. A FlexCAN fault-tolerant unit (FTU) is made of one node and 2 replicas (in a total of 3 nodes) and each node is connected to 3 CAN channels, resulting in 9 network interfaces. SafeCAN is responsible for error detection and node management in an FTU and manages the node components to make them appear as a single node to the application.

SafeCAN assumes a set of application oriented time-triggered cycles termed application cycles of length TS (Figure 3.6). The cycle is initiated by a data producer that sends a pro-

ducer message (PM). Message consumers (e.g., controllers) receive the message, perform some computations, and in turn may generate additional CAN messages for still other consumers (e.g., actuators). Consumers may also generate feedback messages (CM). The set of replicated components are termed primary, secondary, tertiary and so on. It is assumed that only one controller message goes on each CAN channel and this message (C-P) is sent by the primary controller. SafeCAN is responsible for selecting just one primary from a set of replicated components identified by a hardware serial number. All remaining components are designated as secondary, tertiary, etc. according to a ranking mechanism. SafeCAN relies basically on timers. If the primary controller does not send its message C-P after a delay τ_S then the secondary controller S assumes that the primary has failed and will switch to primary and output the C-P message as shown in Figure 3.6. If the secondary has also failed, the tertiary component will output the C-P message after a delay τ_T. This procedure is repeated for additional replicated components. Figure 3.6 also depicts the relative timing of the C-P and CM messages.

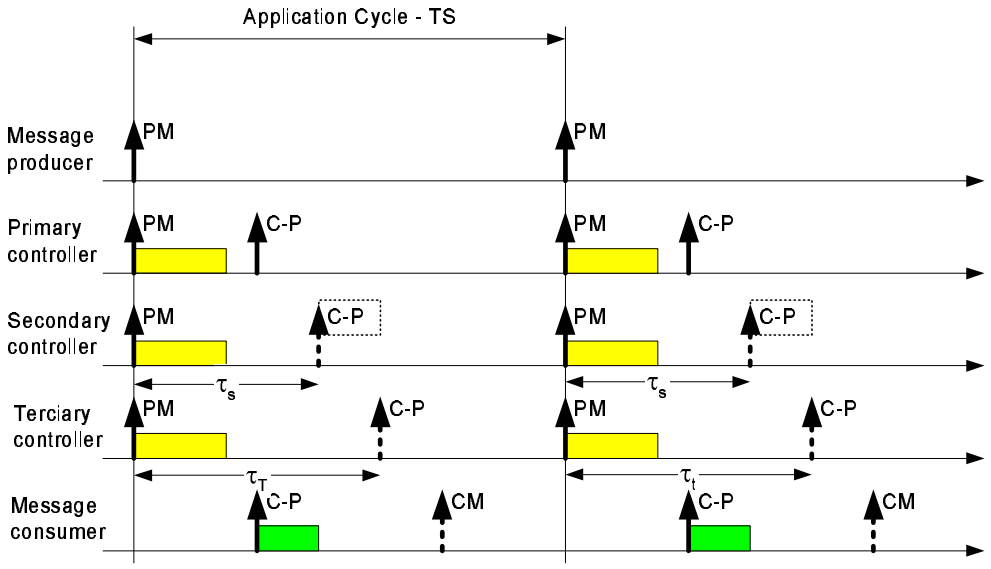


Figure 3.6: Timed synchronization of SafeCAN messages (adapted from [PF04]).

3.3 Time-Triggered Protocol

The Time-Triggered Protocol (TTP/C)[1] [TTT02] has been developed at the Technical University of Vienna in 1993 to be used in safety critical real-time systems. Commercial deployment of the protocol has been undertaken by TTTech since 1998. The main concern of

[1]The "C" stands for class C type systems, according to the Society of Automotive Engineers SAE classification.

the protocol designers was safety, second composability and third flexibility [KG94]. Although it is a communication protocol, its properties influence the design philosophy of the entire computer architecture: the Time-Triggered Architecture (TTA) [HK03].

The TTP/C protocol supports the use of Fault-tolerant Units (FTU) that consist of two or more identical replicas of each control computer. The protocol ensures that all operational replicas receive the same input data and thus are coherent. In each FTU, two replicas are active in sending messages while any additional nodes serve as shadow nodes that are passive until one of the active replicas fails and becomes silent.

A TTP/C node (Figure 3.7) has a host computer, responsible for executing the application software, a TTP/C controller and an input/output interface to sensors/acuators. This electronic module is the smallest replaceable unit (SRU) in case of fault. The communication network interface (CNI) is a dual-port memory that allows simultaneous access from the host CPU and the controller to share data. The communication controller comprises the protocol engine, the storage of TTP/C control data (MEDL – Message Descriptor List) and an independent hardware unit, the bus guardian to protect the bus from timing failures of the controller.

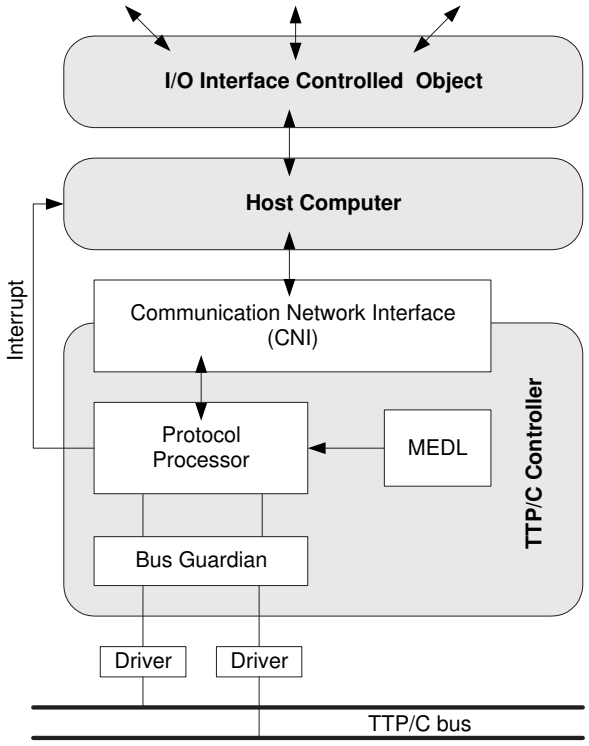


Figure 3.7: Architecture of a TTP/C node.

3.3.1 Network Topology

TTP/C networks can contain up to 64 nodes. The cabling topology can be bus, star, or any combination of the two. Multiple stars or sub-buses on stars are also supported. A redundant star topology with a bus guardian integrated into the star coupler should be adopted in safety-critical TTP/C configurations. A dual bus is used where the two buses are synchronized and always carry the same data. Each TTP controller sends and receives messages on both buses.

3.3.2 Message Transmission

Communication on the network proceeds according to a TDMA scheme, based on static tables loaded at each node. This TDMA scheme divides time into slots each being statically assigned to a particular node. During its slots the node has exclusive write permission to the interconnection network. The slots are grouped into rounds: in the course of a TDMA round every node is granted write permission in exactly one slot. Furthermore, nodes always send in slots having the same relative position within a round; finally, the slots assigned to a particular node have the same length in each round (message payload between 2 and 236 bytes of data). A cluster cycle comprises several TDMA rounds and multiplexes the slots assigned to a node in succeeding TDMA rounds between different messages produced by the node. Every node has knowledge, stored in read-only memory, of the complete communications pattern (and not only of the slots assigned to itself). These data are called message descriptor list (MEDL) and allow nodes to know *a priori* the types of messages being sent or received. Thus, there is no need for transmitting the sender IDs or message IDs explicitly. This static link is exploited in the architecture for the purposes of error detection and bandwidth efficiency.

A distributed fault-tolerant clock synchronization algorithm establishes the global time base needed for the distributed execution of the TDMA scheme.

The scheduling paradigm adopted by TTP/C is static table-based and offline. The pre-computed schedule is loaded into the Message Descriptor List (MEDL) of each controller and a subset of the MEDL is also loaded in each bus guardian. As a result, every node has complete knowledge of the complete TDMA scheme and not only the slots assigned to it.

TTP/C has two types of frames: I-frames (initialization frames) and N-frames (normal frames). I-frames are used for system initialization and convey the internal state of the TTP/C controller (C-state). This allows nodes waiting to participate in the protocol to join the network when they receive an I-frame. I-frames are sent by the communication subsystem during the startup phase of the protocol, and at predefined intervals during normal operation of the protocol to facilitate re-integration of failed nodes. N-frames (normal frames) are used during normal operation and contain application data. The header byte of a N-frame contains two fields: the first 1-bit field identifies the message type, and a three bit mode change field is used to request system-wide mode changes.

Available implementations of TTP/C rely either on a bus or star topology and protocol

processors (controllers) are available, implementing the TTP/C protocol. Currently TTP, supports speeds up to 25 Mbit/s with typical data efficiency of 60%.

TTP/C supports online mode changes, i.e., several communication schedules, or modes, can be used and switched online. For example, an aircraft can be on the ground, in take-off, or landing and every one of these mutually exclusive phases is called a mode. Every mode has its own schedule parameters in the MEDL but TTP/C requires that all modes are based on the same TDMA slot sequence. All synchronized nodes of a TTP/C cluster must operate at the same time in the same mode.

3.3.3 Bus Guardianship

In TTP/C each node has access to two communication channels and thus has two network interfaces each with its own bus guardian. Each bus guardian [Tem98] has an independent copy of the schedule and its own clock oscillator. However they do not synchronize independently and share the same power supply and the same encapsulation as their controllers. Consequently common mode failures are possible. Recently, the bus guardianship has been moved to a central hub [BKS03], a fully independent fault containment region. However, it is at the same time a single point of failure. The possible duplication of the central star hub overcomes this issue. Hence, a safety-critical TTP/C configuration requires an interconnection network with two intelligent star couplers, one for each channel.

The guardians are integrated into the star coupler and each star coupler forms an FCR with its own power supply, distributed clock synchronization system, and timing source. It has *a priori* knowledge of the sending slots that are assigned to each node and opens the respective gate only during the sending slot of a node. The star coupler also regenerates the incoming signal stream based on its own clock and power supply in order to mask Slightly-of-Specification (SoS) failures of a sending node.

3.3.4 Clock Synchronization

In TTP/C the clock synchronization among an ensemble of clocks proceeds according to the three distinct phases:

1. Every clock reads the time values of a well-defined ensemble of clocks. In order to handle a Byzantine fault in each TMDA round, the clock synchronization algorithm must operate in an ensemble of at least 4 nodes. These statically assigned nodes, the master clocks, are used to generate the global time.

2. Every node calculates a correction term for its clock using the fault-tolerant average (FTA) algorithm in order to calculate an average out of four statically selected time-keeping clocks. Low quality clocks, that are not time-keeping clocks, may use dynamic rate correction since they do not influence the characteristics of the global timebase.

3. Every node applies the correction term to its local clock to bring its values into better agreement with the ensemble.

TTP/C also supports external clock synchronization.

3.3.5 Error Detection

In TTP/C the receiver of a frame has knowledge about the physical identity of the sender of a frame. This knowledge is used to provide error-detection capabilities at the architecture level, such as a membership service and a clique avoidance service.

TTP/C nodes are assumed to be fail-silent nodes, i.e., they are either operational or silent towards the rest of the system. The host computer is responsible for silence in the value domain, while the TTP/C controller uses a bus guardian with its own clock that only allows transmissions during the pre-allocated time slots, thus providing fail-silence in the time domain.

A membership agreement service informs all nodes of a cluster about the operational state of each node within a latency of about one TDMA round. A node is operational if:

- the host computer of the node has updated its life-sign within the last TDMA round and
- the communication controller is operating and synchronized with the rest of the cluster.

If the host computer has not updated the life-sign, the host is considered non-operational and the communication controller does not send a frame. The controller remains synchronized by receiving all frames, but it is unable to transmit.

In TTP/C all nodes are forced to implicitly agree on their controller states (C-states). The controller state consists of three fields: the MEDL position, the time, and the membership. The MEDL position field is a pointer to the current entry in the MEDL. The time field contains the global time at the beginning of the current FTU slot. The membership field indicates which FTUs have been active and which FTUs have been inactive at their last membership point. To enforce C-state agreement between a sender and a receiver the CRC of a normal message is calculated over the message contents concatenated with the local C-state. A receiver can only interpret the frame if sender and receiver agree about the controller state at the time of sending and receiving. In case the C-state of the sender differs from the C-state of the receiver, the message will be discarded by the receiver due to the different CRC.

Re-integration frames are used to re-integrate silent nodes. The result of the checksum calculation used in TTP/C is such that all active nodes have a consistent C-state. However, it also means that a node that has a deviating C-state can no longer receive any messages. The reason for the deviating C-state could be a transient error that corrupted a message only

at that node. To allow reintegration, the current C-state is periodically broadcast explicitly in special I-frames; the current C-state is not required to decode this message.

TTP/C supports atomic broadcast in the sense that a TTP/C controller terminates operation when it cannot receive a message that has arrived at the majority of the nodes. This is because the checksum calculation method prevents a node from receiving messages when it has a deviating membership view; when more than half of the message receptions have failed, the node becomes silent.

3.3.6 Operational Flexibility

Although TTP/C can send only time-triggered messages, an application can send both time-triggered and event-triggered messages. The transmission of event-triggered messages is performed over an event channel (bandwidth is reserved for event transmissions inside the TDMA slots, and the messages use identifiers). Obermaisser [Obe02] proposed a scheme to CAN emulation on TTP/C, in which a CAN-compatible interface is provided and the CAN messages are transmitted inside TTP/C data frames. The use of event channels does not interfere with TTP/C composability since bandwidth is not arbitrated among different nodes (as in CAN), but only among different functions within a node. In this way, timing and bandwidth analysis for event transmissions is done on a per-node basis and does not need system-level design. This scheme is somewhat inefficient concerning resource utilization, because it relies on bandwidth reservation made at design time that cannot be adapted on-line. Thus this is still a pre-run time type of flexibility.

At run-time TTP/C can switch between a range of modes [KNH+98] and every host computer may request a mode change that must be accepted by all nodes before becoming active. Although a large number of modes can be defined in a single cluster design (limited by the MEDL memory), they all must be defined at pre-run time. To avoid mode changes at the wrong point in time and to prevent inconsistencies between the states of the nodes, the points in time when mode changes are allowed are determined before runtime, mode changes are executed only when this is required by the state of the controlled object and all nodes of a cluster change mode at the same point in time, i.e., all nodes of a cluster must agree on the current mode at all points in time.

3.4 FlexRay

Some key players of the automotive industry, BMW and DaimlerChrysler, realized that the requirements for future automotive control applications include the combination of higher data rates, deterministic behavior, support of fault tolerance and flexibility in both bandwidth and system extension, could not be met using existing communication protocols. This technical constatation, together with some commercial considerations, related with royalties and exclusivity [Fg02], were the main reasons that made BMW and DaimlerChrysler to join forces,

in September 2000, with Philips and Motorola to form the FlexRay Consortium. Since then the Consortium has grown to include most of the automotive industry's largest manufacturers, including Volkswagen, Bosch, Motorola, Toyota, General Motors, Ford and many others.

FlexRay protocol [NFG+02][Bel02][Fg02][Con04b] aims at combining safety and flexibility by supporting two communication paradigms, deterministic communication (time-triggered and statically defined) and dynamic event-driven communication.

3.4.1 Network Topology

Concerning the network topology, FlexRay supports both active star (with at most 3 cascaded stars between two arbitrary nodes) with possible passive bus extensions. Early results [Fg02] show that the passive bus (at full speed - 10 Mbit/sec) is limited to stubs of around 25 cm and to a maximum of 6 to 8 stubs per network.

The star coupler has autonomous power *moding* i.e., the star decides to go to the sleep mode after network idle for a long time. The star coupler also implements error detection (clamped wire and permanent noise) to exclude affected branches from communication and to auto-recover once the failure has been removed.

3.4.2 Message Transmission

The communication cycle is the fundamental element of the media access scheme within FlexRay and it is defined by means of a timing hierarchy consisting of four levels [Con04b]:

- **Static segment** – a static TDMA scheme is used to arbitrate transmissions.
- **Dynamic segment** – a dynamic mini-slotting based scheme is used to arbitrate transmissions.
- **Symbol window** – a communication period in which a symbol can be transmitted on the network. Symbols are FlexRay commands, e.g., to wake up an active star.
- **Network idle time** – a communication-free period that concludes each communication cycle.

FlexRay distinguishes in its time-triggered part between slots, cycles, frames and messages. A slot is identified by a slot-ID, a cycle by a cycle-ID, a frame by a frame-ID, and a message by a message-ID. The cycle-ID, the frame-ID, and the optional message-ID are all conveyed in every time-triggered FlexRay frame.

The static part of the protocol follows a statically and pre-defined TDMA strategy with support for both single channel and dual channel operation (Figure3.8). The TDMA scheme is established among the controllers using the fault tolerant midpoint clock synchronization algorithm, for offset and rate correction.

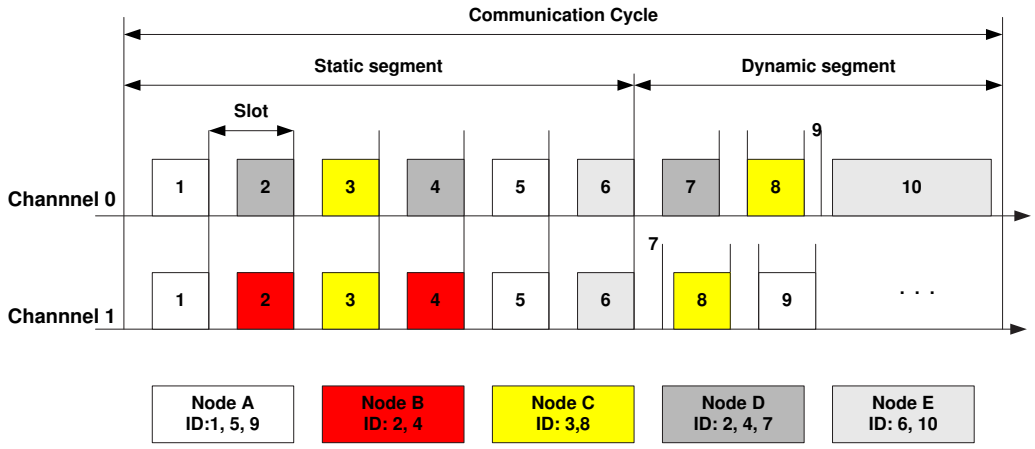


Figure 3.8: Definition of a communication cycle with static segment (adapted from [Bel02]).

The dynamic part adopts the mini-slotting mechanism of the Byteflight protocol [M. 00]. The partition between the two segments is made at design time and loaded into the controllers and bus guardians. The FlexRay protocol can operate in three different modes:

- Static with distributed clock synchronization.
- Mixed static/dynamic with distributed clock synchronization
- Dynamic with single master synchronization, via a start of cycle symbol.

In the static part, nodes that are connected to both channels send their frames simultaneously on both channels. Conversely to the static part, in the dynamic part the traffic on both channels can be different (see Figure 3.8). If two nodes are connected to a different channel each, they may share slots in the static part. Thus it is also possible to have different traffic in the static part of both channels.

A FlexRay frame can hold a payload up to 244 bytes.

3.4.3 Bus Guardianship

The bus guardian has *a priori* knowledge of the transmission times of the node and restricts transmission attempts of the communication controller to the configured time slots. If the bus guardian detects any mismatch between the schedules of the communication controller and the bus guardian, it signals an error condition to the host and inhibits any further transmission attempts. This knowledge (schedule) is downloaded to the guardian during a configuration phase. The bus guardian schedule includes active slots in the static segment, the dynamic segment and the symbol window. The bus guardian can be configured to enable or to disable transmit access during the dynamic segment.

FlexRay bus guardians do not perform an independent clock synchronization and have no independent power supply, thus the node and the guardian form a single fault confinement region. According to [Con04a], the bus guardian functionality can also be located at a FlexRay's active star.

3.4.4 Clock Synchronization

In FlexRay the frames that are used for clock synchronization are selected on the basis of the contents of a message (Sync Bit set). In FlexRay the synchronization ensemble is formed by eliminating the j largest and j smallest clock values of the selected clocks. FlexRay uses the fault-tolerant midpoint (FTM) algorithm in order to calculate the correction value out of the synchronization ensemble. The FTM algorithm calculates its correction value by using only the two clocks at the extreme ends of the synchronization ensemble. FlexRay also performs a dynamic rate correction to compensate the drift-rate of the clocks.

3.4.5 Error Detection

The FlexRay error management policy is very similar to the one adopted by TTCAN, using the dedicated degradation concept that allows three states: normal, passive and halt. Associated with each channel is a status vector (Channel Status Error Vector - CSEV) mapped into the diagnosis interface of the communications controller, which can be configured to be an interrupt source (bit-by-bit enable vector).

The full schedule for the time-triggered portion is not installed in each controller. So, each controller only learns the full schedule when the bus starts up. Bus guardians are informed of which slots are allocated to their transmissions at initialization time.

3.4.6 Operational Flexibility

Since the full schedule for the time-triggered portion is not installed in each controller, the receiver of a frame has no knowledge about the physical identity of the sender of a frame. This increases flexibility, but it reduces the error-detection capabilities at the architecture level, since the receiver can never identify a faulty sending node on the basis of a faulty or missing frame.

It is not possible to commute on-line between different modes, to adapt to evolving requirements or to operational conditions (e.g., take-off and landing in avionics). Changing the communication schedule requires stopping the network, new configurations must be downloaded and the network needs to be re-started. The time required for such actions can inhibit the use of mode changes in time sensitive applications.

3.5 ARINC-629

ARINC-629 [ARI90] is a bus that was specifically designed to provide general purpose data communications between avionic subsystems [AG97], the so-called *terminals* (Remote Data Concentrators and cabinets). ARINC-629 is an evolution of ARINC-429 Digital Data Transfer Systems specified in 1977 by the ARINC committee. The original 429 standard, defining a unidirectional broadcast data bus (single transmitter, multiple receivers), has been used extensively in the civil aircraft field (e.g. in Boeing 767). Research and development work carried out by Boeing in the form of the Digital Autonomous Terminal Access Communication (DATAC) program has, however, resulted in a new standard being agreed for use on modern civil aircrafts demanding more data processing and higher speed data transfers. This was applied to the Boeing 777 and became the ARINC-629 standard in 1989.

3.5.1 Network Topology

The maximum bus length allowed for ARINC-629 is 100m and the maximum possible number of connected Line Replaceable Units[2] (LRU) is 120 at a data rate of 2 Mbps. The maximum stub length connecting each LRU is 40m.

The application is responsible for the coordination of multiple redundant buses.

3.5.2 Message Transmission

The ARINC-629 standard supports two alternative data link level protocols: the **basic protocol** and the **combined protocol**, which cannot coexist on the same bus. Despite this fact, there are many similarities between the two.

The bus access is coordinated at two levels:

- System level decomposition of bus time into cycles, both minor and major. Cycle length may be fixed or variable depending on protocol configuration.
- Medium access contention resolution across multiple terminals within each minor cycle by a combination of carrier sensing and observation of pre-assigned waiting times and common bus idle periods, implementing a CSMA-CA (mini-slotting) media access scheme.

The basic and combined protocols support both periodic and sporadic transmissions. Periodic messages are transmitted according to their predefined parameters. The way in which event-triggered messages are handled is the main difference between the two protocols.

The combined protocol is more suited for mixing periodic and non-periodic traffic. The overall bus scheduling scheme is to first transmit the periodic messages and then to allow sporadic messages to compete for the spare bandwidth at the end of the minor cycle. A timing

[2] Aeronautical terminology, it is equivalent to an Electronic Control Unit (ECU).

analysis for ARINC-629 [AG97] shows that the protocol is capable of supporting periodic and sporadic traffic with deadlines. This is accomplished without any global time source, i.e. each terminal has its own local time source which may drift relatively to others. Multiple timers and redundant circuitry are employed within each terminal to prevent single hardware faults causing multiple nodes to transmit simultaneously.

A message has variable length and can accommodate up to 31 wordstrings. Each word also has variable length and contains a 20-bit word label and up to 256 20-bit data words. An ARINC-629 node consists of a bus controller and the main computational unit both connected via a shared memory. A 2 Mbps serial data transmission rate is specified for twisted pair conductors.

Basic Protocol

The Basic Protocol (BP) has two sub-modes: the periodic mode and the aperiodic mode. In the periodic mode, all transmission times are fixed within each minor cycle. In the aperiodic mode, each individual terminal transmission times may vary between cycles. Notice that in both modes, access is granted to each terminal in a fixed order and without preemption within each minor cycle. The protocol defines three time intervals:

- **Transmit Interval** (TI) – represents a common time interval for which each terminal must wait between its own successive transmissions. A TI timer starts every time the terminal initiates a transmission.
- **Synchronization Gap** (SG) – is a common idle time, longer than any individual terminal gap on the bus and is intended to cope with variations in transmission times of individual terminals between successive minor cycles. The SG timer starts every time the bus is sensed idle. The timer may be reset either before it has elapsed, if any bus activity is detected, or after it has elapsed, the next time the local terminal begins transmitting.
- **Terminal Gap** (TG) – represents a unique time for which a terminal must wait without any bus activity before starting its own transmission. The TG timer starts every time the bus is sensed idle. But unlike SG, the TG timer is reset only after it has elapsed and upon detection of any bus activity.

ARINC 629 uses a three time-out parameters waiting room protocol [Lam74]. The synchronization gap (SG) controls the entrance to the waiting room, the terminal gap (TG) controls access to the bus and the transmit interval (TI) prevents a host from monopolizing the channel. This protocol is depicted in Figure 3.9:

1. Two processes P1 and P2 that want to transmit a message are admitted to the distributed waiting room.

2. Both processes initially must wait a time greater than SG to enter the waiting room.

3. In the waiting room, both processes wait for another period greater than their individual TG.

4. All TGs are different, so processes with shorter TG (i.e. P1 with TG1=2) starts to transmit. In the start of transmission, P1 set its TI to block temporally any further transmissions from that node.

5. As soon as P1 has started transmitting, P2 backs off until P1 has finished.

6. P2 then waits for its TG (i.e. TG2=3) and starts transmitting its message.

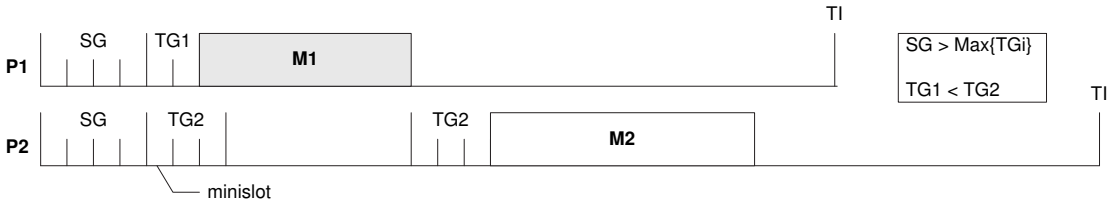


Figure 3.9: The waiting room protocol adopted in ARINC-629.

Combined Protocol

This protocol, initially proposed and developed by British Aerospace and Smiths Industries, aimed to overcome the deficiencies of the basic protocol for systems that require a more effective approach to combined handling of periodic and sporadic data transmissions.

In the Combined Protocol, periodic transmissions, the so-called level-1, are handled in a fixed order and without preemption, as in the Basic Protocol, but bus cycle times are fixed in duration. Any sporadic message request that arrives during the current minor cycle may only be serviced within the cycle time available after all periodic messages have completed. These are infrequent high priority messages with short duration and are called level-2 messages. If time allows, other sporadic messages are serviced in the remaining cycle time and can be longer, less frequent and low-priority (level-3). Level-2 and level-3 messages are limited to one wordstring in length.

As in the Basic Protocol, each terminal is pre-assigned an unique terminal gap (TG). The transmit interval (TI) is applicable only to the first (periodic) terminal transmission in each minor cycle. For all other terminals, this is overridden by a *concatenation event* which forces all un-elapsed TI timers to be canceled. This has the effect of compressing periodic messages into a burst of activity (separated only by TG delays) at the start of each cycle.

Two types of synchronization gaps are defined: the periodic synchronization gap (PSG), which is used to achieve synchronization at the minor cycle level and an aperiodic synchronization gap (ASG), that is employed within each minor cycle to synchronize the transitions between level-1 and level-2 transmissions and, in turn, level-2 and level-3 transmissions. The ASG timer is started upon detection of bus idle after the initial burst of periodic transmissions and it is reset upon detection of any bus activity. This is similar to the aperiodic mode of the Basic Protocol and so, sporadic messages only consume resources when required, i.e. all terminals are offered access to the bus in level-2 and level-3, but this is only used by terminals that have sporadic messages ready to transmit, which gives level-3 sporadic messages a better chance (on average) of being serviced.

Within each minor cycle, each terminal is restricted to one level-2 transmission but may perform multiple level-3 transmissions if required and if there is time available. In order to enforce fixed duration minor cycles, an aperiodic access time-out (AT) is used to indicate the time to next periodic (level-1) transmission, so that any sporadic transmission with the potential to take longer than this time is prevented from starting in the current cycle. Level-3 sporadic messages are permitted to span multiple minor cycles, i.e., they can be put on hold during several cycles until they can be transmitted. Backlog level-3 messages always take priority over those generated in the current cycle. Level-2 sporadic messages must be transmitted within the current minor cycle, otherwise they are lost (i.e., they are not queued).

3.5.3 Bus Guardianship

ARINC-629 does not consider the use of bus guardians to protect the nodes against*babbling idiot* failure mode. This is a direct result of the reasoning behind ARINC-629 design: to leave all fault tolerance support to the application layer. ARINC-629 only provides a single parity bit. This reasoning is based on the assumption that there will be multiple redundant buses and the application should be able to tolerate some network errors that may result in message faults (omissions).

It could be argued that moving the fault-tolerance mechanisms higher in the OSI stack, as in ARINC-629 or in FlexCAN [PF04], increases both the application design complexity and potentiates design defects, because every new fault-tolerance mechanism needs to be designed from scratch for every different application. This is an important factor to take into account and whose relevance keeps increasing with the application size: the more complex it is, the higher is the number of potential software defects and more difficult becomes to identify them. However, this scenario is not so critical in the specific case of aviation industry where time to market is not so crucial as in other industries and so projects can be made according to best practices. In other industries, the pressure imposed by the time to market factor may introduce many defects in the software if an application oriented fault-tolerance approach is followed. This is why some authors claim [Kop97][PMJ00] that the fault-tolerance mechanisms

should be implemented as low as possible within the OSI stack to partially hide them to the designers. Those mechanisms are carefully designed and properly validated once and may be used thereafter with some guarantees. This is clearly the approach followed by TTP/C and in a smaller scale by FlexRay (it does not provide a membership service). The drawback of providing fault-tolerance mechanisms at the lower levels of the OSI stack is the overhead introduced by these mechanisms in applications that do not require them.

3.5.4 Clock Synchronization

ARINC-629 adopts a time-controlled medium access strategy where time is partitioned into mini-slots each longer than the propagation delay of the channel, without requiring a global synchronized timebase in every terminal. Every terminal is assigned a unique number of mini-slots that must elapse, with silence on the channel, before it is allowed to transmit.

The mini-slot counters are synchronized by the reception of a message and the longest period of bus silence is relatively short. Thus relatively high clock drifts can be tolerated.

3.5.5 Error Detection

ARINC-629 standard only provides a single parity bit, all fault tolerance support is application layer's responsibility. However some ARINC-629 controllers provide some extra functionalities as, for example: Manchester encoding error, short string error and bus quiet error.

3.6 Brief Comparison and Conclusion

Despite being designed for diverse application scenarios, the protocols previously presented can be compared according to their fundamental properties. In the scope of this thesis the key property to evaluate is *flexibility*.

When addressing operational flexibility, event-triggered communication systems such as native CAN are typically well positioned because they react promptly to communication requests that can be issued at any instant in time letting the bus available in the absence of events to communicate. In other words, they react to instantaneous (variable) communication requirements. On the other hand, time-triggered systems, such as TTP/C and TTCAN, are not so flexible because communication takes places at pre-defined instants, only, not taking into account run-time variations in the application communication requirements. This can be improved, as in TTP/C, allowing mode changes between a set of pre-defined (static) modes. Moreover, any time-triggered protocol allows reserving space at design-time for nodes and message streams that can then be added on-line but this is an inefficient technique because extra bandwidth is kept allocated even when it is not needed for long periods.

Protocol	Network topology	MAC protocol	Online scheduling[3]	Built-in fault-tolerance	Flexibility of TT traffic
CAN	bus	CSMA-BA	n.a./yes	+++	n.a.
TTCAN	bus	TDMA/CSMA-BA	no/yes	++	no
FTT-CAN	bus	M-MS[2]/CSMA-BA	yes/yes	++	++
TTP/C	bus/star	TDMA	no/no	+++++	+
FlexRay	bus/star	TDMA/FTDMA	no/yes	++++	no
ARINC-629	bus	FTDMA	yes[1]/yes	+	no

[1]Possible, although not common; [2]Master-Multislave; [3]Time-triggered traffic/Event-Triggered traffic

Table 3.1: Summary of communication protocols' properties.

The systems that combine both event and time-triggered paradigms present an intermediate level of operational flexibility due to the event-triggered part, despite the low flexibility of the time-triggered one. This is the case of TTCAN, FlexRay and ARINC-629. This is even the reason why FlexRay claims to be flexible. Moreover, TTCAN, FlexRay and TTP/C, block on-line changes to the traffic schedule matrix or table. These can only be performed in configuration mode, which implies halting the system.

The FTT-CAN protocol deserves a particular reference in what concerns operational flexibility since it was built explicitly with the purpose of improving this aspect. Therefore, it not only supports event and time-triggered traffic with temporal isolation but it also delivers flexible time-triggered communication services, allowing on-line changes to the periodic communication requirements with timeliness guarantees.

Concerning dependability, TTP/C and FlexRay are, probably, the most robust solutions with an advantage towards TTP/C because of its membership service. The star topology that these protocols support also helps improving error detection and isolation capabilities. On the other hand, the philosophy of ARINC-629 is to leave all dependability issues to the application level, removing that concern from the communication protocol. TTCAN does not seem to meet all requirements for safety critical distributed systems (e.g. redundant communication, fail-silent nodes). In fact, some authors [BBRN04][RP03][FOFF04] claim that native CAN is more dependable than TTCAN. The original proposal for FTT-CAN did not consider dependability aspects and thus presents a level similar to that of TTCAN in this aspect.

Table 3.1 summarizes some of the properties of the communication protocols as discussed above. Among these protocols, FTT-CAN is in a favored position to combine a high level of operational flexibility with a high level of dependability as long as appropriate mechanisms are added to it. This is the main motivation for this work.

Chapter 4

Impairments to dependability of CAN and FTT-CAN

4.1 Introduction

FTT-CAN dependability can be compromised by faults in the channel or faults in the nodes. A fault in the channel is one fault that affects any element of the physical layer of the network (cable, connectors, transceiver's circuitry, etc.). A fault in a node is one fault that affects any component of the node (microcontroller, memory, sensors, etc.). Both types of faults can have either external origin, e.g. due to *electromagnetic interference (EMI)*, or internal origin, e.g. due to a defective component, connector or a cold soldering.

The CAN protocol defines its own mechanisms to detect and signal channel errors. Theoretically, these mechanisms enforce any frame which suffers an error to be consistently rejected by every node of the network. However, as it will be detailed later, this is not true in some specific scenarios. Concerning physical faults of the nodes, the CAN protocol does not define any mechanism to deal with these faults. Therefore, CAN nodes may theoretically fail in arbitrary ways. It is responsibility of the fault tolerance mechanisms to restrict the failure semantics of nodes and channels. In this context, it is important to assess the impact of faults in CAN and in FTT-CAN to design fault tolerant mechanisms able to efficiently circumvent those faults.

4.2 Consequences of faults in the channel

The CAN protocol defines its own mechanisms in order to tolerate transient channel errors, by means of error detection, error signaling and frame retransmission.

For the purpose of this work, only transient faults that change the value of, at least, one bit that is either transmitted to, or received from the channel, are considered.

In principle, and according to the error detection and signaling capabilities of CAN, any

frame which suffers an error would be consistently rejected by all the nodes of the network. However, some failure scenarios have been identified [RVA+98][PMJ00] that can lead to undesirable symptoms such as inconsistent omission failures and duplicate message reception. In this context, we will start this section by discussing CAN fault confinement mechanisms to better understand the failure scenarios that lead to inconsistent failures.

In CAN each node that detects an error sends an error flag composed of six consecutive dominant bits enabling all nodes on the bus to be aware of a transmission error. The frame affected by the error automatically re-enters into the next arbitration phase. The error recovery time (the time from detecting an error until the possible start of a new frame) varies from 17 to 31 bit times [NYQS00].

To prevent an erroneous node from disrupting the functioning of the whole system, e.g. by repetitively sending error frames, the CAN protocol includes fault confinement mechanisms that are able to detect permanent hardware malfunctioning and to remove defective nodes from the network. To do this a CAN controller has two error counters; the transmit error (TEC) and the receive error (REC) counters which are incremented/decremented according to a set of rules [BOS91][ISO93]. Each time a frame is correctly received or transmitted by a node, the value of the corresponding counter is decreased. Conversely each time a transmission error is detected the value of the corresponding counter is increased.

Depending on the value of both counters, the station will be in one of the three states defined by the protocol: error active, error passive and bus-off. In the error active state (REC<128 and TEC<128) the node can send and receive frames without restrictions. In the error passive state ((REC>127 or TEC>127) and TEC<=255) the node can transmit but it must wait 8 supplementary bits after the end of the last transmitted frame and it is not allowed to send active error frames upon the detection of a transmission error, it will send passive error frames instead. Furthermore, an error-passive node can only signal errors while transmitting.

After behaving well again for a certain time, a node is allowed to re-assume the error-active status. When the TEC is greater than 255 the node CAN controller goes to the bus-off state. In this state, the node can neither send nor receive frames and can only leave this state after a hardware or software reset and after having successfully monitored 128 occurrences of 11 consecutive recessive bits (a sequence of 11 consecutive recessive bits corresponding to the ACK, EOF and the intermission field of a correct data frame).

When in the error passive state, the node signals the errors in a way that cannot force the transmitter to retransmit the incorrectly received frame. This behavior is a possible source of inconsistency that must be controlled, as it will be detailed in the next section. As an example of the consequences this can have, consider the case of an error-passive node being the only one to detect an error in a received frame. The transmitter will not be forced to retransmit and the error-passive node will be the only one not to receive the message. Several authors proposed avoiding the error passive [RVA+98][HKD97][FNP+98] state to eliminate this problem. This

is easily achieved [RVA+98][PMJ00] using a signal available in most CAN circuits, the error warning notification signal. This signal is generated when any error counter reaches the value 96. This is a good point to switch off the node before it goes into the error-passive state, assuring that every node is either helping to achieve data consistency or disconnected.

In the next section, other inconsistency problems reported in the literature that are more difficult to solve will be described. All these problems appear even if no node is in the error-passive state.

4.2.1 CAN Inconsistency Scenarios

Inconsistency scenarios in CAN are a direct consequence of the instant in which the transmitter and the receiver validate a message.

According to CAN specification version 2.0, part A page 21, point 4 - Message validation [BOS91]:

> "*The point of time at which a message is taken to be valid is different for the transmitter and the receiver of the message.*
> ***Transmitter:*** *The message is valid for the transmitter, if there is no error until the end of END OF FRAME. If a message is corrupted, retransmission will follow automatically and according to prioritization. In order to be able to compete for bus access with other messages, retransmission has to start as soon as the bus is idle.*
> ***Receivers:***
> *The message is valid for the receivers, if there is no error until the last but one bit of END OF FRAME.*"

Due to this frame validation rule, the behavior of the CAN controllers, in case of error in the last bit of the EOF, is special and it can generate very specific error scenarios [RVA+98][PMJ00]. Whenever a transmitter detects an error in the last bit of the EOF, it handles it in the usual way, *i.e.* it starts an error flag in the next bit, it invalidates the current frame transmission, and it retransmits the message (new higher priority messages will be transmitted first). Conversely, when a receiver detects an error in the last bit of the EOF, it correctly accepts the frame and it generates an overload flag. This behavior is depicted in Figure 4.1a.

A set X of receiving nodes detects an incorrect dominant value in the last bit of the EOF, while the transmitter and the set Y of receiving nodes, see a correct recessive bit. The nodes of X start the transmission of an overload flag immediately after the error. The remaining nodes detect the first dominant bit of the overload flag in the first bit of the inter-frame space and they also start the transmission of overload flags. Therefore, both the transmitter and the nodes belonging to Y consider the frame as being correctly transmitted. The nodes belonging to X also accept the frame and consistency is achieved, despite the last-bit mismatch.

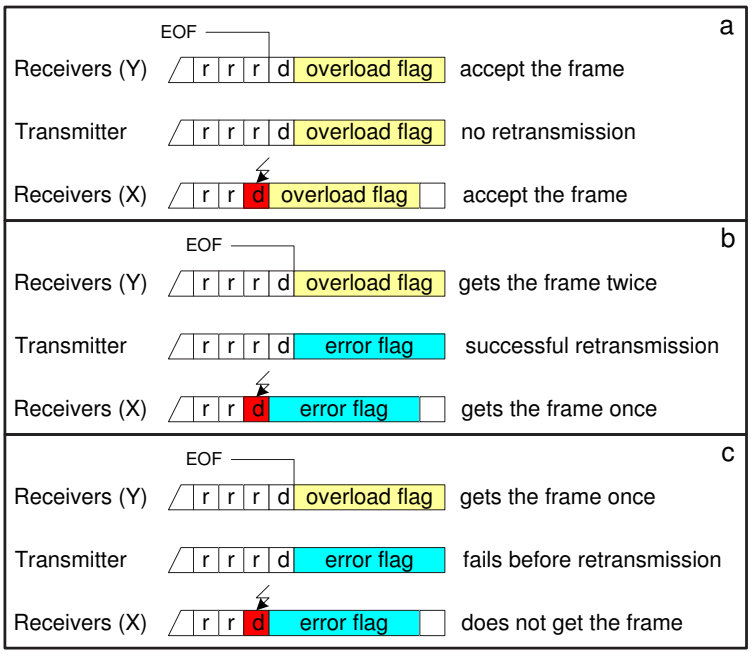


Figure 4.1: Some possible error scenarios in CAN (adapted from [RP03]).

The scenario depicted in Figure 4.1b causes nodes to receive the same frame twice. In this case an error corrupts the last but one bit of the EOF of the nodes belonging to the set X, so in the next bit these receivers start the transmission of an error frame. The first dominant bit of this error flag is seen by the transmitter and by the nodes belonging to Y as an error in the last bit of their EOF, and so, the nodes belonging to X reject the frame. Notice that the nodes belonging to Y accept the frame because of the frame validation rule. The transmitter, in turn, retransmits the frame causing the nodes belonging to the Y set to eventually receive the frame twice. This possible error scenario in which an error in the last but one bit may cause an inconsistent message duplicate (IMD) (Figure 4.1b), *i.e.* some nodes receive duplicates of a frame while others only receive a single frame, was firstly identified by Rufino [RVA+98].

In Figure 4.1c another possible error scenario, also firstly reported by [RVA+98], is depicted. This case is similar to the one illustrated in Figure 4.1b, but now the transmitter fails by crashing after the first transmission of the frame and consequently, it does not retransmit the frame. Therefore, the nodes belonging to Y receive the frame whereas those of X do not. This causes inconsistent message omissions (IMO). Notice that the transmitter does not need to fail by crashing after an error in the last but one bit of the EOF to cause an inconsistent message omission. If a time-triggered decision at the receivers side has to be made while the sender is retransmitting the message or the transmitter is not allowed to retransmit the frame (as in TTCAN), then the inconsistency will not be removed in some receivers.

The error mechanisms of CAN also ensure that, under certain fault assumptions, the

transmitter of a frame is always able to detect whether this frame has been rejected by any node. Only in some specific scenarios, which are described in [PMJ00], the transmitter would not able to detect that the transmission has been inconsistent. However, we assume that the probability of these scenarios is very low and, as a result they will not be considered further.

The probability of those error scenarios depends on an important factor, the CAN bit error rate. The analysis presented in [RVA+98] is based on the assumption that the *bit error rate* varies from 10^{-4}, in case of an aggressive environment, to 10^{-6} in the case of a benign environment. The results obtained, based in these assumptions, for IMO/h and IMD/h are rather high and are a serious impairment of using CAN (or CAN based protocols) in safety-critical applications.

4.2.2 FTT-CAN inconsistency scenarios

In FTT-CAN networks, additionally to the previous scenarios, there are some new scenarios that may cause inconsistent message omissions.

- Inconsistent message omissions of the synchronous (time-triggered) messages (Figure 4.2). An inconsistent synchronous message omission occurs whenever it is inconsistently received by some nodes and the sender is not able to retransmit it. This is a direct consequence of truncating the synchronous traffic at the end of the respective window.

- Inconsistent message omissions of the Trigger Message (Figure 4.3). An inconsistent message omission of a TM occurs whenever the TM is inconsistently received and the master is not able to retransmit the frame during the trigger message transmission window (e.g. due to a burst of errors in the channel).

Concerning the impact of message omission and/or duplicates in FTT-CAN, three different perspectives can be considered, according to the criticality and nature of the exchanged messages: Trigger messages, synchronous messages and asynchronous messages. Solutions to circumvent these issues will be presented in the next Chapter.

4.3 Consequences of physical faults of the nodes

In principle, the CAN protocol does not restrict the *failure semantics* of the nodes, so that they may fail in arbitrary ways. Some of these failures are automatically handled by CAN's native implementation of error detection capabilities and automatic fail-silence enforcement, described previously, leading the erroneous node to a state, the bus-off state, where it is unable to interfere with other nodes. However, in some specific situations, these mechanisms do not fully contain the errors within the nodes. Specifically, a CAN node only reaches the bus-off state (fail silence) after a relatively long period of time (when the TEC reaches 255). For example, in the case of an erratic transmitter in a 32 node CAN network at 1 Mbps, the

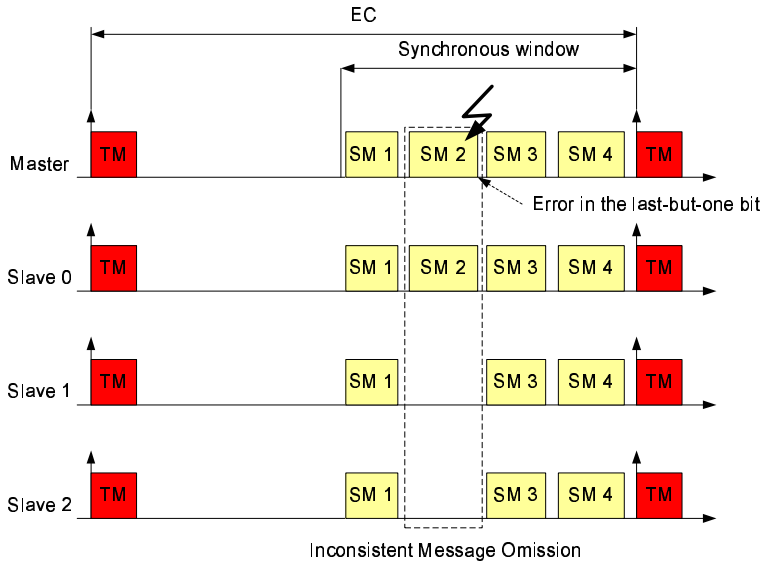


Figure 4.2: synchronous message inconsistent message omission scenario. Slave nodes 1 and 2 do not receive synchronous message 2 that is correctly received by slave 0.

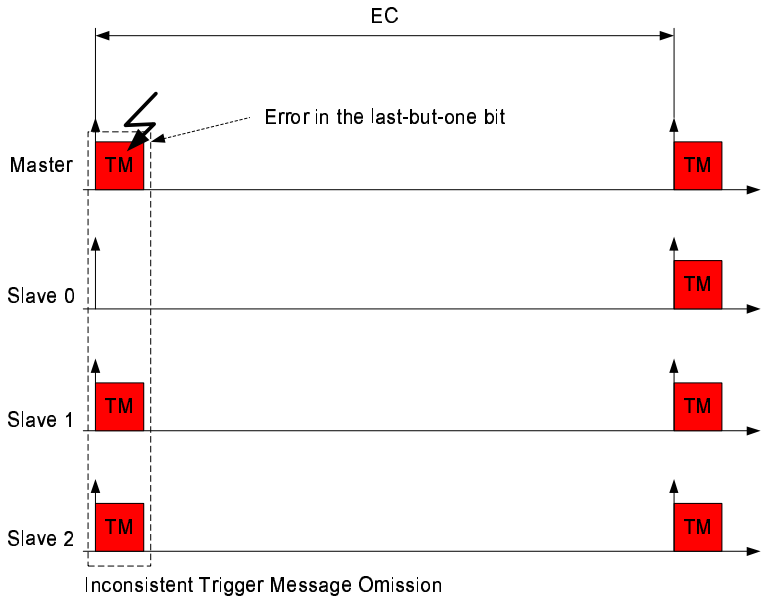


Figure 4.3: Trigger message inconsistent message omission scenario. Slave nodes 1 and 2 correctly receive the trigger message while slave 0 does not.

worst-case time to bus off is 2.48 ms [RV97]. Moreover, a CAN node running an erroneous application can also compromise most of the legitimate traffic scheduled according to a higher layer protocol implemented in software in a standard CAN controller, simply by accessing the network at arbitrary points in time (*babbling idiot* failure mode). Notice that a faulty application running in a node with a CAN controller may transmit high priority messages at any time without causing any network errors, and consequently the CAN controller will never reach the bus-off state. An uncontrolled application transmitting at arbitrary points in time via a non-faulty CAN controller is a much severe situation than a faulty CAN controller also transmitting at arbitrary points in time because, in the first case, a non faulty CAN controller has no means to detect an erroneous application transmitting legitimate traffic. In the second case the CAN controller would enter bus-off state after a while and the error would be eventually confined.

The initial FTT-CAN protocol proposal [AFF98] does not restrict the failure semantics of either master node or slave nodes. Thus the consequences of physical faults in FTT-CAN nodes are the same as in CAN plus the ones resulting from the single point of failure nature of the master node. This calls for the development of some specific circuits to restrict the ways an FTT-CAN node may fail and enforce a fail-silent failure mode, as it will be further detailed in Chapter 7.

4.4 Inconsistent Message Delivery and Bit Error Rate

CAN is particularly suited to be used in real-time systems because the response time of any message can be upper bounded [TBW95], depending both on the bus load and on channel errors. Bus load can be easily computed, however, as it was previously said, channel errors are a random variable that depends on several factors: the environment where the system operates (dynamic in most cases), the bus transmission rate, the wiring harness, the electromagnetic shielding of the devices, etc. The resulting uncertainty in the calculation of the worst case response time of a CAN message is one of the factors that limits the adoption of CAN in some safety-critical applications.

Over the years several studies have been conducted [NSS00][HNP00][BBRN02] to assess the worst case response time of CAN messages under channel errors. These studies used generic error models, that take into account the nature of the errors, either single bit errors or burst of errors, and their minimum inter-arrival time. However, no error statistics were provided to support the error models. The existence of error statistics in CAN would contribute to more realistic fault models. Worst case and average case scenarios are still open issues, and so the degree of pessimism adopted in the design of CAN based safety-critical real-time distributed systems tends to be very high, resulting in high resource demanding solutions.

In TTCAN the automatic message retransmission of CAN is disabled, while in FTT-CAN it is restricted. In these protocols, an error in a given message does not affect the response

time of others, and the error detection and signalling of CAN would normally ensure that the error would be consistently detected by all network nodes except in the error scenarios described in the previous section, where inconsistent message delivery may occur.

Inconsistent message reception scenarios are much more frequent in the case of TTCAN [RP03] than in native CAN and their frequency depends on the ***bit error rate*** of the CAN bus. All the analysis in [RVA+98] is based on the assumption that the *bit error rate* varies from 10^{-4}, in case of an aggressive environment, to 10^{-6} in the case of a benign environment. These assumptions, although realistic in other networks, seem somewhat pessimistic considering the specific case of CAN and specially the characteristics of the CAN physical layer.

4.4.1 Probability of inconsistencies in CAN, TTCAN and FTT-CAN

In [RVA+98], the probability of inconsistency scenarios in CAN is calculated as a function of the channel bit error rate ($\boldsymbol{B}$). In [RP03] the analysis is adapted to include the cases with no retransmissions upon error (TTCAN).

According to Rufino [RVA+98] the probability of having an error in the last but one bit of the EOF is given by P_{IFO} (inconsistent frame omission (IFO)), assuming that the number of bits of a frame is τ_{data} and that the probability of having an error in one particular bit follows a geometric distribution as expressed in Equation 4.1. The probability of a node crash failure, in turn, obeys to a Poisson distribution with a failure rate λ, as shown in Equation 4.2, considering a Δt period corresponding to the interval between the end of a transmission and the end of the last retransmission. If the sender crashes within Δt after the first error, with probability $(1 - e^{-\lambda \times \Delta t})$, then an inconsistent message omission (IMO) occurs (Equation 4.3). If the sender eventually retransmits the message within Δt period, an inconsistent message duplicate (IMD) will be delivered (Equation 4.4).

It is assumed that the probability for the same bit error being simultaneously perceived by all nodes is much lower than having it perceived by a subset of the nodes only. This implies that the probability of inconsistent frame omissions P_{IFO} only accounts for the temporal distribution of the errors occurring in the last but one bit of a frame of length τ_{data} and that every error will be perceived only by a subset of the nodes.

$$P_{IFO}^{CAN} = (1 - B)^{\tau_{data} - 2} \times B \tag{4.1}$$

$$P_{Fail}^{CAN} = 1 - e^{-\lambda \times \Delta t} \tag{4.2}$$

$$P_{IMO}^{CAN} = P_{IFO} \times P_{Fail} \tag{4.3}$$

$$P_{IMD}^{CAN} = P_{IFO} \times (1 - P_{Fail}) \tag{4.4}$$

Considering the specific case of Time-Triggered CAN (TTCAN), where the automatic retransmission of messages upon error or arbitration loss is disabled (single shot transmission mode), the effect of not retransmitting a frame is identical to a crash failure in the sender.

Thus, Equations 4.3 and 4.4, in the case of TTCAN, are transformed into[RP03]:

$$P_{IMO}^{TTCAN} = P_{IFO} \tag{4.5}$$

$$P_{IMD}^{TTCAN} = 0 \tag{4.6}$$

This result means that in TTCAN networks, even if one considers a benign environment [RVA+98] with a *bit error rate* of 10^{-6}, the probability of IMO is in the order of magnitude of a few tens per hour, which is a rather high value that could be experimentally measured and validated.

The FTT-CAN has some similarities with TTCAN, since during the synchronous windows message retransmission is controlled and could even be disabled in case of transmission error but not on arbitration loss. During the asynchronous window it behaves like native CAN. The analysis of the FTT-CAN protocol in terms of inconsistent message delivery reflects this duality and the probability of inconsistent message delivery depends on the average ratio (ρ) between the length of the asynchronous window (AW) and elementary cycle length. To simplify the analysis it is assumed that the trigger message is included in the asynchronous window and it can be retransmitted until the end of the asynchronous window. It is also assumed that synchronous messages retransmission upon error is disabled.

Equation 4.4, in the case of FTT-CAN, is transformed into4.7, because message duplicates may occur only during the asynchronous window.

$$P_{IMD}^{FTT} = P_{IMD}^{CAN}(onAW) = P_{IFO} \times (1 - P_{Fail}) \times \rho \tag{4.7}$$

Message omissions may occur both during the asynchronous and synchronous windows. In the synchronous windows the probability of omissions is the same as in TTCAN, while in the asynchronous windows the probability of omissions is the same as in native CAN. In this way, Equation 4.3, is transformed into 4.8.

$$\begin{aligned} P_{IMO}^{FTT} &= P_{IMO}^{CAN}(onAW) + P_{IMO}^{TTCAN}(onSW) \\ &= P_{IFO} \times P_{Fail} \times \rho + P_{IFO} \times (1 - \rho) \\ &= P_{IFO}(\rho \times P_{Fail} + 1 - \rho) \end{aligned} \tag{4.8}$$

The probability of inconsistencies in CAN, TTCAN and FTT-CAN are presented in Table 4.1. These results are based in the same assumptions of Rufino's work considering a node failure rate $\lambda = 10^{-4}$ failures per hour. In FTT-CAN, the ρ factor was set to 0.5 meaning that in average the asynchronous and synchronous windows have the same size.

	CAN		**TTCAN**	**FTT-CAN**	
bit error	$\Delta t = 5ms$			$\rho = 0.5$	
rate	IMD/h	IMO/h	IMO/h	IMD/h	IMO/h
10^{-4}	2.84×10^3	3.94×10^{-7}	2.84×10^3	1.42×10^3	1.42×10^3
10^{-5}	2.86×10^2	3.98×10^{-8}	2.86×10^2	1.43×10^2	1.43×10^2
10^{-6}	2.87×10^1	3.98×10^{-9}	2.87×10^1	1.43×10^1	1.43×10^1

Table 4.1: Estimated rates of IMO per hour in CAN, TTCAN and FTT-CAN

It should be stressed that these results are based in BER assumptions only, not in real experimental data. Nevertheless the probability of inconsistent message duplicates or/and omissions is too high to be ignored, specially in the case of TTCAN and FTT-CAN. The safe case scenario specification in the design process of any safety-critical system based on CAN, should consider the occurrence of inconsistencies regardless of their probability of occurrence. However, depending on that probability, the safeguard mechanisms can be designed at different levels and with various overhead penalties and efficiencies.

The pessimistic nature of the *bit error rate* assumptions together with the absence of public data concerning CAN bit error rates were the main drive to experimentally assess the CAN bit error. So, the next section presents some results of CAN *bit error rate* taken from an experimental setup specifically designed to measure it. The experimental data gathered also contributes to the definition of a more realistic CAN fault model to be used in the existing CAN response time analysis tools.

4.5 Assessing CAN Bit Error Rate

There are two possibilities to measure the number of inconsistent message omissions in CAN: a direct one and an indirect one. The direct approach relies on counting the errors, at the receiver, that affect only the last but one bit of a CAN frame. This approach cannot rely on COTS CAN controllers since they do not provide information about the bit position where an error has occurred, if after the ID field. The second approach, the indirect one, is based in the measurement of the *bit error rate* parameter and compute the probability of error in the last but one bit using Equation 4.1. Again, it is not possible to perform this experiment based on a COTS CAN controller since there is no way to count every single bit error. Notice that the CAN controller error counters are incremented in a non-linear way and in some cases there is no way to distinguish a bit error from a burst of errors, since the resulting increment in the error counters would be similar.

This calls for the use of specialized hardware able to detect, at the bitstream level, mismatches between the transmitted and the received bitstream. The bitstream comparison should be made at the receiver side just after the bit sampling stage to reflect the low level

nature of the *bit error rate*, a fundamental channel parameter that depends only on the relation between the bits transmitted and received at the sampling point, without any further protocol processing.

Besides the bitstream comparison, the experimental setup, depicted in Figure4.4 and in Figure 4.5, explicitly counts the errors in the last but one bit of the EOF. Using this setup both direct and indirect approaches are followed and two parameters can be directly extracted from the experiment: the upper bound of inconsistent message omissions (Possible IMO/h in Table 4.2) and the number of bit errors.

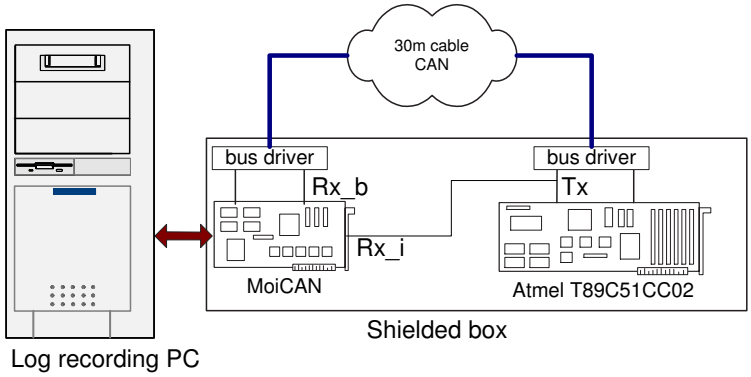


Figure 4.4: Experimental setup.

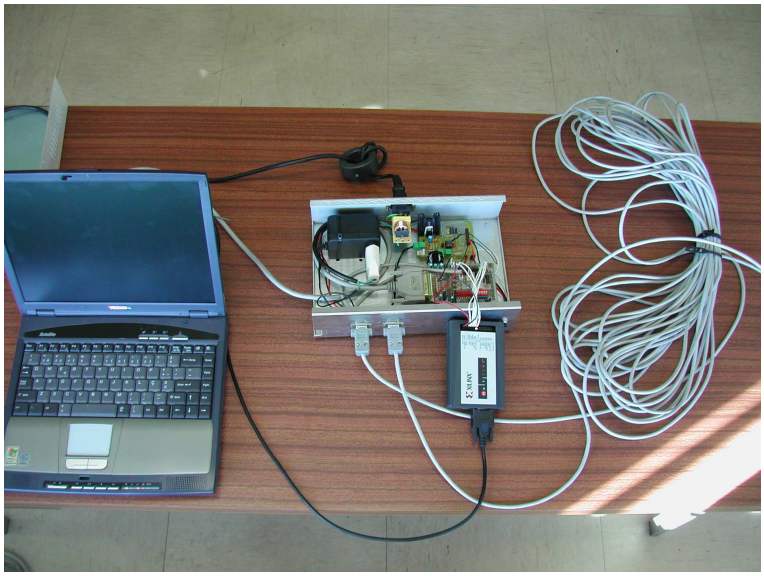


Figure 4.5: View of the experimental setup.

The heart of the instrument designed to monitor errors is the MoiCAN IP core [Oli03],

optimized for implementation in Xilinx FPGAs. MoiCAN is a subset of a CAN 2.0A controller with two operating modes: monitor and active. In monitor mode it watches the bus and receives frames without interfering with the bus in any way. In active mode, besides monitoring the bus, it also acknowledges frames, signals errors with error flags and reacts to overload flags, *i.e.* it acts as a standard CAN controller in the receiving mode only.

The MoiCAN core implements all CAN features directly related with frame reception. It also has a set of special purpose 8-bit registers: the Bit Error Counter register, the Stuff Error Counter register, the CRC Error counter register, the Form Error Counter register, the Acknowledge Error Counter register, the Last-but-one-bit Error Counter Register. MoiCAN also has a 24-bit register which is incremented every time a valid message is received and a 16 MHz 32-bit timer to timestamp every error. So, whenever an error occurs, its timestamp, the frame field where the error occurred and the field bit position are stored in a table with 256 entries. All these registers and the error table can be periodically polled via a synchronous microprocessor interface.

Notice that MoiCAN has not been formally submitted to CAN conformance tests. However a superset of MoiCAN, the CLAN [AO03][OFSF03], has been extensively used together with COTS CAN controllers with no problems being reported so far, increasing the confidence level on its conformance.

Besides the MoiCAN board, the experimental setup includes a transmitter board based on the Atmel T89C51CC02 CAN controller and a log recording PC, which connects to the MoiCAN via the parallel port. Both boards use identical CAN bus drivers, the Philips 82C250. The boards ensemble is depicted in Figure 4.6.

A particular feature of the Atmel T89C51CC02 CAN controller is the single shot transmission mode (no automatic retransmission upon error or arbitration loss). This transmission mode is not available in many CAN controllers, since it is not part of the CAN protocol (although it is mandatory for TTCAN).

Each frame transmitted by the Atmel CAN controller conveyed a four-byte message counter (sequence number), that is incremented every time a message is transmitted (either successfully or with errors). In a sense this counter passes the transmitter view of the network to the receiver because the transmitter always knows when a message is transmitted without errors (CAN frame validation rule). On the other hand, a receiver has no way to know if a message that was correctly received has been correctly transmitted (a last but one bit error could have occurred at the sender). In this way the receiver is able to compare the number of received messages with the sequence number of the received message, and to count the total number of messages lost during the experiments. This was the reason why single shot transmission mode was adopted in the experiments. However this scheme does not allow the detection of errors in the last but one bit. These errors are detected at the receiver side by specialized hardware and are stored in the Last-but-one-bit Error Counter Register.

The inclusion of a sequence number in every transmitted frame is also used to partially

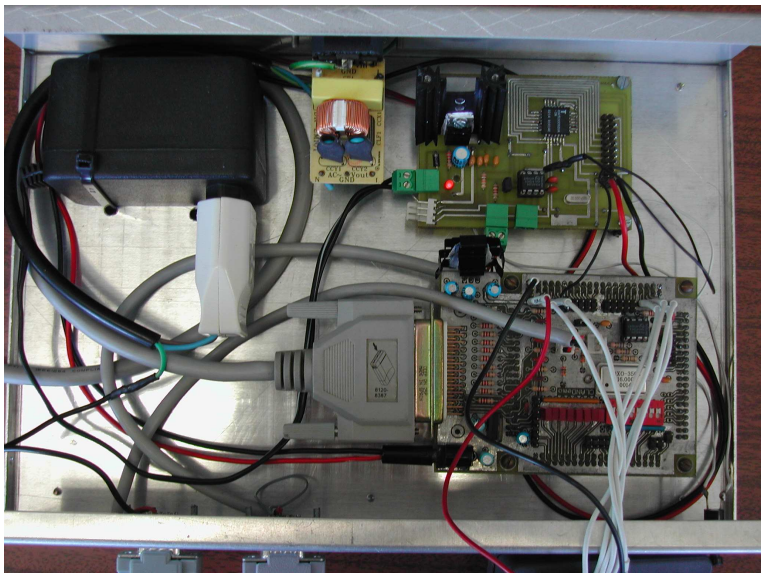


Figure 4.6: View of the metal box containing the MoiCAN board and the ATMEL controller board (the smaller one).

validate the MoiCAN core, because the difference between message's sequence number and the MoiCAN message counter should be consistent with the values stored in MoiCAN's error counter registers.

The CAN bitstream produced by the Atmel controller at 1 Mbps goes through a 30m CAN bus to the MoiCAN board. The same bitstream also goes via a 10cm internal shortcut to the MoiCAN board, where they are bitwise compared. MoiCAN board registers are polled every 10sec and their values are stored in a log file in the PC. It is assumed that transmission errors do not affect the 10cm shortcut, placed inside a shielded box.

The MoiCAN bit timing was divided into 8 time-quanta (tq), corresponding $1tq$ to the SYNC_SEG, $1tq$ to the PHASE_SEG1, $4tq$ to the PROP_SEG and $2tq$ to the PHASE_SEG2 [BOS91]. The Atmel controller transmits 8-byte data frames every $400\mu s$, using approximately 25% of the CAN bus bandwidth.

During all the experiments the MoiCAN core was in the active mode, i.e. it transmits acknowledge bits and error frames. This is so because in the passive monitor mode the MoiCAN core does not transmit acknowledge bits and thus the ATMEL CAN controller would transmit an error frame for every frame transmission, making impossible the occurrence of errors in the last but one bit of the CAN frame.

The experimental work was divided in two phases. The experiments performed during the first phase measured the bit error rate and the number of errors in last but one bit of the EOF field, over a long time interval in order to obtain statistically relevant results. The experiments conducted on the second phase were intended to assess the electromagnetic interference caused

by a mobile phone in several operational scenarios with different electromagnetic shielding properties. Each of these experiments lasted for a short time interval and their results cannot be directly extrapolated, specially in the cases were no errors were measured.

4.5.1 Experiments conducted over a long time interval

Three sets of lengthly experiments were conducted: one at the University laboratories (*benign environment*), other at a factory, near a high-frequency arc-welding machine (*aggressive environment*) and another at the factory production line (*normal environment*).

The tests were conducted at João R. Matos, S.A. (www.electrex.pt), a manufacturer of welding equipment. Figure 4.7 presents a view of one of several welding machines testing workbenches (*aggressive environment*). The CAN error measurements were carried out during several days. During this time several different models of welding machines were tested, positioned at an *average* distance of 2 meters from the measurement equipment.

Figure 4.7: Experimental setup illustrating the aggressive environment.

Figure 4.8 presents a view of the factory production line were several tests were also conducted over a period of several days. We considered this environment a *normal* one, although it is still disturbed by some electromagnetic interference of several welding machines. The difference from the aggressive environment is that they are located more than 20 meters away from the the CAN bit error rate measurement equipment.

The CAN cable was folded in order to make a winding with just one turn and the MoiCAN

Figure 4.8: View of the factory production line illustrating the normal environment.

and Atmel boards were shielded within a metal case.

The results of the experiments are presented in Table4.2. Notice that in the computation of the bit error rate, all errors within error bursts were accounted individually. The *Possible IMO/h* value in Table 4.2 represents the experimental upper bound for the real value of IMO/h, in case all last but one bit errors cause an omission. Notice that the value of *Possible IMO/h* was normalized for a network load of 100%.

Besides the bit error rate, Table 4.2 also presents a parameter, the interference rate, which accounts for the total number of interferences, i.e., single bit errors and error bursts, instead of bit errors only. The interference rate, i.e., the ratio of all interferences over all transmitted bits, gives a better insight of the real interference pattern. From the error model point of view, the total number of interferences and their frequency is probably more relevant than the total number of bit errors. An interference causes a message transmission fault, except for the cases where the interference corrupts the last but one bit of the EOF field, and the difference between a single error and a burst is just the extra bus inaccessibility time imposed by the error burst duration. To better illustrate the importance of this parameter consider the following scenario:

- 10^{12} bits were transmitted in a given channel.
- 10^{8} bit errors were measured.

	Benign environment	**Normal environment**	**Aggressive environment**
Bits transmitted	2.02×10^{11}	1.98×10^{11}	9.79×10^{10}
Bit errors[1]	6	609	25239
Errors in last but one bit	0	0	8
Bit Error Rate	3.0×10^{-11}	3.1×10^{-9}	2.6×10^{-7}
Interference Rate	3.0×10^{-11}	8.2×10^{-10}	6.3×10^{-8}
Possible[2] IMO/h	—	—	2.9×10^{-1}

Table 4.2: Experimental results ([1] Accounting for error bursts; [2] upper bound, if all last but one bit errors cause an omission).

- All these errors were part of just 100 error bursts.

The bit error rate of such channel would be 10^{-4}, a poor value, while the interference rate would be 10^{-10}. The question is to decide which of the parameters better characterizes this particular channel. A system designed based just in the bit error rate would require a considerable overhead in terms of bandwidth, to cope with an average of a bit error in every 10^4 bits transmitted, and possible bus redundancy, to increase the probability of error free transmission. In contrast, a system based in the interference rate assumption would necessarily be more resource efficient.

Table 4.3 presents the probability of inconsistencies both for CAN, TTCAN and FTT-CAN. The probability of inconsistencies presented in the first three rows of Table4.3 is based in the same bit error rate as in Rufino's work [RVA+98], while the other three rows correspond to the **measured** values of the bit error rate during the experimental work. The same assumptions of Rufino's work [RVA+98] were considered to compute the values of Table 4.3 considering a node failure rate of $\lambda = 10^{-4}$.

Observing Table 4.3, it seems that in native CAN the occurrence of inconsistent message omissions has a lower probability than previously assumed. At least in the environments considered in the experiments. In fact it is below the 10^{-9} threshold usually accepted for safety-critical applications [Kop97]. However, the probability of inconsistent message duplicates (messages are eventually delivered but they could be out of order) is still high enough to be taken into account.

Concerning both TTCAN and FTT-CAN, one cannot neglect inconsistent message omis-

	CAN		**TTCAN**	**FTT-CAN**	
bit error	$\Delta t = 5ms$			$\rho = 0.5$	
rate	IMD/h	IMO/h	IMO/h	IMD/h	IMO/h
10^{-4}	2.84×10^{3}	3.94×10^{-7}	2.84×10^{3}	1.42×10^{3}	1.42×10^{3}
10^{-5}	2.86×10^{2}	3.98×10^{-8}	2.86×10^{2}	1.43×10^{2}	1.43×10^{2}
10^{-6}	2.87×10^{1}	3.98×10^{-9}	2.87×10^{1}	1.43×10^{1}	1.43×10^{1}
2.6×10^{-7}	7.59	1.05×10^{-9}	7.59	3.79	3.79
3.1×10^{-9}	8.93×10^{-2}	1.24×10^{-11}	8.93×10^{-2}	4.46×10^{-2}	4.46×10^{-2}
3.0×10^{-11}	8.75×10^{-4}	1.22×10^{-13}	8.75×10^{-4}	4.37×10^{-4}	4.37×10^{-4}

Table 4.3: Estimated rates of IMO per hour in CAN, TTCAN and FTT-CAN.

sions because they are rather frequent. Nevertheless it should be mentioned that there is a mismatch between the expected (from equations 4.1 and 4.5) value of IMO/h (7.59) and the measured value (2.9×10^{-1}) in the aggressive environment experiment. Thus the probabilistic error model, based on the *bit error rate*, using the statistical data on errors in the last but one bit was not experimentally verified.

Table 4.4 presents the sizes of the error bursts observed both in the aggressive and in the factory floor environments.

In both cases, the observed errors are either single bit errors or 6-bit wide errors. Error bursts of different duration are quite rare. Since these error bursts were measured in an highly noisy environment, we conjecture that they are a consequence of the fault containment mechanisms of CAN, when the Atmel CAN controller reaches the bus passive state before the MoiCAN controller. In this case, when an error is detected, the Atmel CAN controller transmits a passive error flag (6 recessive bits) while the MoiCAN controller would transmit an active error flag (6 dominant bits). The error detection circuitry compares the passive error flag transmitted by the Atmel CAN controller, via the 10 cm wire, with the active error flag conveyed in the bus lines and detects a 6-bit error burst.

The electromagnetically noisy environment contributes to this conjecture in the sense that it could drive a CAN controller to an error passive state in a relatively short time interval, when the soldering machine is activated. Conversely, in between soldering machine activations it is expectable that the CAN controller would be driven back to an error active state. It is also conceivable that both CAN controllers could reach the error passive in different instants, giving their different views of the bus.

If, after further experiments with a modified version of the MoiCAN core, this conjecture proves to be correct, it will mean that these 6-bit error frames are artifacts that do not correspond to real error bursts, instead they correspond to single bit errors. This finding does not invalidate the conclusions taken from the experimental data because we are being

	Aggressive environment	**Factory floor**
Error burst (in bits)	Number of errors	Number of errors
1	2431	75
2	0	0
3	1	0
4	1	0
5	2	0
6	3712	88
7	31	0
8	1	0

Table 4.4: Error bursts size in the aggressive environment experiment and in the factory floor experiment.

pessimistic and the real value for the bit error rate would be even lower.

Table 4.5 presents the distribution of single errors and start of error bursts over the possible frame states of the MoiCAN state machine.

4.5.2 Experiments conducted over a short time interval

Several additional short experiments, intended to assess the interference caused by a mobile phone in the CAN network, were conducted at a University laboratory. The motivation to perform these experiments is related with the widespread use of mobile phones near CAN networks, e.g., in cars, and with the reporting of interferences, although not quantified, of these devices in CAN networks [HNP00][NYQS00].

Eight scenarios were tested lasting 120 seconds each, during which 10 phone calls were made. The difference between each test scenario was the mobile phone positioning, the EMI shielding of the measurement instruments (MoiCAN and ATMEL boards) and the CAN cabling. In the first experiment the mobile phone was put over a plastic surface that was covering the open metal box that contains the boards, as depicted in Figure4.9. In the second experiment the mobile phone was put 12 cm above the open metal box that contains the boards, as depicted in Figure 4.9. In the third experiment, the mobile phone was placed over the CAN sockets still with the metal box open (Figure4.10). In the fourth experiment the mobile phone was placed over the CAN cable 15 cm way from the open metal box (Figure4.10). In the fifth experiment, the mobile phone was placed over the closed metal box (Figure4.11). In the sixth experiment, the mobile phone was placed over the CAN sockets but this time with the metal box closed (Figure 4.11). During these six experiments, the 30 m shielded cable was used. In the seventh experiment the previous cable was replaced by a 1 m cable that was

	Aggressive environment		**Factory floor**	
Frame state	Single errors	Bursts	Single errors	Error bursts
INTERMISSION	544	3667	7	88
SUSPEND_TX	312	0	0	0
IDLE	0	0	47	0
SOF	704	0	0	0
ID	39	0	4	0
RTR	76	0	2	0
RSVD_BITS	100	0	1	0
DLC	344	0	0	0
DATA	179	0	12	0
CRC_SEQ	5	0	1	0
CRC_DELIMIT	0	6	0	0
ACK_SLOT	2	0	0	0
ACK_DELIMIT	11	17	0	0
EOF	45	57	0	0
OVLD_FLAG	0	1	1	0
OVLD_DELIMIT_WAIT	0	0	0	0
OVLD_DELIMIT	0	0	0	0

Table 4.5: Distribution of single errors and start of error bursts considering the states of the MoiCAN state machine.

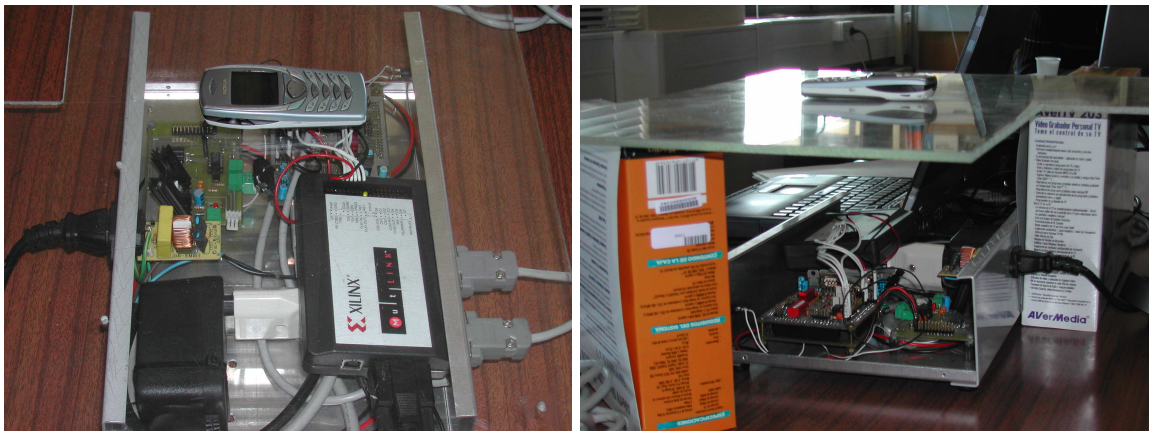


Figure 4.9: First experiment, on the left side, and second experiment, on the right side.

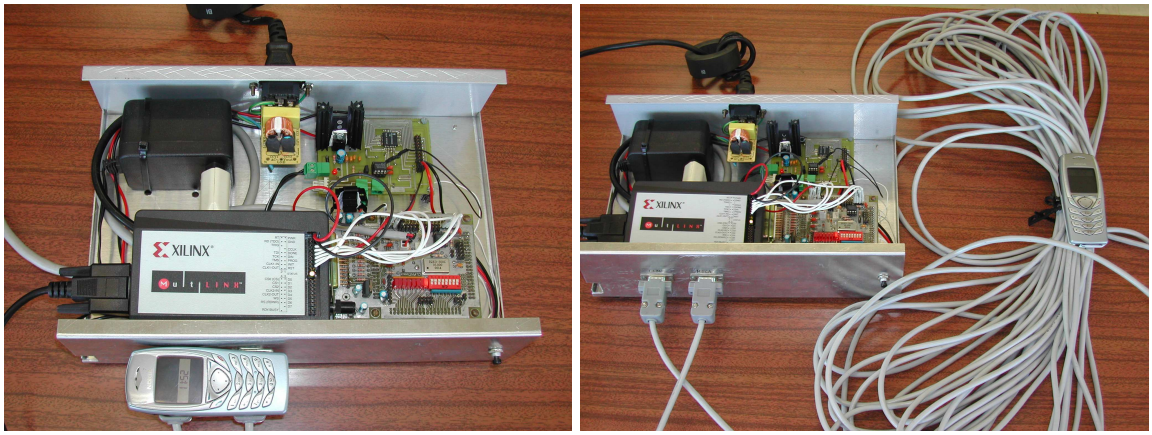


Figure 4.10: Third experiment, on the left side and fourth experiment, on the right side.

folded on top of the metal box and the mobile phone was placed over it (Figure4.12), while in the eighth experiment only one of the wires of the 1 m cable was folded (Figure4.12).

These experiments were not intended to measure the bit error rate in each of those scenarios, instead they were designed to illustrate the close interdependency between the EMI shielding and the number of transmission errors. Nevertheless, the data collected in these experiments will be presented in terms of total number of errors, error burst size distribution and error location, in each experiment.

Table 4.6 presents the sizes of the error bursts, the total number of errors and the number of interferences, while Table 4.7 presents the distribution of single errors and start of error bursts over the possible frame states of the MoiCAN state machine and the errors in the last but one bit. These tables only contain information concerning experiments one and three, since no errors were detected in the other six experiments.

Figure 4.11: Fifth experiment, on the left side and sixth experiment, on the right side.

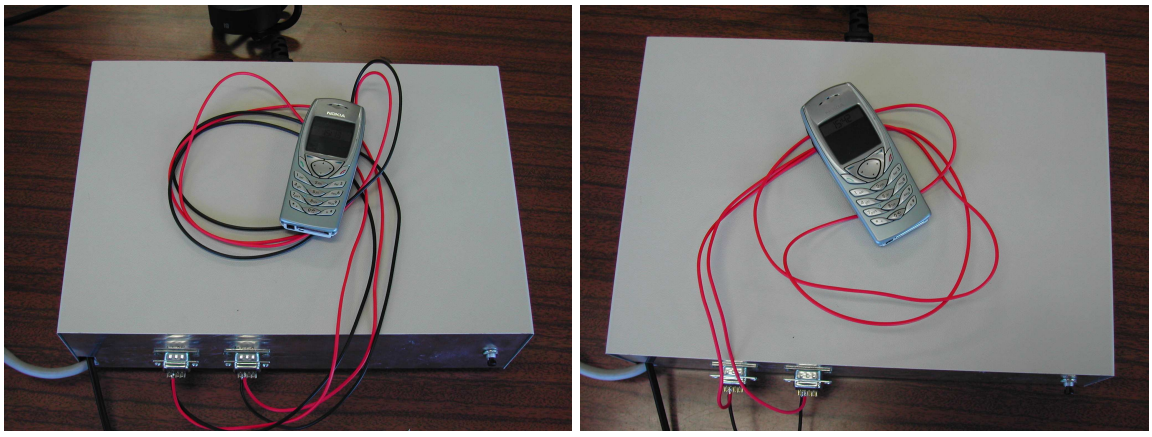


Figure 4.12: Seventh experiment, on the left side and eighth experiment, on the right side.

The main conclusion that can be draw from these experiments is that electromagnetic interferences caused by a mobile phone are quite severe if the CAN controllers are not fully shielded and the mobile phone is very close to the them. Notice that the assumption about no errors being induced in the 10 cm wire connecting the Atmel board to the MoiCAN board no longer holds when the metal box is open. In this case the Tx signal that goes on the 10 cm wire is much more susceptible to interferences that the CAN wires that carry differential signals.

These results also show that the interferences induced in the cabling alone (experiments 4, 7 and 8) do not cause transmission errors. This last conclusion needs to be put in perspective, since the experiments were run only for a very short time interval. Notice also that the interference pattern induced by the mobile phone differs from the one found in the industrial environment. Most of the errors caused by the mobile phone occur within the IDLE frame state while in the industrial environment they occur in the INTERMISSION frame state.

Error burst (in bits)	**Experiment 1**	**Experiment 3**
1	1233	556
2	16	8
3	0	0
4	4	0
5	2	0
6	30	5
7	1	0
8	0	0
9	1	0
Errors	1471	594
Interferences	1357	571
Errors (last but one bit)	0	0

Table 4.6: Sizes of the error bursts, total number of errors and number of interferences.

4.6 Fault Hypothesis

Having described the impairments to CAN and FTT-CAN dependability and presented some results, based in experimental data, of their probability, it is now possible to state the fault hypothesis that will be used throughout this work, towards a dependable FTT-CAN system.

Concerning channel faults, only *transient* faults that change the value of, at least, one bit

	Experiment 1		Experiment 3	
Frame state	Single error	Burst	Single error	Burst
INTERMISSION	23	152	1	29
SUPS_TX	0	0	0	0
IDLE	1161	35	545	5
SOF	0	0	0	0
ID	5	4	4	2
RTR	0	0	0	0
RSVD_BITS	0	0	0	0
DLC	0	0	0	0
DATA	37	35	4	1
CRC_SEQ	8	12	2	1
CRC_DEL	0	0	0	0
ACK_SLOT	0	0	0	0
ACK_DELIMIT	0	0	0	0
EOF	0	0	0	0
OVLD_FLAG	0	0	0	0
OVLD_DELIMIT_WAIT	0	0	0	0
OVLD_DELIMIT	0	0	0	0

Table 4.7: Distribution of single errors and start of error bursts considering the states of the MoiCAN state machine.

are considered. The specific fault scenario reported in [PMJ00], which causes an inconsistent transmission that the transmitter is not able to detect, is not considered in this work.

Also, no masquerading faults are considered, e.g. a malicious node forcing the transmission of incorrect trigger messages.

Although channel bit error statistics have been collected, no special assumption is made about the frequency or the duration of channel faults. Thus, it is assumed that inconsistent message omissions and duplicates may occur. The reasons for this are twofold, in the first place faults induced in the channel are heavily dependent on the particular environment where the system operates. Secondly, the most relevant result that emerged from the bit error rate experiments, i.e., inconsistent message omissions below 10^{-9} occurrences per hour, is applicable to systems based on CAN, only.

Similarly to the master node, the transmission medium is also a single point of failure of FTT-CAN. The mechanisms proposed in this work do not aim to tolerate permanent faults of the transmission medium, such as a partition of the cable, and therefore these faults are not considered. Notice, however that the bus may easily be duplicated, e.g., using Rufino's

Columbus egg idea [RVA99b] to handle physical bus partition.

Concerning physical faults of the nodes, both masters and slaves are assumed to exhibit fail-silence failure semantics. This means that they can only fail by not issuing any message to the network. For the masters, this assumption is substantiated by their internal redundancy. In contrast, the internal structure of the slaves does not enforce this assumption. Nevertheless, crash failure semantics can be assumed for them as similar techniques could be used in order to achieve this property.

The clock synchronization problem is not addressed in the dissertation and it is assumed that nodes, both masters and slaves, are always synchronized. This assumption is based in the fact that the trigger message transmitted by the active master, besides conveying the scheduling information, also acts as a synchronization mark to all network nodes. That is, the master node is also the time master and it is assumed that in between two consecutive trigger messages (typically $5msec$ to $10msec$) the clock timers of each node do not diverge more than a negligible amount of time. Notice that FTT-CAN time granularity is quite coarse since it corresponds to the elementary cycle duration. Slave nodes do not require a global time base to operate, they only need a simple timer to enforce the separation between the event- and time triggered phases of the protocol, i.e., the asynchronous and synchronous windows respectively.

4.6.1 System properties

Considering the previous fault hypothesis and the mechanisms of CAN and FTT-CAN described in Chapter 3, some underlying properties related to fault-tolerance aspects of both CAN and FTT-CAN are devised.

CAN.p1 Whenever a receiver rejects a frame, the error signaling mechanisms of CAN compel the transmitter to reject this frame as well. In other words, one transmitter considers that a frame has been successfully transmitted only if this frame has been consistently received by the rest of the nodes in the network.

CAN.p2 Whenever a node attempts to transmit a CAN frame and it does not succeed due to an error, then some correct CAN controllers may receive the frame correctly while others may reject the same frame. This inconsistency can only occur if there was an error in the last but one bit of the frame. In all other situations, all CAN controllers consistently reject the frame. The following two properties are direct corollaries of this one.

CAN.p3 Upon message retransmission caused by a network error, a correct CAN controller may receive the same frame correctly more than once.

CAN.p4 Whenever a correct CAN controller receives a valid frame, without further information there is no way to tell whether the sender successfully sent that frame or failed in sending it due to network errors.

In what concerns FTT-CAN, the transmission of the Trigger Message is particularly relevant since it conveys the master view of the network, mainly the EC-schedule, and serves as a synchronization mark. Its transmission mechanism is the following: the active master transmits the TM and, in case of error, tries to retransmit it during a given window (typically about half the EC duration). Eventual backup masters, required to circumvent the single point of failure of the FTT-CAN master node (this issue will be detailed further on Chapter 6), try to transmit a TM with a small delay relative to the active one and in single-shot mode with immediate abort after transmission request, i.e. transmission takes place if bus is idle, only. Among several possible backup masters, only one can succeed and only if the active master failed, resulting in the following properties:

FTT-CAN.p1 Only one master can transmit a trigger message successfully within each elementary cycle.

FTT-CAN.p2 One backup master can succeed in transmitting the trigger message only if the active master is faulty. Therefore, there is only one active master at a time.

FTT-CAN.p3 Whenever the active master fails, the backup master which wins arbitration and becomes the active master is the one with the shortest replacement delay. If more than one backup master has the same replacement delay, the one which wins arbitration is the one that attempts to transmit the TM with the lowest identifier (a pre-configurable parameter in each master).

4.7 Achieving fault-tolerance in FTT-CAN

The FTT-CAN protocol was originally developed to fulfill three basic requirements: timeliness, flexibility and efficiency [AFF98] [APF02]. This was achieved by combining the advantages of time- and event-triggered paradigms and providing flexibility to the time-triggered traffic. Without discussing the merits of such an approach neither presuming a causal effect, recall that both TTCAN and FlexRay proposals, which are posterior to FTT-CAN, are somewhat based in the same basic idea of combining time- and event-triggered communication paradigms. This reveals a clear trend towards communication systems capable of supporting both communication paradigms. Still, FTT-CAN goes a bit further by allowing time-triggered messages to be scheduled dynamically and online, in contrast with both TTCAN and FlexRay where time-triggered massages are **static** and scheduled at **pre-runtime**. This fact makes the development of fault-tolerant mechanisms to TTCAN and FlexRay quite straightforward because there is an *a priori* common knowledge of the message scheduling by all nodes. This is not the case in FTT-CAN, however, as it will be shown later, it is possible to build fault-tolerant mechanisms for FTT-CAN that preserve the protocol inherent flexibility particularly concerning the time-triggered traffic. The complexity of those fault-tolerant mechanisms is comparable with the solutions adopted in TTCAN and FlexRay.

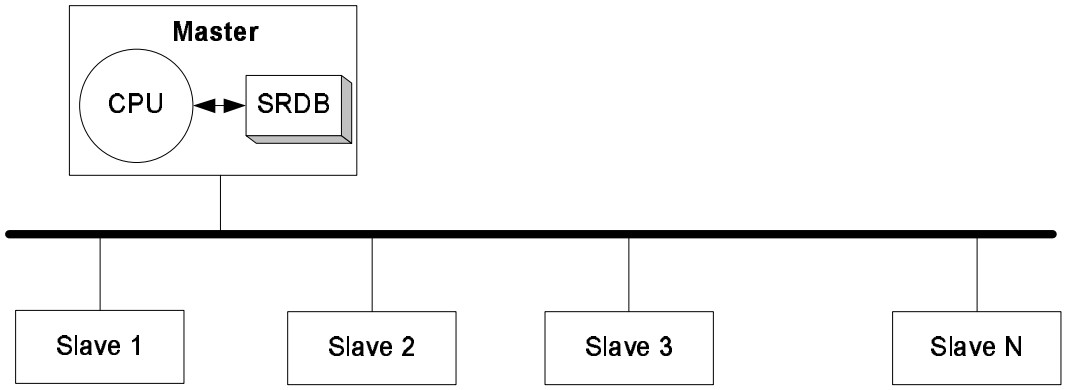


Figure 4.13: FTT-CAN basic architecture.

The initial FTT-CAN architecture emerging from the protocol specification considered a single master, responsible for the network traffic scheduling, admission control of new message streams or modifications to the current ones and for the cyclic transmission of the trigger message. Slave nodes could transmit event-triggered messages during the asynchronous window and master scheduled time-triggered messages during the synchronous window. This basic architecture is depicted in Figure 4.13.

The first step towards the definition of a fault tolerant FTT-CAN architecture is the identification of the impairments to dependability of the original FTT-CAN architecture [AFF98][APF02] as well as of native CAN. The previous sections have identified those impairments and presented some new experimental data, related with the probability of inconsistent message omission. Experimental data indicates that the probability of inconsistent message omissions, which depends on the channel bit error rate, might be substantially lower than previously assumed. In fact, in the experiments it is below the 10^{-9} occurrences per hour, the commonly accepted threshold for safety-critical applications. This fact enables the direct use of native CAN in safety critical applications without requiring sophisticated and bandwidth inefficient atomic broadcast algorithms. Since the asynchronous messaging system of FTT-CAN preserves all native CAN properties, we conjecture that FTT-CAN asynchronous messages do not require an atomic broadcast algorithm to achieve consensus among FTT-CAN nodes. Notice, however, that this conjecture is only valid within the reported experiments scenarios and the experimental results do not pretend to be universally applicable. The same experimental results have also shown that the CAN bit error rate is high enough to make FTT-CAN synchronous messages very susceptible to inconsistent message omissions.

Other impairment to FTT-CAN dependability is related with the failure semantics of the master and the slave nodes. The fault hypothesis assumes that nodes exhibit a crash failure semantics, i.e., nodes can only fail by not issuing any message to the network (fail-silence failure mode). Unfortunately this does not match standard CAN and FTT-CAN nodes, thus,

some mechanisms must be developed to enforce such behavior.

Besides the inconsistent message transmission and the failure semantics of FTT-CAN nodes, another obvious problem with FTT-CAN is the single point of failure nature of the master node. If the master node fails to transmit trigger messages, transmit them out of time or with erroneous contents, then all network activity could be seriously compromised or even disrupted.

Having analyzed CAN and FTT-CAN impairments to dependability and defined an adequate fault hypothesis characterizing the possible fault scenarios that an FTT-CAN based system is supposed to tolerate, one can now design fault-tolerant mechanisms to limit the impact of such faults. The faults included in the fault hypothesis are handled by deterministic fault tolerant mechanisms that enforce a continued correct system service. On the other hand, the faults that are not considered by the fault hypothesis, at the design stage, can cause severe consequences with unpredictable outcomes. To handle these faults, two approaches are typically adopted. Either the system is driven into a static and safe position or, for situations that require a fail-operational behavior, best-effort strategies must be used, to drive the system back to a correct state, usually referred to as *never give up strategies*.

The available spectrum of techniques to tolerate the faults included in the fault model is, as referred in Chapter 2, quite broad, ranging, for example, from distributed consensus algorithms to fully replicated infrastructures with design diversity and voting. Hence, the particular configuration of a system can be more or less complex and fault-tolerant as desired by the system designer, knowing that the more faults a given system tolerates the more expensive it will be.

As it was previously referred, in the specific case of FTT-CAN, an additional issue exists: its flexibility in terms of the time-triggered traffic. Opposed to FlexRay, TTP/C or TTCAN, where such traffic is static and defined at pre-run time, in FTT-CAN it is scheduled online, supporting dynamic communication requirements.

Allowing flexible communication requirements in a real-time distributed system brings up some concerns regarding safety, since a change in communication requirements can possibly lead to a network overload and consequent timing failures. Furthermore, if the communication requirements can change online and unboundedly, it is not possible to use *a priori* knowledge to distinguish correct transmissions from wrong ones. The use of *a priori* knowledge is of utmost importance in fault-tolerance techniques, to distinguish between what is correct and what is wrong. However, if the on-line requests to change the communication requirements are admissible only within strict boundaries, both temporally and in value, then it will become possible to guarantee the continued safe and timely behavior of the network. This requires the filtering of the requests in order to accept only those that conform to specifications. In this way, the system will still be flexible with respect to the communication requirements although the flexibility is limited to an extent up to which safety is not jeopardized.

The impairment caused by the single point of failure formed by the master holding the

SRDB and the traffic scheduler can be circumvented using replication, with one or more similar nodes acting as master backups. In this way, as soon as a missing trigger message is detected, a backup master comes into the foreground and transmits it, maintaining the communication.

Replication is a common technique to provide fault-tolerance and it is used, for example, in TTCAN to cope with failures of the time master. However, master replication in FTT-CAN includes an extra complexity resulting from the dynamic nature of the SRDB. Since the system requirements, replicated at all masters SRBS, are flexible and may evolve over time, there is a potential for inconsistency in the replicated SRDB images and particularly in the SRTs (Synchronous Requirements Table), compromising the replica determinism requirement. Two situations are particularly relevant:

- During an asynchronous startup/restart a master node may lose the contents of its SRT. This calls for the definition of a protocol to transfer the SRT from the active master to the unsynchronized one.
- During the processing of an update request to change the communications parameters, it must be ensured that all replicas process the same request, in the same order and commit the request synchronously. This calls for the definition of an adequate protocol to enforce consensus preventing a master to accept a request while others may reject it or different masters committing a request at different instants.

Apart from these situations and during standard operation mode, there are no reasons for the replicas to diverge. To cope with these issues, the master node replication in FTT-CAN is addressed by means of four mechanisms:

1. A policing mechanism that allows backup masters to detect loss of synchronization.
2. A synchronization protocol to allow out-of-sync backups to re-synchronize in a short interval.
3. An agreement protocol to handle SRT updates in order to minimize loss of synchronization during SRT update requests.
4. A master replacement mechanism upon active master failure.

Enforcing Master's fail-silence assumption both in time and value domains calls for the internal replication of the SRDB and the traffic scheduler in two different CPUs and compare both outputs in terms of value and timing. The trigger message is issued only if both schedule outputs, i.e. EC-schedules, are identical and produced within a narrow time window.

Slaves should also be fail-silent and although one could adopt the same mechanism used in master nodes, that would be expensive. Thus, slave nodes fail-silence enforcement both in time and value domain should only be adopted in special cases where the slave node information

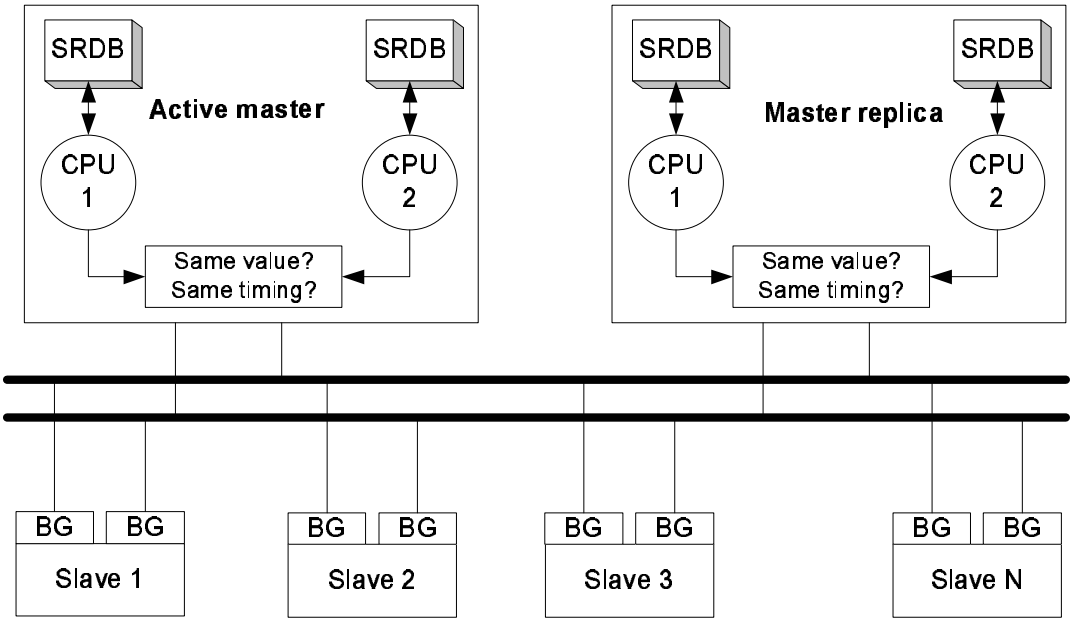


Figure 4.14: Fault-tolerant FTT-CAN architecture based on a replicated broadcast bus, master replication and bus guardians.

(value **and** timing) is absolutely essential. In other cases, limiting slave nodes ability to transmit uncontrollably will suffice. This corresponds to enforce fail-silence behavior in the time domain only, and it could be accomplished by a bus guardian.

Apart from the considerations above, it is also important to refer that the proposed architecture may also include replicated transmission paths in the communication system. This is important for the case in which the physical partition of the communication system must be tolerated. In this case, every node connects to both paths and transmits simultaneously on both, too. However, errors or physical partitioning may disturb the synchronization of both paths. Upon reception, a specific driver gets rid of duplicated messages by using messages identification and a maximum separation interval between the arrivals in both paths. As it was referred in the previous section bus replication is not considered in this work. Nevertheless, for the sake of completeness the proposed fault-tolerant FTT-CAN architecture already includes that feature, as depicted in Figure 4.14.

Given FTT-CAN flexibility with respect to the traffic scheduling, arising from the master node ability to schedule network traffic according to several policies and to accept runtime requests to change the message parameters, the bus guardians should also be flexible to adapt to evolving schedules. From a slave perspective, a schedule is valid only within the scope of an elementary cycle, thus the bus guardian policing a node only needs to be aware of the node schedule in an EC by EC basis. In this way the node and its bus guardian should decode the trigger message contents in parallel and the bus guardian should block any unscheduled

transmission from the node.

Possible electromagnetic interference or other source of errors may cause a fault in message transmission with the correspondent message omission. Depending on the omission location within the EC, several schemes capable of recovering from such situations need to be designed. If the omission occurs in the trigger message transmission it is possible to retransmit it during the trigger message transmission window in order to remove the omission. If a synchronous message is omitted, this can be detected by the absence of answer to the trigger message in the respective EC. In this case, the master may use fault-tolerant scheduling techniques to try to recover the missing message, e.g. by accounting for possible retransmissions and rescheduling the message again within the deadline, if possible. The detection of missing synchronous messages can be also used to implement a membership service for slave nodes. In what concerns masters, a specific membership service is implemented based on a polling mechanism. An omission of an asynchronous message could be removed by retransmitting the omitted message as in CAN. These schemes are only valid in case of transient interferences that cause sporadic message omissions. In order to tolerate permanent node failure a replication scheme needs to be adopted. This has already been referred for the master node but it also applies to slave node producing critical information.

4.8 Conclusion

This Chapter presented an overview of CAN and FTT-CAN impairments to dependability. One of the main concerns is the problem of possible inconsistent message duplicates and omissions when a fault occurs in the last but one bit of a CAN frame. The impact of inconsistent message duplicates/omissions depends on CAN bit error rate. There was no public data available on this issue and the initial bit error rate assumptions seemed to be rather pessimistic, given the CAN electrical physical layer properties. This was the main motivation to experimentally assess the CAN bit error rate. Experimental results have confirmed the pessimistic nature of the initial bit error rate assumptions since measured values are, in fact, substantially lower than previously assumed. An important result is that the number of inconsistent message omissions per hour in an aggressive environment may be below the 10^{-9} threshold commonly considered for safety-critical systems. This result, if confirmed with more experiments in typical application scenarios, makes hardly justifiable the existence of algorithms to enforce atomic broadcast in CAN. Although inconsistent message omissions are rare, the number of inconsistent message duplicates must be taken into account, but, in this case the mechanisms to account for duplicates can be made quite simple and light.

However, the considerable number of inconsistent message duplicates in CAN is a serious impairment to both TTCAN and FTT-CAN, since every such duplicate is transformed into an inconsistent omission because message retransmission is disabled (in FTT-CAN synchronous windows retransmissions may be allowed in a controlled way).

Based in known CAN impairments to dependability, described over the years in the literature, and in the experimental work presented in this Chapter, a fault hypothesis with no special assumptions about the frequency or the duration of channel faults was also presented.

The FTT-CAN impairments to dependability presented in this Chapter are the main motivation to the development of fault-tolerance mechanisms able to support the use of FTT-CAN in safety-critical systems. An architecture to enforce fault-tolerance in FTT-CAN concluded this Chapter. This architecture assumes fail silence nodes, master replication, bus redundancy and a set of other mechanisms to support these assumptions.

Chapter 5

Handling Message Omissions

5.1 Introduction

Possible electromagnetic interference or other source of errors may cause a fault in message transmission, possibly leading to a message omission. In case that omission is transient, it might be removed by means of the automatic retransmission upon error of CAN. Notice that the omission caused by an error in the last but one bit of a CAN frame is included in the previous case, despite some nodes possibly receiving and others not receiving the message, causing an inconsistent message omission. A message retransmission, necessary to recover from the omission, may influence the timeliness of other messages and, in a limit situation, a single transmission error could compromise the timeliness of all messages in a domino like effect.

In the case of FTT-CAN, the temporal isolation enforced by the protocol between the synchronous and asynchronous windows, by suspending all transmission activity at the end of each window, guarantees that no domino effect happens across those windows in case of network errors. In this way, errors in the asynchronous window will cause message retransmission possibly extending to subsequent asynchronous windows, but without interfering with the timeliness of the synchronous messages. Notice that, despite the time gaps between the consecutive asynchronous windows, FTT-CAN enforces that message arbitration and retransmissions occur in a logically continuous manner, so that priority inversions do not occur in the transitions between ECs. The influence of transmission errors during the synchronous windows is also bounded because automatic message retransmission upon error is disabled, as it will be discussed later, thus leading to message omissions. From this brief discussion it follows that asynchronous messages behavior in FTT-CAN is similar to native CAN while the behavior of synchronous message is more peculiar since messages are not automatically retransmitted upon transmission error.

A non-faulty message transmitter is always aware, by property CAN.p1 of Chapter4, of a message omission. Thus, and depending on the omission location within the EC, several

schemes capable of recovering from such situations should be designed.

For example, if an omission occurs during the trigger message transmission it would be desirable to retransmit the TM during, at least, a given retry window in order to attempt removing the omission. If a synchronous message is omitted, this can be promptly detected by the master monitoring the synchronous traffic. In this case, the master may use fault-tolerant scheduling techniques to try to recover the missing message, e.g. by accounting with time for possible retransmissions and rescheduling the message again within the deadline, if possible. Another possibility to circumvent the occurrence of a synchronous message omission is to add extra time, at scheduling time, to the synchronous window, according to some fault model, to accommodate possible message retransmissions without violating the schedule. An omission of an asynchronous message, as referred before, can be removed simply by retransmitting it, as in native CAN, provided the sender does not crash in the meantime.

These schemes are only valid in case of transient interferences that cause sporadic message omissions. In order to tolerate permanent node failure a replication scheme needs to be adopted both for slave and master nodes. The issue of enforcing master replica determinism will be addressed in Chapter 6. This Chapter presents mechanisms to handle asynchronous, synchronous and trigger message omissions, based on both spatial and time replication.

5.2 Handling trigger message omissions

The mechanisms proposed to handle trigger message omissions are twofold and depend on the nature of the failure, either transient, where a temporal redundancy scheme is adopted (controlled retransmission), or permanent, where a spatial redundancy mechanism (replication) is used. Both mechanisms are used in parallel, however, the temporal redundancy scheme has a higher priority than the one based in replication preventing a master replica to replace the current active master in the case of a transient interference. The master is substituted when it fails by not issuing a TM, only.

5.2.1 Transient trigger message omissions

In case of error during the transmission of the TM, the mechanism proposed relies on its retransmission, but confined to a window called the Trigger Message Transmission Window (TMTW), as depicted in Figure 5.1-A. The length of this window is configurable at startup time. Given the temporal validity of the TM, i.e., the temporal validity of the EC-schedule it conveys, which is one EC, it becomes meaningless to transmit it close to the end of the cycle. Notice that, to maintain the coherency of the temporal framework of the synchronous messaging subsystem, the next TM is not delayed and the master tries to keep its periodicity.

This controlled TM retransmission scheme also maintains the temporal isolation between the asynchronous and synchronous protocol phases by computing the synchronous window starting time relatively to the reception of the next TM. A slave node, after receiving a TM,

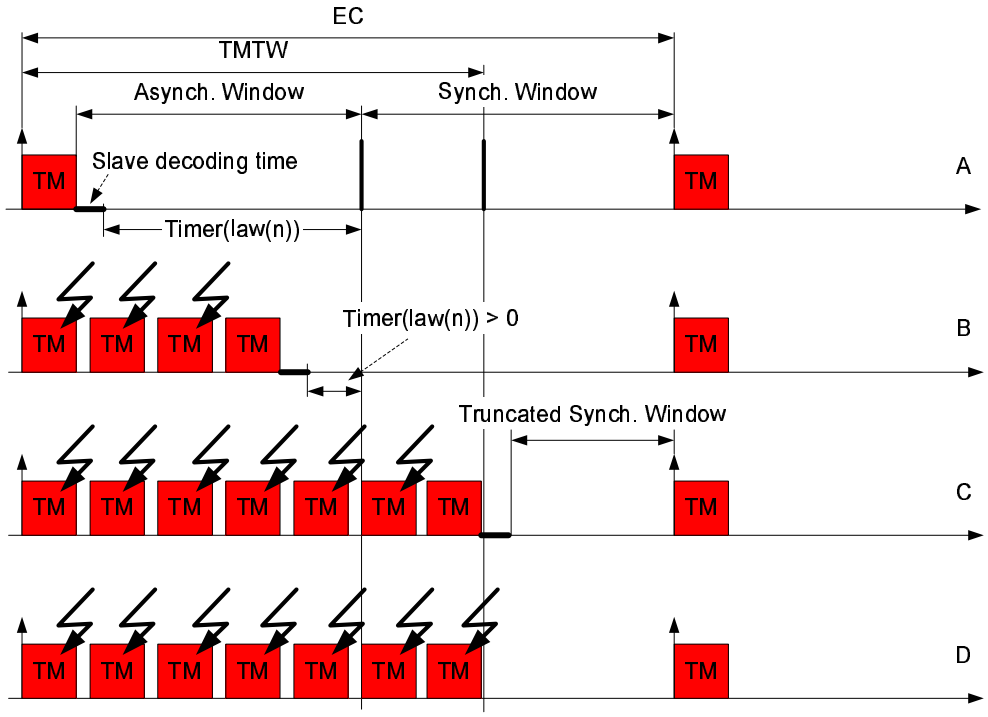


Figure 5.1: Controlled retry mechanism used to transmit the trigger message.

takes some time to decode it, both to extract the scheduling information and the length of the asynchronous window ($law(n)$). A timer (Timer($law(n)$)) is set with the length of the asynchronous window extracted from the TM, which is used to enforce temporal isolation between protocol phases. Notice that each slave node maintains another timer which periodically generates an interrupt during the expected time window corresponding to the reception of the TM, the EC timer. This interrupt is used to open the asynchronous message transmissions during the TM transmission in order to prevent priority inversions of the asynchronous messages. If the TM is not received when expected the slave node will attempt to transmit any pending asynchronous message, however it will fail to do so because the master node is also attempting to transmit a higher priority message, the trigger message, and will win the arbitration process. Thus the slave node will eventually receive a TM and in order to compute the effective LAW it only has to subtract the current value of the EC timer from the LAW value conveyed in the trigger message.

In the scenario depicted in Figure 5.1-B, the active master only succeeds to transmit the trigger message at the fourth attempt, thus the length of the asynchronous window is reduced, but without interfering with the synchronous messaging subsystem.

In the scenario depicted in Figure 5.1-C, the active master only succeeds to transmit the trigger message at the seventh attempt, but still before the end of the TMTW. In this case and since the effective LAW is negative, the synchronous window will be truncated and only

the sub-set of the synchronous messages with the highest priority, scheduled for that EC, that fit in the remaining time will be effectively transmitted.

Another possible scenario, illustrated in Figure 5.1-D, corresponds to the case where the transient interference spans across all the TMTW causing consecutive trigger message transmission errors. In this case the active master does not fail, but it is unable to successfully transmit the trigger message during the TMTW. Thus, several or all network nodes, either masters or slaves, will not receive a valid trigger message. Consequently, slave nodes cannot transmit during the respective EC. However, if an error in the last but one bit occurs during the TMTW some nodes may receive a valid trigger message and attempt to transmit synchronous messages, while others would not. This has no serious consequences in the protocol, because if a backup master receives a valid trigger message it just stops trying to become active in this EC. For the case of a slave node, it will begin normal protocol operation but this process will be interrupted by the reception of the next trigger message. The active master that was unable to transmit the trigger message during the TMTW will compute the next elementary cycle schedule and will transmit it when the EC timer expires. Backup masters that have received the trigger message out of the expected receiving instant or have not received a trigger message or were not able to transmit their own, increment their EC count, and wait for the next EC timer expiration to try sending the next TM. This scheme avoids unnecessary master replacements and ensures that only one backup master can succeed in transmitting the trigger message and only if the active master fails by crashing. It also allows coping with interferences that last for more than one EC. When the TM is omitted in a given EC, all masters maintain their previous state, active or backup.

5.2.2 Master replication and replacement

Master replication and replacement is required to tolerate permanent failures of the master node. Recall that according to the fault hypothesis the master node is fail silent and so when it fail it stops transmitting.

The master failure detection and replacement technique consists in making the backup masters try to transmit the TM a short delay, the replacement delay (Figure 5.2), after the expected start of the TM transmission by the active master. The masters use a single shot transmission by issuing an abort request immediately after the transmit one. If the active master is already transmitting a trigger message on the bus, then the abort request is successful, otherwise, if the backup master effectively started transmitting the TM, the abort request fails and the trigger message produced by the backup master is effectively transmitted. In the former situation, the backup master suffers a CAN receive interrupt and stays in the same state. In the latter situation the backup master suffers a CAN transmit interrupt and changes its state to active.

In case of error during the transmission of the TM, all masters, active and backup, retry

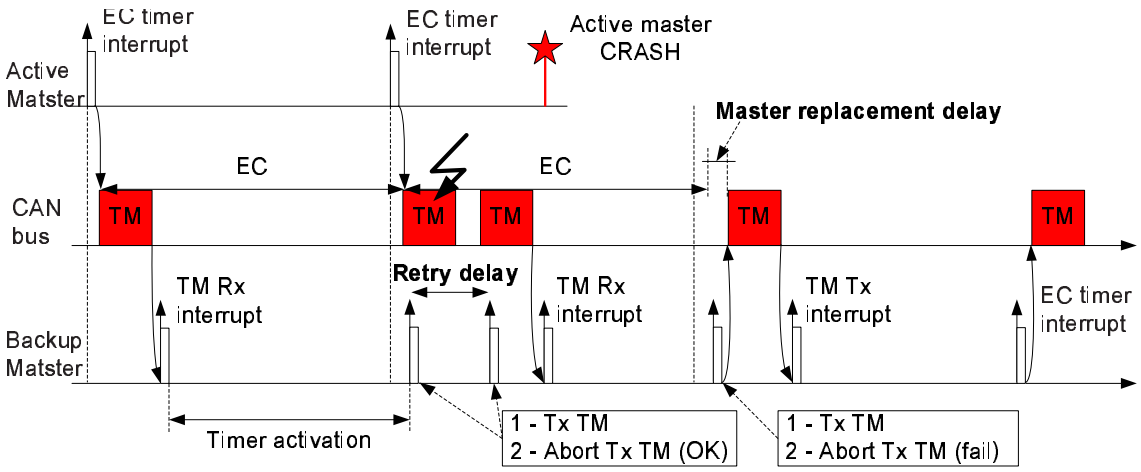


Figure 5.2: Master replacement process.

transmitting it in single shot mode after a short delay, the retry delay (Figure5.2).

This controlled retry mechanism gives all masters the time required to check the reception of a TM before attempting to send it. Consequently, only one TM is successfully transmitted in each EC. This process can be repeated several times during the Trigger Message Transmission Window (TMTW). If no trigger message is successfully transmitted during the TMTW all masters increment their EC count, and wait for the next EC timer expiration to try sending the next TM. This allows coping with interferences that last for more than one EC. When the TM is omitted in a given EC, all masters maintain their previous state, active or backup.

If there are several backup masters present in the network the situation is similar, since possible backup master competition is handled by the native CAN arbitration. This implementation is quite efficient since the master replacement delay is a fraction of the trigger message duration, and so the induced jitter due to master replacement is low.

A simplified version of this replacement mechanism, without the controlled retransmission, was successfully implemented in the CANivete [FSMF98] system, with a master replacement delay of $300\mu sec$ [FPAF02].

5.3 Spatial redundancy to handle synchronous and asynchronous message omissions

A slave node responsible for transmitting critical data, e.g., an alarm sensor, needs to be replicated in order to deliver continuous service upon primary slave failure. Since slave nodes may transmit either synchronous or asynchronous messages or both types of messages, the slave replication scheme must cover all these cases.

As it was discussed in section 5.2, there are two possible types of redundancy that could be

used in different contexts: temporal and spatial redundancy. Temporal redundancy consists in retransmitting messages over time while spatial redundancy implies the physical duplication of nodes. Temporal redundancy tolerates transient interferences that inhibit a node to transmit during a short time interval. Spatial redundancy, on the other hand, tolerates node crashing. As it was referred in Chapter 2, it is impossible to detect node failures in asynchronous distributed systems without relaxing the asynchrony assumptions, so we assume that asynchronous messages have a maximum inter transmission time. Moreover, it is difficult to instantly distinguish a transient interference from a node crash in an asynchronous system even with maximum inter transmission times. In this case a node transmitting a late message due to, e.g., electromagnetic interference, could be considered crashed by a failure detector just after the maximum inter transmission time has elapsed, causing an unnecessary node replacement.

This section presents a scheme to account for spatial redundancy of FTT-CAN slave nodes, while the next section will present a mechanism that implements temporal redundancy for transmitting synchronous messages. The last section of this Chapter discusses the particular case of atomicity of asynchronous messages that may convey safety-critical data, e.g., alarms or FTT-CAN protocol management information.

Depending on the type of traffic a given node transmits, two schemes to support spatial redundancy are proposed, one for nodes that only transmit asynchronous traffic and other for the nodes that transmit synchronous traffic. Notice that the scheme to replicate nodes that transmit asynchronous traffic only, could also be adopted in native CAN networks.

5.3.1 Nodes transmitting asynchronous messages only

Replication of nodes that only transmit asynchronous messages is based on a failure detector installed at each replica. It is assumed that the replica priority is lower than the primary ($ID_{replica} = ID_{primary} + 1$) and every asynchronous message has a known maximum inter transmission time (MIT) assigned (thus they are not asynchronous messages in the strict sense). Based on the maximum message inter transmission time the failure detector, located in the replica, could detect a faulty primary slave and replace it. However, transient bus interferences may cause maximum inter transmission time violation that could be interpreted by the failure detector as faulty primary slave. In this case both the primary and the slave replica will compete for message transmission and, if they enter arbitration, the one transmitted by the primary slave will win arbitration given its higher priority. The slave replica will only succeed replacing the primary slave if this one fails by crashing. In this way, the slave replica does not interfere with the network unless it suspects the primary slave has failed by not transmitting a message before its maximum inter transmission timer expires. Figure5.3 presents the automaton of the replica transmitting asynchronous messages only.

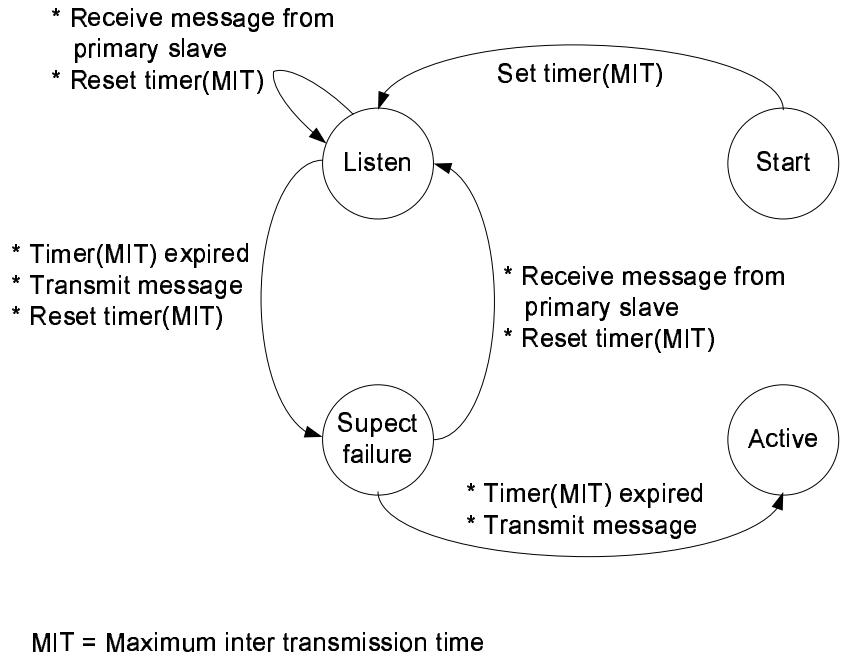


Figure 5.3: Automaton of slave replica that transmits only asynchronous messages.

5.3.2 Nodes transmitting synchronous messages only

The principle adopted in the replication of nodes transmitting only synchronous messages is different from the one adopted for the previous case, since it does not rely on a failure detector. The slave replicas always try to transmit the synchronous messages scheduled by the master. The scheme assumes single shot transmission mode and requires that the priority of the slave replica is lower than any primary node that transmits synchronous messages ($ID_{replica} < \forall ID_{primary}$). According to this scheme, the slave replica only succeeds in transmitting a synchronous message if the primary slave fails to transmit it, either because it has crashed or because the single shot transmission has suffered a transient interference and there is enough room in the synchronous window for that transmission. Notice that the EC-schedule conveyed in the trigger message only accounts for a single transmission of each message during the synchronous window. The computation of the synchronous window duration for a particular EC-schedule is conservative in the sense that it accounts for all possible stuff bits, which usually gives a slack space that is not used. Depending on the size and location of the interference it is possible that both the primary slave and its replica will transmit within the same elementary cycle. This is not a problem because the contents of both messages would be identical and the receiver of the messages could simply discard one of the message instances or rewrite the first one. Thus, this scheme of spatial replication also indirectly implements, in some cases, temporal redundancy as depicted in Figure 5.4-B.

Nodes transmitting both asynchronous and synchronous messages should implement both

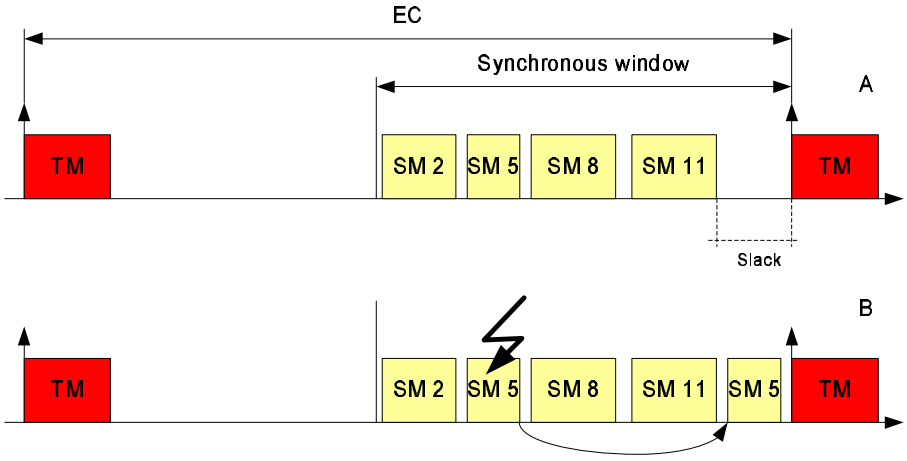


Figure 5.4: Temporal replication of slave node transmitting only synchronous messages.

replication behaviors.

5.4 Temporal redundancy to handle synchronous message omissions

The previous section focused on slave node spatial replication and presented two schemes for implementing such replication in nodes transmitting either synchronous or asynchronous messages. If a transient interference disturbs a slave node transmitting a synchronous message, the replication scheme also provides temporal replication because that message could be transmitted by the slave replica possibly within the same elementary cycle.

This section addresses the cases of non-replicated slave nodes transmitting synchronous messages. In principle these slave nodes are not so critical as the ones that require replication. However it might be desirable to retransmit a faulty message (affected by a transient interference) before its deadline expires.

Prior to discuss the impact of sporadic errors occurring during the FTT-CAN synchronous window, it is important to specify that, in our assumption, sporadic errors do not include transmitter or receiver failures (leading to a bus-off state). Our analysis considers the single error sources only: bit errors, stuff errors, CRC errors, form errors and acknowledgment error. The worst case scenario considering all these possible error sources occurs when an error corrupts the last bit of the End-of-Frame field of a maximum sized message ($133\tau_{bit}$ being τ_{bit} the time required to transmit one bit) and the subsequent error frame has the maximum size ($20\tau_{bit}$). Notice that the maximum message length, for CAN base frame format, is 133 bits (with a maximum of 22 stuff bits)[CiA99]. This single error causes a message loss and consumes a bandwidth equivalent to $153\tau_{bit}$ (inaccessibility time).

As it was discussed in Chapter 2 the interference pattern could be described by an error model, either stochastic or deterministic. The fault model adopted in this work is a deterministic one and it foresees the occurrence of N errors inside each EC phase. However, the remainder of this section will start by considering a single error occurrence per EC. The results will then be extended to the case of N errors. Notice that from the experimental results presented in Chapter 4, considering the occurrence of a single error within each EC phase is quite pessimistic.

Depending on the context where the FTT-CAN based system is used, either soft or hard real-time, different mechanisms could be adopted to confine or to tolerate transmission errors during the synchronous window. These mechanisms could be either passive or active. Passive ones are located at each slave and rely on the addition of extra time to the synchronous windows, according to some fault model, to accommodate possible message retransmission within the same elementary cycle where the error has occurred. Active mechanisms are master node responsibility and rely on the detection of missing synchronous messages that should have been transmitted by slaves. Knowing which messages failed to be transmitted, the master node could try to recover the error by, e.g., re-scheduling missing messages within the deadline, if possible.

5.4.1 Passive mechanisms

The basic idea behind the passive mechanisms to handle synchronous message omissions is to mask the error occurrence extending the synchronous window duration, at scheduling time, according to an error model.

Handling soft real-time synchronous message omissions

Assuming that a synchronous message can be lost in case of error (soft real-time), one may choose to drop the message in which the error has occurred or retransmit this one and drop a lower priority one. This option is related with the synchronous message transmission method adopted. Dropping the message affected by the error implies the use of a special transmission mode: the single shot transmission in case of error and not on arbitration loss, as common. Conversely, if one chooses to drop a lower priority message, FTT-CAN continues operating in the CAN usual way with automatic retransmissions in case of error or arbitration loss, until the end of the synchronous window. At this point the protocol will truncate any pending traffic, thus dropping any non transmitted messages.

These two options are illustrated in Figure5.5 and Figure 5.6, respectively. In this example, a synchronous window with 4 messages SM1, SM2, SM3 and SM4 (SM1 highest priority) is disturbed by an error during the transmission of SM1. The error is detected and, in a worst-case scenario, the bus stays inaccessible for $133\tau_{bit} + 20\tau_{bit}$. This inaccessibility time corresponds to an error occurring in the last bit of the End-of-Frame field of a maximum sized

message and that error causes a maximum sized error frame.

In the case depicted in Figure 5.5 the message is not retransmitted and, since this particular synchronous window was designed to accommodate 4 messages, only SM2, SM3 and SM4 will be transmitted given that transmission of message SM1 has been aborted before completion. In order to accommodate a single error in a maximum sized message, the initial window length has to be extended by $20\tau_{bit}$. The overhead introduced to accommodate one error and the consequent message loss is relatively low.

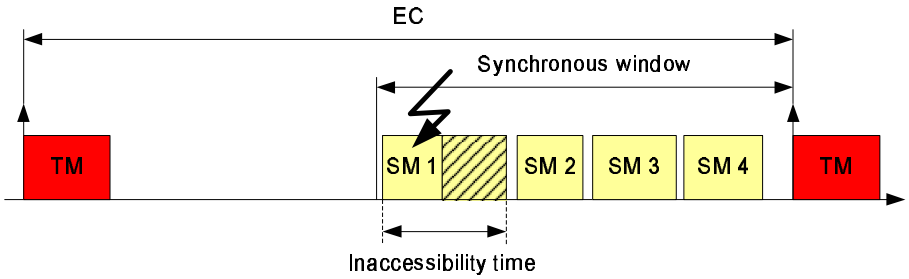


Figure 5.5: Error inside a synchronous window; the message where the error occurs is lost.

In the case depicted in Figure 5.6 the message is retransmitted using the usual CAN automatic retransmission mode in case of error or arbitration loss. Since the synchronous window was designed to accommodate 4 messages, only the messages with higher priority are transmitted (SM1, SM2 and SM3). The node producer of message SM4, after the message SM3 completes, tries to transmit but there is no time left in that window to successfully complete the transmission, so the message is dropped. Again the initial synchronous window length has to be extended by $20\tau_{bit}$.

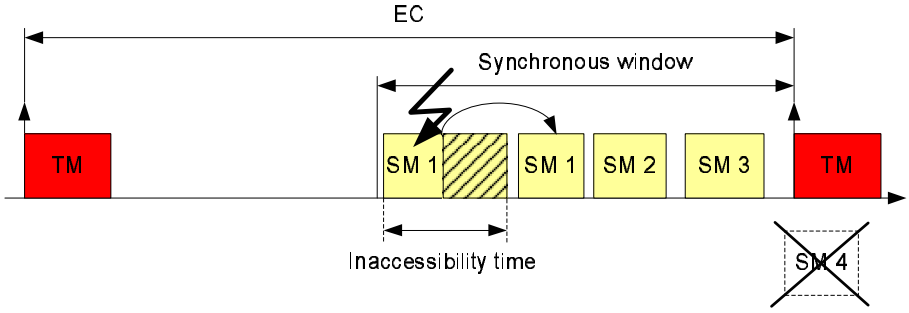


Figure 5.6: Error inside a synchronous window; the message where the error occurs is retransmitted and one with lower priority is lost.

The synchronous message transmission method depicted in Figure 5.6 is easier to implement in existing CAN controllers, than the one depicted in Figure 5.5, since it is based on the standard CAN operation mode, i.e. automatic retransmission in case of error or arbitration

loss. Other consequence of using this method is that high priority messages are the ones with higher probability of being effectively transmitted, because upon transmission error high priority messages will keep winning arbitration.

The synchronous message transmission method illustrated in Figure5.5 requires a special transmission mode in which retransmission is attempted in case of arbitration loss, only, but not on transmission error. This operation mode depends on the particular CAN controller adopted, e.g., NEC μPD789850 DCAN controllers provide direct support for this mode, but the vast majority of the others do not.

Current implementations of FTT-CAN have been using the method depicted in Figure 5.6.

Handling hard real-time synchronous message omissions

An FTT-CAN based system with hard real-time requirements must be able to accommodate a sporadic error occurring in the last bit of a maximum sized message that causes a maximum sized error frame, keeping the network inaccessible for $133\tau_{bit} + 20\tau_{bit} = 153\tau_{bit}$ and still be able to retransmit the affected message before its deadline. Therefore, it is important to assure that there is sufficient slack time available in the bus schedule.

Keeping the same error assumptions as in the previous case, i.e. at most one error every EC, and following the reasoning adopted for the soft real-time case, the FTT-CAN synchronous window has to be extended at most by $153\tau_{bit}$ to accommodate the retransmission of the erroneous message and the error frame. So, the effective EC time available for message scheduling is also reduced by $153\tau_{bit}$.

In the example illustrated in figure 5.7 a message SM2 is distressed by an error that causes an inaccessibility time of $153\tau_{bit}$, afterwards the message is retransmitted successfully.

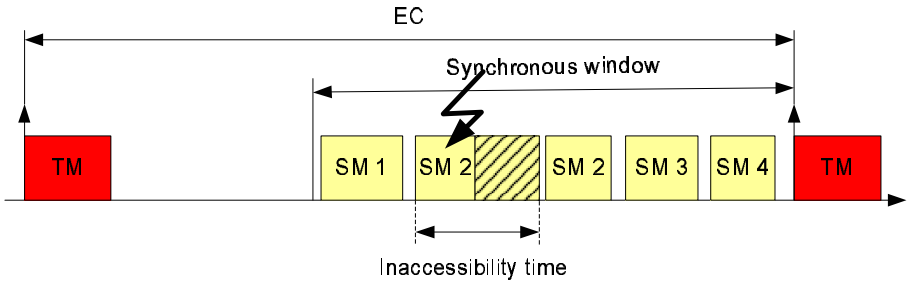


Figure 5.7: Error inside a synchronous window causing message retransmission and no message loss.

In the previous examples, a simplified and rather pessimistic error model limited to one frame error per EC was considered. However, some sources of errors generate bursts with a short inter-occurrence interval, possibly leading to more than one error in each EC. Neverthe-

Synch. window length (ms)	Message omitted impact of 20 τ_{bit} (%)	Message retransmitted impact of 153 τ_{bit} (%)
1	2.00	15.30
2	1.00	7.65
4	0.50	3.83
6	0.33	2.55
8	0.25	1.91
10	0.20	1.53
20	0.10	0.77

Table 5.1: Impact of an error in an FTT-CAN synchronous window (at 1 Mbps), in terms of bandwidth and the mechanism adopted to handle errors.

less, the same reasoning explained before is still valid, but for each individual error occurrence. Thus, for the case of soft real-time communication, if the errors are all part of a single burst (error inter-arrival time is less or equal than $1\tau_{bit}$), the scheduler must consider an extra $N + 20\tau_{bit}$ in each synchronous window for each possible error occurrence (N), or $N \times 20\tau_{bit}$ considering a worst-case scenario when the error inter-arrival time is greater than a maximum sized error frame and each error causes a maximum sized error frame.

For the case of hard real-time communication, the scheduler must consider$153\tau_{bit}$ for each possible error occurrence in synchronous windows, accounting, thus, a total of$N \times 153\tau_{bit}$ in each synchronous window. In this way, the FTT-CAN protocol would support a guaranteed timely behavior in the presence of errors as long as the error frequency is less than or equal to the number of errors considered by the error model. Notice that if the asynchronous traffic also has hard real-time requirements, the same reasoning should be applied, and consequently, the asynchronous window duration should also be dimensioned appropriately, for example using the method proposed by Broster *et al.* [BBRN02].

With a deterministic error model, this approach may lead to a low efficiency scenario in which a significative amount of bandwidth is allocated for error recovery, if too many bit errors are to be tolerated in the synchronous windows. Table 5.1 summarizes the impact of tolerating a single bit error inside each FTT-CAN synchronous window considering window lengths from 1 ms to 20 ms. If one consider the case of a 4 ms synchronous window, tolerating an error without retransmitting the message costs, at most, 0.5% of the window bandwidth, while if the message is to be retransmitted to preserve its real-time properties, the impact on the bandwidth is 3.83%. Notice that these values, although affordable, are too pessimistic because it is considered that an error might occur in every elementary cycle.

5.4.2 Active mechanisms

A different approach to cope with a given error pattern is to react to an error occurrence and to verify online its impact in the system overall timeliness.

The idea consists in monitoring the bus traffic and feedback the gathered data to an online scheduling algorithm able to perform traffic scheduling, so it can take the appropriate measures in case of error occurrence: re-scheduling the affected message(s) for subsequent ECs, allowing message loss if the affected message is not critical, starting emergency operation procedures, etc. In any case, possible retransmissions are controlled by the scheduler and the automatic retransmission upon error is disabled. However, as referred previously, this increases the probability of occurrence of synchronous inconsistent message omissions and, in that case, the master node may detect a missing synchronous message that may have been received by some nodes and, in other cases, will receive a synchronous message that may have not been received by some nodes. A case of special concern is when different masters (active and replicas) detect a synchronous message omission in an inconsistent way. Thus, some will try to re-schedule the missing message while others will perform an error free scheduling. However, this does not raise major concerns since the view of the active master prevails and replicas with different schedules will issue a synchronization request to the active master, as it will be detailed in the next Chapter.

The general technique of scheduling using a fault model to account for temporal redundancy is generically called fault-tolerant scheduling and it has been substantially developed, both concerning the execution of tasks in uniprocessor systems [LC86][BPSW99b] or multiprocessor systems [GMM97][WBDP98].

Given the centralization of the scheduling function in FTT-CAN, many of the existing approaches for uniprocessor systems can be easily adapted, just considering the non-preemption of message transmission and the execution model based on a sequence of separate phases, i.e., the synchronous windows.

However, the additional computing overhead can be too high for low processing power 8-bit microcontrollers, as the ones typically adopted in CAN networks. An easy technique consists in building a vector of flags each synchronous window, indicating the synchronous messages received, using the same format as the EC-scheduling. At the end of the synchronous window and after computing the schedule of the next EC, the master only needs to execute an XOR of the EC-schedule with the flag vector to detect and identify the missing messages. This corresponds to the *verify and update* time slot depicted in Figure 5.8. This hardware dependent time interval needs to be added after the synchronous window. It corresponds to a bus idle time and allows the master to update the EC-schedule before transmitting the next trigger message. If possible, the omitted messages are added to the corresponding EC-schedule, without interfering with the normal scheduling operations.

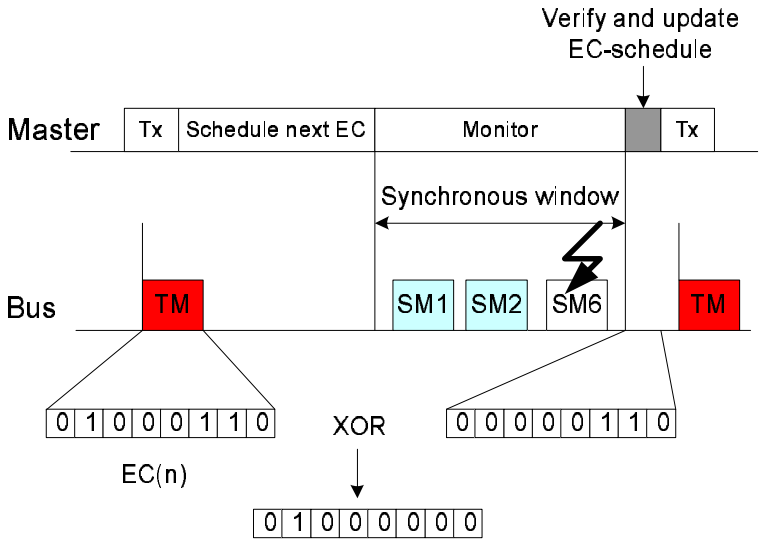


Figure 5.8: Possible fault-tolerant scheduling technique.

5.4.3 Asynchronous message atomicity

It was conjectured, based on the experimental assessment of CAN bit error rate presented in the previous Chapter, that in native CAN the probability of occurrence of inconsistent message omissions is below the 10^{-9} threshold usually accepted for safety-critical applications [Kop97]. However, the experimental results obtained cannot be universally applicable since they largely depend on the considered interference pattern that, in a limit scenario, could corrupt all legitimate bus traffic.

In the case of FTT-CAN, asynchronous messages are transmitted according to CAN standard without any supervision from the master node. Moreover, there are specific scenarios, despite infrequent, in which CAN does not enforce atomic broadcast, i.e. a transmitted message may not be consistently delivered by all the correct nodes of the network and in the same order [CASD85]. Therefore, several solutions to achieve atomic broadcast in CAN have been suggested [LB03][PV03][PMJ00][KL99][RVA+98].

Despite the gaps between the consecutive asynchronous windows, FTT-CAN enforces that message arbitration and retransmissions occur in a logically continuous manner, so that priority inversions do not occur in the transitions. In this way the asynchronous windows of FTT-CAN act as the native CAN bus where the time taken by the trigger message and the synchronous window corresponds to inaccessibility time. Omissions of asynchronous messages are eventually removed by means of retransmissions. For the peculiar scenarios referred previously, solutions in [LB03], [PV03] and [RVA+98] can be used but they are based on asynchronous message confirmation/retransmission and, thus, need certain adaptations to comply with the timing of the FTT-CAN protocol, e.g., to account for the variable inaccessible time

corresponding to the length of the synchronous window and the trigger message. Moreover, the computation of the worst-case response time for achieving consensus using those solutions depends on the message set properties and thus such computation must be done on-line. The overhead imposed in terms of number of transmissions required also makes these protocols relatively bandwidth inefficient. The approach in [KL99] is based in extra hardware and uses continuous retransmission trials in case of network errors, interfering with the timing definitions of FTT-CAN. The MajorCAN protocol [PMJ00] enforces atomic broadcast at the frame level and would solve most of the consistency and synchronization problems related with replica management. However it goes beyond the CAN standard, since it proposes a new format for the CAN error frames that copes with the last but one bit error problem.

For the above reasons, a simplified scheme specifically tailored to FTT-CAN was developed, taking advantage of some unique properties such as the consistent view of the system imposed by the master and the fact that a transmitter always knows if it has successfully transmitted a message. This scheme was developed to enforce consistent updates of the SRTs of active and backup masters and will be discussed further on in Chapter6.

5.5 Conclusion

This Chapter discussed the issues related with transmission errors in FTT-CAN causing message omissions. If an error is transient possibly caused by electromagnetic interference, techniques based in temporal replication (message retransmission) could be used to recover the message affected by that error. If the error cause is permanent, techniques based in spatial replication need to be adopted (node replication). Schemes to replicate slave nodes transmitting either asynchronous or synchronous messages have been proposed and discussed.

The impact of errors in FTT-CAN varies according to the protocol phase where they occur. Errors in the trigger message transmission that lead to its omission may cause temporary network silence. To circumvent this problem the active master node continuously attempts to retransmit the trigger message during the trigger message transmission window. After that, and if it does not succeed, it will transmit the next TM conveying the subsequent EC-schedule when the EC timer expires.

A transmission error during the asynchronous window is handled as in native CAN, i.e., the slave node responsible for the transmission of the faulty message will retry its transmission until it succeeds. If that slave node crashes, its replica will transmit the message when the timer that controls the message's maximum inter transmission time expires.

Two alternatives to synchronous message transmission were discussed, one based in normal CAN transmission mode, with automatic retransmission enabled in case of error or arbitration loss within the synchronous window, and other based in a special transmission mode where the automatic message retransmission is enabled only for the case of arbitration loss.

Transmission errors during the synchronous window could be masked, using a passive

mechanism that adds extra time to the synchronous windows and allows a small number of retransmissions, or recovered from, using an active mechanism capable of detecting missing synchronous messages that should have been transmitted by slaves, and re-scheduling them whenever possible. It has been conjectured that existing fault tolerant scheduling techniques can be used in the FTT-CAN master, with a simple adaptation concerning the non-preemption of message transmission and the transmission within a sequence of separate windows. A similar adaptation has been previously carried out concerning the use of classical preemptive task scheduling analysis [AF01]. The verification of that conjecture is, however, outside the scope of this dissertation.

Chapter 6

Enforcing Master Replica Determinism

6.1 Introduction

As it as described in Chapter 3, the whole FTT-CAN based distributed system is synchronized by the reception of the EC trigger message. When this message is omitted, either due to a master permanent failure or to some temporary glitch, a loss of connectivity happens. That is, the FTT-CAN master node is a single point of failure. This, of course, is no longer true when the master is replicated so that upon failure of the active master, a master replica enters into action within a sufficiently short interval. It is necessary, however, that the masters fail in a silent way and that they are synchronized with respect to the traffic dispatching, i.e., in which EC they must generate the same EC-schedule.

In FTT-CAN, the static traffic dispatching table usually adopted in other master slave architectures is replaced by a dynamic table containing the communication requirements, i.e. properties of the message streams such as period, phasing, transmission time. This table is then scanned on-line by a traffic scheduler. In this case, there is no longer the concept of a cycle count relative to a fixed referential such as the top of a dispatch table.

Thus, other mechanisms must be used to detect synchronization loss between active and backup masters. The idea is having master replicas to monitor both the timing and the contents of the EC trigger messages delivered by the active master. On one hand, if the next trigger message is delayed more than a given tolerance a replica enters into action and transmits the missing EC trigger message. From that moment on, the replica becomes the primary master and the previous primary, if still operational, will become the master replica. Notice that more than one backup master may be used, as long as each one is assigned a different identifier (FTT-CAN allows 8 different IDs for master nodes). On the other hand, if the contents of the EC-schedule conveyed in the trigger message differs from the one computed by the replica, the active master view prevails and the replica SRT becomes unsynchronized.

To enforce consistency and synchronization the replica that assumes the role of primary master must have the same knowledge of the faulty master, i.e., they must be replica determinate. The master nodes are replica determinate if, starting from the same initial conditions (same Synchronous Requirements Table – SRT) and fed by identical inputs (change requests), they produce the same result (EC-schedule) at the same time (every EC). However, since the system requirements, replicated at all masters' SRT are flexible and may evolve over the time, the procedure to synchronize the replicated masters is critical since a communication fault during this process can lead to inconsistency in the replicated SRT images, compromising the replica determinism requirement.

Furthermore, there is also a problem of coherency between the multiple instances of the SRT. There may be change requests that, due to omission communication faults, are taken by the active master but not by one or more replicas, or vice-versa. When this happens, it is important to reestablish coherency and synchronization among all masters as fast as possible.

There are, thus, two situations that can cause inconsistencies among the SRTs located at each master:

1. After an asynchronous startup/restart a master has an outdated SRT. This calls for the definition of a protocol to transfer the SRT from the active master to the unsynchronized one.

2. During the processing of an SRT update request, issued by a node, it must be ensured that all replicas process the same request, in the same order and commit the request synchronously. This calls for the definition of an adequate protocol to enforce consensus preventing, thus, a master to accept a request while others may reject it or to different nodes commit the request at different instants.

Apart from these situations and during standard operation mode, there are no reasons for the replicas to diverge. This is so, because the fault hypothesis considers that master nodes are fail-silent both in the temporal and value domain and that, between two trigger messages, the clock drift among the active master and the replicas is always smaller than the master replacement delay.

This Chapter presents two techniques and protocols proposed to enforce replica determinism among FTT-CAN master replicas, i.e., a solution for the synchronization of starting/restarting masters problem and a protocol to enforce coherent updates of the SRTs in all master replicas [FAF+03][RNPR+04]. In what concerns the synchronization of starting/restarting masters, two techniques are shown. One is for the case in which the masters rely on the planning scheduler [FPAF02], for reduced computing overhead. The other targets systems in which the traffic scheduling is performed every EC, requiring more computational power. In particular it will be considered the case in which masters are equipped with a specialized hardware scheduling co-processor [MFA+02].

6.2 Masters synchronization relying on a planning scheduler

As it was referred in Chapter 3, an on-line scheduler builds the synchronous schedules for each EC, based on the SRT. These schedules are then inserted in the data area of the respective EC trigger message and broadcast with it. Due to the on-line nature of the scheduling function, changes performed in the SRT at run-time will be reflected in the bus traffic within a bounded delay, resulting in a flexible behavior [APF02].

The planning-scheduler [APF99] is a software-based implementation that allows reducing the processing overhead of on-line scheduling. This technique consists on building a static schedule table for a given period of time into the future called plan and rebuilding that table on-line at the end of each plan. The plan duration is not correlated with the messages periods and thus the memory requirements to hold a plan table are bounded and known*a priori*. On the other hand, the plan schedule, once built cannot be changed. Thus, the planning scheduler establishes a compromise between computational overhead and reactivity to change requests an it is particularly well suited to systems with low computational capacity nodes (e.g. based on simple 8-bit microcontrollers).

An important aspect is the synchronization between primary and backup masters in what concerns traffic scheduling. Since the schedulers that run on the masters are dynamic, it must be guaranteed that in each EC they generate similar schedules at the same time. Thus, in every EC all backup masters compare their own schedules with the schedule conveyed in the trigger message, to detect any mismatch. Moreover, they also compare a short cyclic sequence number (3-bit) that is also encoded in the trigger message, to tolerate up to 7 trigger message omissions.

During operation, an asynchronously starting master, or a master restarting in the course of a reset, can cause inconsistencies among the masters. In these cases the view of the active master prevails and thus some masters may need to re-synchronize themselves. Thus, whenever an inconsistency is detected the backup master issues a synchronization request, causing the current primary master to download the SRT as well as the relative phasing information necessary to resume scheduling synchronously.

The synchronization process may take a few ECs depending on the size of the SRT and on the current network utilization. This is a time critical task since during its execution modifications to the SRT are not allowed and the synchronizing master is not yet capable of replacing the primary in case of failure.

Since the synchronization protocol relies in asynchronous messages to transmit the required information, it introduces an overhead that affects the performance of the asynchronous messaging system.

The data that has to be received by a backup master in a synchronization process can be divided in two groups, one containing static properties of messages and other containing scheduling state dependent properties. Example of static properties are the data size, period

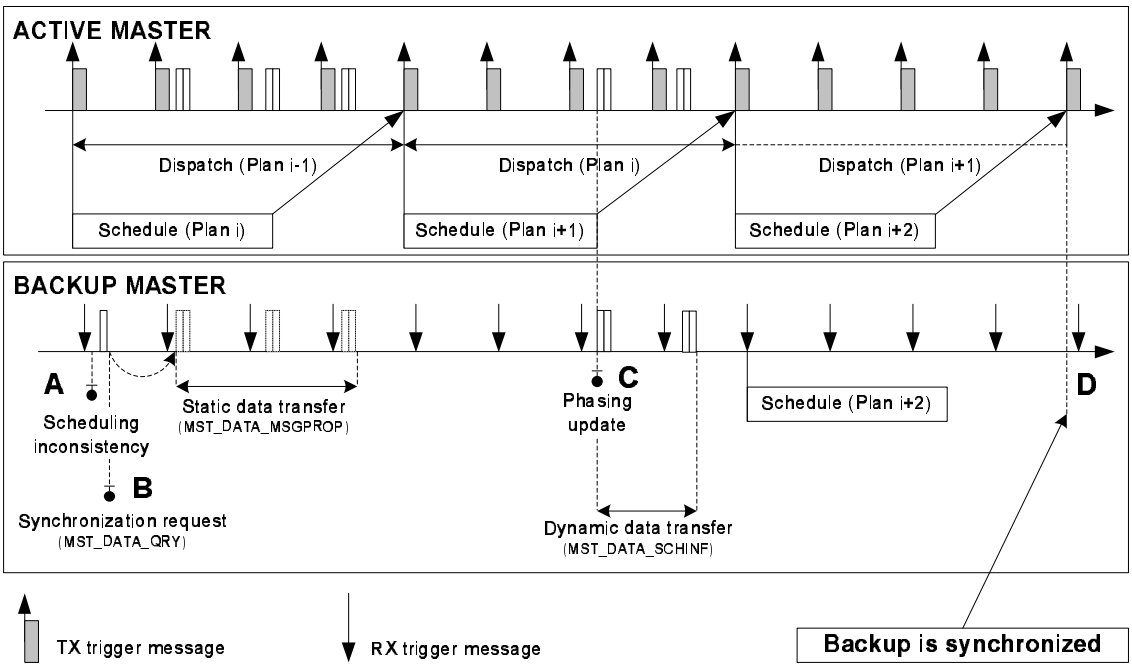


Figure 6.1: Timeline of the scheduling synchronization process.

and deadline and for scheduling state dependent data the instantaneous relative phasing.

The timeline of the synchronization process is depicted in figure 6.1. A schedule inconsistency is detected at A, which causes the backup master to issue a synchronization request at B. Once the active master receives the synchronization request (MST_DATA_QRY), it starts to download the SRT table and the relative phasing data in two rounds starting in the following EC. In the first round, the SRT is fragmented and conveyed by several messages (MST_DATA_MSGPROP). These messages carry only the static properties (e.g. period, deadline, message IDs, etc). Once the first state transfer round is complete, the dynamic scheduling dependent data (e.g. relative phasing) is also fragmented into several messages (MST_DATA_SCHINF) and transmitted. The transmission of this last transfer round must be enclosed within a single plan, *plan i* in the example, and only after the scheduling of the next plan, *plan* $i+1$, is completed (C) in order to assure the consistency of the time dependent scheduling data. Notice that the relative phases of all messages are expressed with respect to the start of each plan. Once this data is fully received by the backup master, it waits for the beginning of the next plan, *plan* $i+1$ in this example, to start the scheduling for the following plan, *plan* $i+2$. The backup master is then ready to monitor the trigger messages produced by the active master and replace it in case of failure as soon as a new plan begins (D). Notice that to facilitate the synchronization of the planning scheduler, the start of a new plan is encoded in the trigger message.

From the master synchronization point of view, the existence of the planning scheduler

is quite convenient since it gives room for state transfer without compromising the normal network activity and also without imposing an excessive load to the masters.

6.2.1 Computing the worst-case scheduler synchronization latency

As it was previously described, the synchronization of a backup master node requires the proper reception of a set of data from the active master. During this process the backup master is unable to replace the current active master since it has not enough information either in the time or value domain, to build equal schedules in parallel. Therefore, to assess the system reliability it is important to compute an upper bound for the time required by the synchronization process.

The number of CAN frames required (N_F) to send either static or scheduling state data depends on the size of the SRT. Particularly, it depends on the number of messages (N_{RT}) and on the amount of data required to represent the respective set of properties for each one (MP_{Len}). Knowing that the maximum number of data bytes that can be carried in each CAN message is 8, equation 6.1 gives the number of CAN data frames and their respective size, needed to transmit static and scheduling state data of the SRT.

$$N_F = \begin{cases} \left\lfloor \frac{(N_{RT} \times MP_{Len})}{8} \right\rfloor_{|DLC=8} + 1_{|DLC=x} & \text{if} \quad x \neq 0 \\ \\ \left\lfloor \frac{(N_{RT} \times MP_{Len})}{8} \right\rfloor_{|DLC=8} & \text{otherwise} \end{cases} \tag{6.1}$$

with

$$x = (N_{RT} \times MP_{Len}) - \left\lfloor \frac{(N_{RT} \times MP_{Len})}{8} \right\rfloor \times 8$$

Besides these data frames, the synchronization process also requires one more data frame:

- MST_DATA_QRY : sent at the beginning of the synchronization process requesting the data from the active master (DLC=0).

The successful end of the transaction is signalled in the identifier of the last message.

The FTT-CAN protocol supports real-time asynchronous messages, with guaranteed response time [PA00]. Given the set of asynchronous messages exchanged in the system, the minimum bandwidth reserved for the asynchronous windows and the relative priority of the asynchronous messages used by the scheduler synchronization protocol, it is possible to obtain an upper bound for the time required to transfer the complete set of messages. However, it is also required that the scheduler state data can be transferred between the end of the planning scheduler execution and the end of the respective plan. This can be enforced using a plan length with adequate duration and establishing a WCET for the scheduler [Alm99].

Such upper bound can be determined as follows (Figure 6.2): at a given point in time a backup master declares itself out of synchronism and issues a synchronization request. This

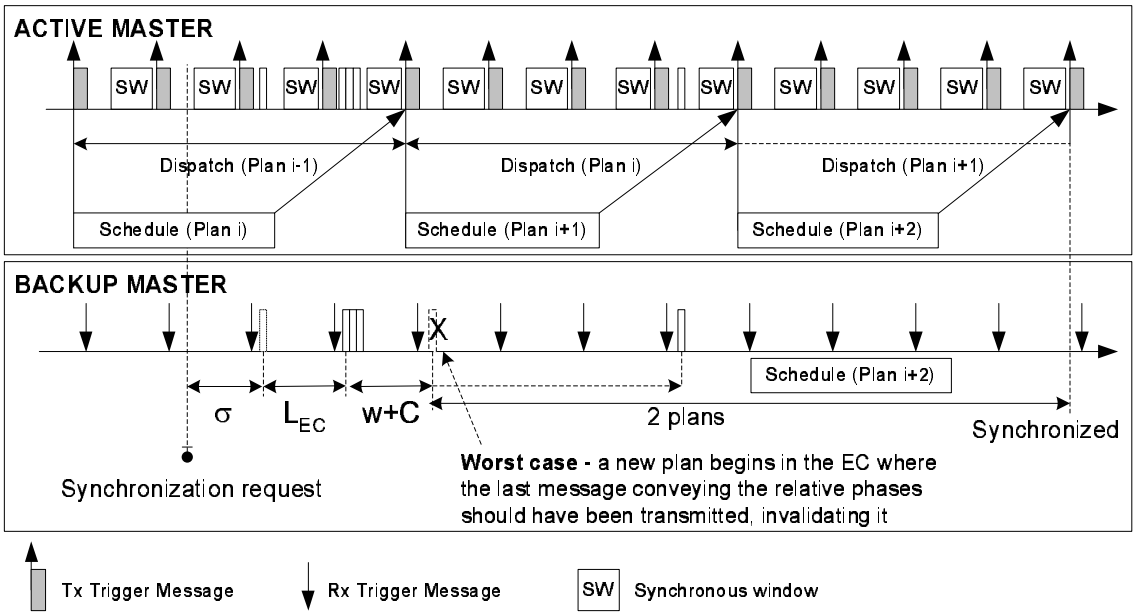


Figure 6.2: Computing the worst case synchronization time for the planning scheduler based scheme.

request is blocked σ by a synchronous window and a trigger message. It is then transmitted in the following EC. The active master takes a snapshot of the SRT and prepares the transfer, which starts in the following EC. If the relative phases are available, the master also prepares them for transfer. Otherwise, it waits for the termination of the current planning scheduler instance. Suppose they are available, thus all messages in MSGPROP and SCHINF are transferred in sequence. The length of the transfer $(w + C)$ can de determined with the method presented in [Alm99], where C is the transmission time of the last message in the whole transaction and w is the interference that this message may suffer from higher priority messages, including the preceding messages in the transaction. However, it can happen that the plan changes in the EC in which the transfer would end, thus invalidating the schedule state data. This causes the backup master to wait for the planning scheduler to finish and for the new relative phases to be transferred. Once this transfer is done the backup master waits for a new plan to trigger its own planning scheduler. At the end of this second plan, the backup master is synchronized. The total time is thus:

$$ST_{WC} = \sigma + L_{EC} + \left\lfloor \frac{w+C}{L_{EC}} \right\rfloor \times L_{EC} + 2 \times L_{Plan} \tag{6.2}$$

where:

σ is the maximum blocking that affects the last message in the synchronization protocol before it enters arbitration.

L_{EC} is a constant value corresponding to the length of the EC.

L_{Plan} is a constant value corresponding to the length of a plan.

Notice that, if it is not necessary to retransmit the scheduling state data the upper bound is reduced by one plan.

6.2.2 Experimental results

To assess the feasibility and correctness of the proposed synchronization process, some experiments were carried out using a 5-node network made of CANivete [FSMF98] boards based on Philips 80592 microcontroller. The EC duration was set to$8.9ms$, the trigger message used 2 data bytes, supporting a maximum of 8 synchronous messages, and the maximum duration of the synchronous window was set to$4.5ms$. The plan duration was 30 ECs and the transmission rate was 125 Kbps. Apart from the synchronous and scheduler synchronization messages, asynchronous ones with up to 8 data bytes were also injected in the bus but with lower priority than those involved in the SRT transfer. The synchronous message set used in this experimental set up is represented in Table 6.1. As referred in previous sections, in this case the static scheduling data consists of the message identifier, data size, period and deadline, while the scheduling state data consists only in the ID plus the relative phasing of the messages at the beginning of the next plan. All these properties are encoded in one byte each.

ID	Period #(ECs)	Deadline #(ECs)	Init. Phase #(ECs)	Size (bytes)
1	1	1	0	1
2	1	1	0	3
2	2	2	0	3
4	3	3	0	2
5	4	4	0	5
6	4	4	0	5

Table 6.1: Synchronous message set properties.

Using equation 6.1, the total number of messages needed by the master synchronization protocol is three 8 byte messages for the static data (ID, P, D, S) and one 8 byte plus one 4 byte messages to send the schedule state data (ID, Ph).

The upper bound for the synchronization time (equation 6.2) of

$$ST_{WC} = 6.2 + 8.9 + \lfloor \frac{11.4}{8.9} \rfloor + 2 \times 30 \times 8.9 \simeq 558ms$$

The experiment was repeated several times in different conditions and, on average, the time to fully synchronize was around $385ms$ (less than one and a half plans). The maximum interval measured in the experiments was ($550ms$), which corresponded to the worst-case situation in which an extra plan was required to repeat the transfer of the scheduling state data. Notice that due to low processing power of the microcontrollers used in the test platform, the use of such a large plan is a requirement.

6.3 Masters synchronization based on a scheduler co-processor

If the scheduling is made in an EC by EC basis, instead of a plan by plan basis, or if the plan duration is not enough, the synchronization solution proposed in the previous section cannot be used, since the active master would not have enough time in one plan to transfer the scheduling state data (relative phasing). Another solution to this problem is to take a *snapshot* of the SRT of the active master and transfer it to the backup master that issued the synchronization request, accounting for the duration of this process (in ECs). After the reception, the backup master holds an outdated image of the SRT and thus must build successive ECs from the *snapshot* instant until reaching synchrony with the active master. This approach requires, however, a higher computing power which sometimes is not available in simple microcontrollers. A possible solution is the use of specialized dedicated hardware, such as the MESSAgE coprocessor [MAF05], capable of scheduling an EC in a small fraction of its duration.

This new solution assumes that all master nodes are equipped with a microcontroller and a MESSAgE coprocessor, as depicted in Figure 6.3.

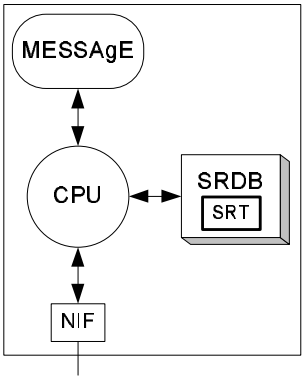


Figure 6.3: FTT-CAN master node architecture, including a scheduling co-processor.

In this case, the coprocessor works as a slave of the node CPU, taking care of message scheduling and schedulability analysis, while the CPU is responsible for dispatching and node management. The coprocessor schedules synchronous traffic on an EC basis, generating EC-

schedules. Between the generations of successive EC-schedules the node CPU can change the message parameters inside the coprocessor, as well as add to or delete messages from the set or update these parameters.

6.3.1 MESSAgE coprocessor

The coprocessor is capable of scheduling up to 32 messages according to one of three selectable criteria: rate monotonic, deadline monotonic or general fixed priority-based. Its functionality is best understood by considering Figure 6.4 which illustrates its programming model, as seen by the node CPU.

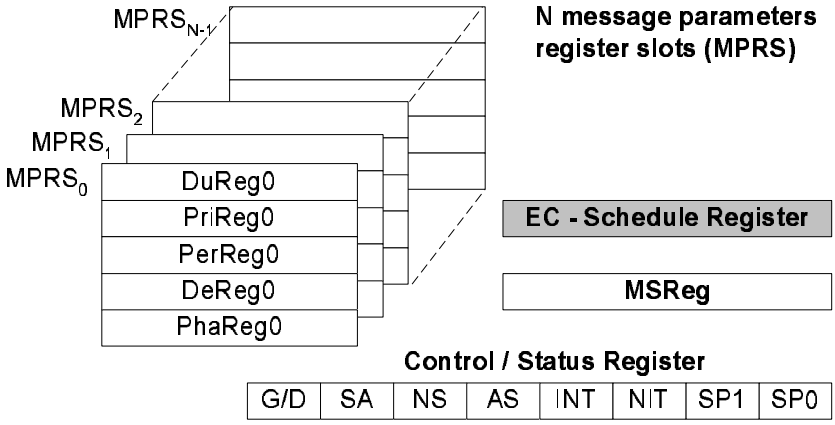


Figure 6.4: MESSAgE programming model.

Each message to be scheduled is assigned to one of the 32 available Message Parameters Register Slots (MPRSs). Each MPRS holds five, 8-bit parameters which characterize the message. The message schedules produced by the coprocessor are made available on the 32-bit EC-Schedule Register (ECSReg), using the same format as the TM. This allows placing the contents of the ECSReg directly in the data field of the TM. The Control and Status Register (CSReg) is used to control the coprocessor in scheduler or analyzer modes, and to check its status.

The Message State Register (MSReg) includes a bit flag for each MPRS, which indicates if the corresponding message has already been release but was not yet scheduled (it is delayed). This information can be read to check the scheduler state, or written to force a particular state. MSReg is used in the master's synchronization protocol, allowing the state of the active master's coprocessor to be copied to the coprocessors in the backup masters.

In its current FPGA-based implementation, MESSAgE runs at a maximum clock rate of 24MHz, which allows it to generate an EC-schedule in at most $25\mu s$. This is about two orders of magnitude less than any real-world EC duration (1-10ms), which leaves virtually all the EC duration available for schedulability tests. As we shall see, this fast scheduling time is also

important in the masters' synchronization protocol.

6.3.2 Synchronization protocol

The synchronization protocol between masters involves, basically, the transmission of the SRT as well as the coprocessor scheduler state from the active to the backup masters. This data transfer is carried out using high-priority asynchronous messages.

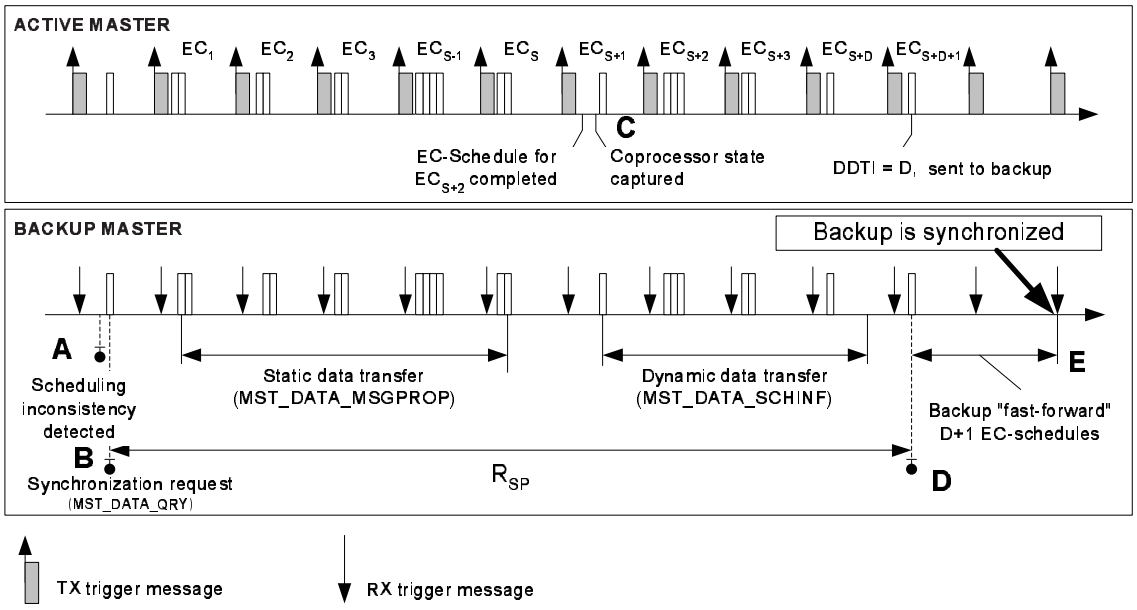


Figure 6.5: Master synchronization protocol timeline.

The timeline of the synchronization process is depicted in Figure 6.5. It begins with a backup master detecting inconsistency on the received EC-schedule (A) and issuing a synchronization request (MST_DATA_QRY message) to the active master (B). This causes the latter to download the scheduling data in two rounds. The first round comprises the transmission of the static portion of the SRT (MST_DATA_MSGPROP). This data is fragmented in a set of CAN messages and transmitted in the asynchronous windows of a number of ECs.

In the second round the active master transmits the scheduling state data (MST_DATA_SCHINF). This comprises the relative phases of all messages and their respective states. Because this transmission may span more than one EC, the data actually sent to the backup master is a snapshot of the phases and message states captured by the node CPU in the coprocessor, just before the start of this round (C).

At the end of this round (D) the active master sends a special message containing the dynamic data transfer interval (DDTI), which is the count of the number of elapsed ECs since the coprocessor state was captured. This number tells the backup master the scheduling delay (in ECs) relative to the active master. Since the coprocessor in the backup master is

now updated with the scheduling data just received, the node CPU just needs to command its coprocessor to generate that number of consecutive EC-schedules (fast-forward). After that, the backup master is synchronized (E). Note in Figure 6.5 that the backup master synchronizes only one EC after the reception of the DDTI message. This is to guarantee that the backup master has enough time (at least one EC long) to generate the required number of EC-schedules specified in the DDTI message even when this message arrives late in the EC. Since we expect DDTIs of just a few ECs (because the dynamic portion of the scheduling data can be packed in a few CAN messages - see next section) and given the high scheduling speed of the coprocessor (0.5% of a 5ms EC), it is reasonable to assume that backup masters can achieve synchronization one EC after they receive the DDTI message.

It is also important to remind that the use of a scheduling coprocessor is not mandatory. This technique can be used as long as the master microprocessor has sufficient computing power. Possibly, it might be necessary to increase the "*fast forwarding*" period to more than one EC.

6.3.3 Worst-case synchronization time

An upper bound for the synchronization time using the a scheduling coprocessor can be determined adopting a reasoning similar to the one followed in section6.2.1. At a given point in time (A) a backup master declares itself out of synchronism and issues a synchronization request (B). This request is blocked σ by a synchronous window and a trigger message. It is then transmitted in the following EC. The active master takes a snapshot of the SRT and prepares the transfer, which starts in the following EC and lasts for ($R_{SP} = w + C$), where w and C have a similar meaning as in 6.2.1. Once this transfer is completed (D) the backup master generates $D+1$ consecutive EC schedules and it becomes synchronized with the active master (E). The total time is thus:

$$ST_{WC} = \sigma + L_{EC} + \left\lceil \frac{w+C}{L_{EC}} \right\rceil \times L_{EC} + L_{EC} = R_{SP} + 2 \times L_{EC} \tag{6.3}$$

where:

R_{SP} is the response time of DDTI message measured from the point where the synchronization request is issued by the backup master.

L_{EC} stands for the EC duration.

Notice that this analysis considers that all messages either from MSGPROP or SCHINF are sent in sequence, without idle time in between. Equation6.3 gives us an absolute upper bound. For the same example considered in the previous method, the worst-case synchronization time ST_{WC} is now according to expression 6.3:

$$ST_{WC} = 6.2 + 8.9 + \lfloor \frac{11.4}{8.9} \rfloor \times 8.9 + 8.9 \simeq 42ms$$

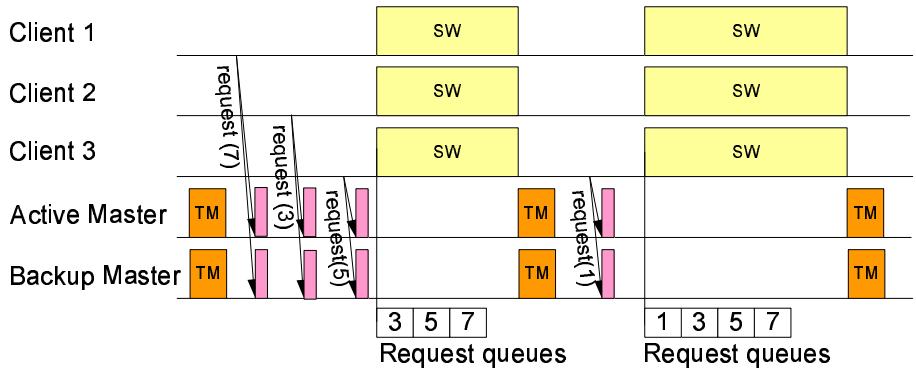


Figure 6.6: Queuing of SRT update requests at each master.

This is about one order of magnitude faster than the synchronization scheme presented in the previous section.

6.4 SRT update protocol

As already explained, FTT-CAN needs replicated masters to avoid the single point of failure that a single master would represent. In such a scheme, every master keeps a local replica of the SRT, and uses this replica in order to produce its output (i.e. the trigger message and the respective EC-schedule). If the local copy in the backup master is inconsistent such master cannot replace the active one upon its failure. Due to this, guaranteing that every master has a consistent replica of the SRT becomes vital for FTT-CAN.

There is only one master operation which may modify the local copy of the SRT, i.e., the processing of an update request. Therefore, as long as each request is consistently processed by all the masters, the consistency of their SRT copies is preserved. Moreover, if more than one request is received then these requests must be processed in the same sequence. However, inconsistent channel errors turn out to be an impairment to this consistency because they may cause different masters to process requests in a different order, as it is discussed next.

Slaves (clients in Figure 6.6) are allowed to send update requests only within the asynchronous using specific high priority control messages. Every master stores the slave's requests in a local queue, which is sorted by priorities (see Figure6.6). In this way, the highest priority request of the local queue is always the first one to be processed. Note that masters choose the highest priority request after the end of the asynchronous window, when no more requests can be received (until the next EC).

Therefore, if the request queues were consistent at that point, then all masters would process the same request and the consistency of the SRT would be preserved. In contrast, if the queues were inconsistent at that point then the requests processed by different masters might differ, leading to an inconsistent update of the SRT. This means that the problem of

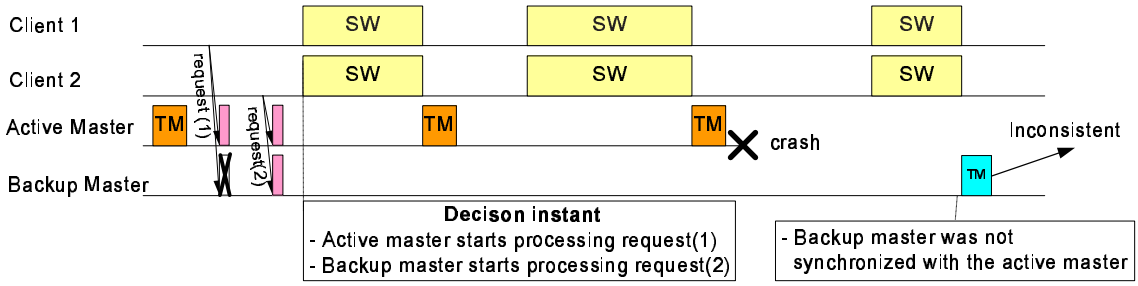


Figure 6.7: An example of unsynchronized masters caused by an inconsistent slave request

guaranteing consistency among the masters could be solved by just enforcing that every master has a consistent request queue at the end of every asynchronous window. Unfortunately, this consistency is not automatic in FTT-CAN due to the so-called last bit error scenario (property CAN.p2 from Chapter 4).

6.4.1 Consistency of the request queues

The error detection and signaling mechanisms of CAN theoretically ensure that any frame is either consistently received or consistently rejected by every node of the network. If this property was actually satisfied in any possible scenarios then the request queues would be always consistent.

Nevertheless, it has been reported [RVA+98] that in certain fault scenarios, a subset of nodes may receive a frame which other nodes reject. In a standard CAN network, the system recovers from this transient inconsistency as soon as the transmitter retransmits the frame, even if the retransmission is delayed by higher priority messages or by further channel errors. But in an FTT-CAN network, the time available for frame retransmission within an asynchronous window is limited. This fact, implies that a transient inconsistency may not be solved by the end of the current asynchronous window, the *decision instant*. This is illustrated in Figure 6.7. In this example, slave 1 and slave 2 send a request within the asynchronous window of the first EC. The request sent by slave 2 is consistently received by both masters whereas the request sent by slave 1 is only received by the backup master. This transient inconsistency makes each master choose a different request for processing. From this point on, the state of the masters is inconsistent. Therefore, in case of active master's failure, the backup master takes over and sends a TM conveying an erroneous EC-schedule.

There are several solutions in the literature that address the problem of data consistency in CAN networks [RVA+98][PV03][KL99] and which were referred in the previous Chapter. It was then stated that, for different reasons, these solutions are not directly applicable to FTT-CAN.

More general agreement protocols have also been presented (e.g. in [DSS98] [Sch90]

[WPS+00]), both in the area of distributed systems and in the area of distributed databases. These protocols are intended for application domains with high computational power, high network bandwidth and efficient operating system support. But if one considers the specific case of fieldbus-based distributed embedded real-time systems with low bandwidth (typically below 1 Mbit/s) and scarce computational resources (8-bit microcontrollers), the performance and timeliness penalties arising from those protocols are too heavy to cope with.

Since none of the mentioned solutions can guarantee consistency among the masters in a satisfactory manner, a new agreement protocol has been designed. This protocol is especially tailored for FTT-CAN as it takes advantage of its particularities in order to reduce the computation and communication overhead of the protocol itself. The details of this protocol are given in the next section.

6.4.2 Protocol Description

In order to eliminate inconsistencies among masters, the active master assumes the role of *leader*, whereas the backup masters assume the role of *followers*. Instead of using specific messages, the active master uses the TM to spread its state. In particular, the TM conveys the identifier of the request which is being processed as well as the state of this processing. The state may be *idle*, if no request is being processed, *admission control*, if one request is in process, or *reply*, if the request processing has finished and the result is being notified to the requester.

The FTT-CAN message ID encoding schema allows for 64 different IDs (6 bits) for SRT update requests, which correspond to the IDs of the synchronous messages the requests refer to. Each request ID is associated with a particular message stream and the requests to update the SRT can be issued by a slave node or by the application running in the master node. In this last case the request must also be broadcast to the network so that the backup master can also process it. Subsequently, the requests queue within each master is sorted by decreasing CAN IDs mainly to enforce the fixed priority philosophy of CAN and facilitate the detection of request duplicates. As referred above, the request ID together with the status of the request process are piggybacked onto the TM, using one byte of its data field. The bit encoding is the following:

- The six least significant bits encode the request ID.
- The two most significant bits are used to indicate the current state of the request process

  00 - idle, no request being processed

  01 - admission control

  10 - reply accept

  11 - reply reject

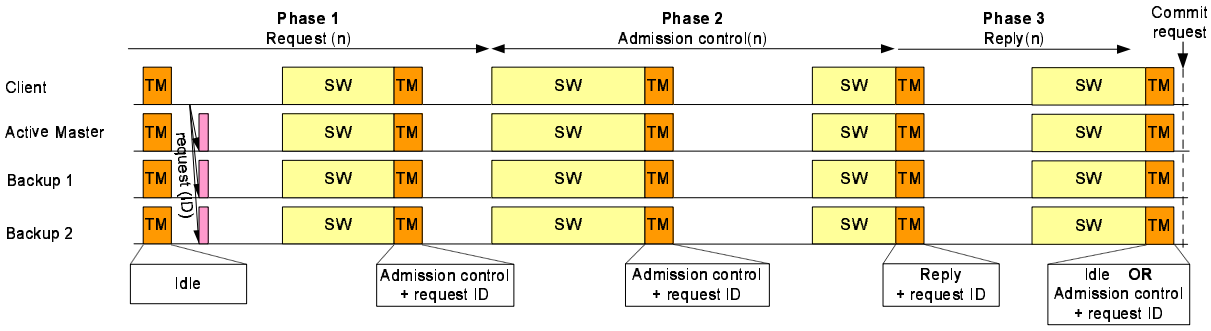


Figure 6.8: Phases of the update protocol.

Whenever a backup master is not able to follow the active master (e.g. because it does not have the request to be processed), it considers itself as being unsynchronized, and initiates the re-synchronization process referred to in sections 6.2 or 6.3.

Figure 6.8 illustrates how the SRT update protocol works in the absence of channel and node faults. The whole process is divided into three phases: **request**, **admission control** and **reply**. In the request phase, the slave broadcasts an update request. Since there is no pending request with higher priority, the active master changes its state to admission control and piggybacks the request ID and the protocol state on the next TM. After receiving this TM all the active and backup masters wait until the end of the asynchronous window and then start the admission control test.

As soon as the admission control test concludes, the active master broadcasts the result. Again, the result of the admission control and the new protocol state are piggybacked on the TM. This phase is called reply phase. At the end of this phase, all nodes (both masters and slaves) know which request has been processed as well as the result of the admission control test. Therefore, each master can update its local replica of the SRT, if necessary.

Once the processing of an update request has concluded and the result of the admission control has been broadcast, the active master is ready to process another request.

The worst case response time to an SRT update request depends on whether there were inconsistencies in the transmission of TMs during the request processing. To determine an upper bound it is necessary to use an appropriate error model that allows estimating the number of extra ECs that each phase may require for consistent TM transmission. Moreover, it is also necessary to establish a minimum inter-transmission time of the requests from the same node, which can be easily enforced by the protocol. With an estimation of such worst-case response time, a timeout can be setup in the requesting nodes, after which they give up waiting for a reply.

Without errors, one single update request takes 4 ECs to be committed but if there are more queued requests to be processed immediately after an initial one, each will take 3 ECs to be committed. If there are N sources of update requests, considering a sufficiently long inter-

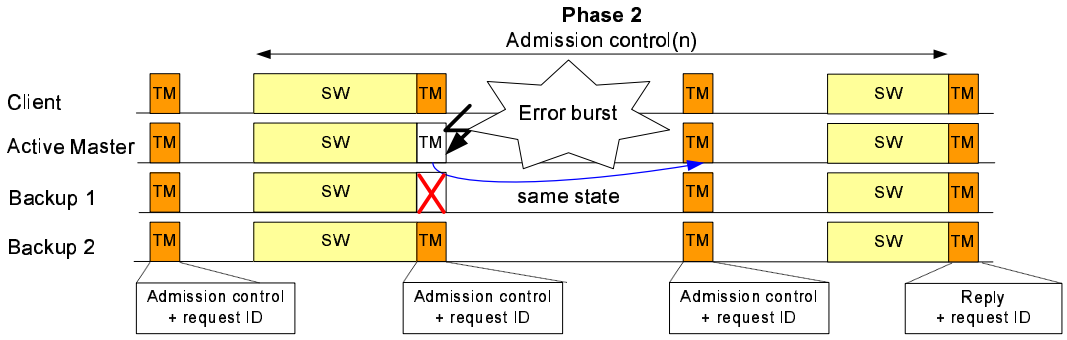


Figure 6.9: Delay in the SRT update caused by a burst of errors.

transmission time for each of them, the maximum number of queued requests is$N$, taking a time equal to $4 + (N - 1) \times 3$ ECs to be committed. However, since the queue is prioritized (N levels, 1 highest), the worst-case time to commit the request with$ID = k$ $(1 \leq k \leq N)$ is given by $4 + 3(k - 1)$ ECs.

6.4.3 Protocol behavior in the presence of channel and node faults

A potential problem of a leader-follower approach, like the one previously described, is that an inconsistent transmission of the TM might cause inconsistency among the backup masters. However, the active master knows whether the transmission of its TM has been inconsistent (property CAN.p1) and can take advantage of this feature. Therefore, upon TM transmission error (possible inconsistency), the active master retransmits the same TM. If, finally, the retransmission is not successful in one EC, e.g., due to an error burst, then in the following EC the master issues a new TM with an updated EC-schedule but maintaining the same protocol state information. In this way, the delayed leader delays all followers (see Figure 6.9).

Therefore, although replicated masters could face transient inconsistencies, such inconsistencies would be eventually solved after a consistent transmission of the TM. Thus, channel errors may enlarge the protocol execution but do not jeopardize consistency.

In case the active master crashes while processing an update request (Figure 6.10), a backup master takes over according to the master replacement mechanism described in the previous Chapter. Note that due to property CAN.p2, a backup master cannot determine if the other nodes have received the last TM and, therefore, it cannot know the state in which the other backup masters are. Due to this, each backup master tries to send a TM which conveys the same protocol information it received in the last TM. In this way, whenever a backup master becomes the active master, it behaves like the previous active master by maintaining the same protocol state information one EC and delaying the remaining backup masters. In addition, a backup master which did not receive the last TM is compelled to be the last one

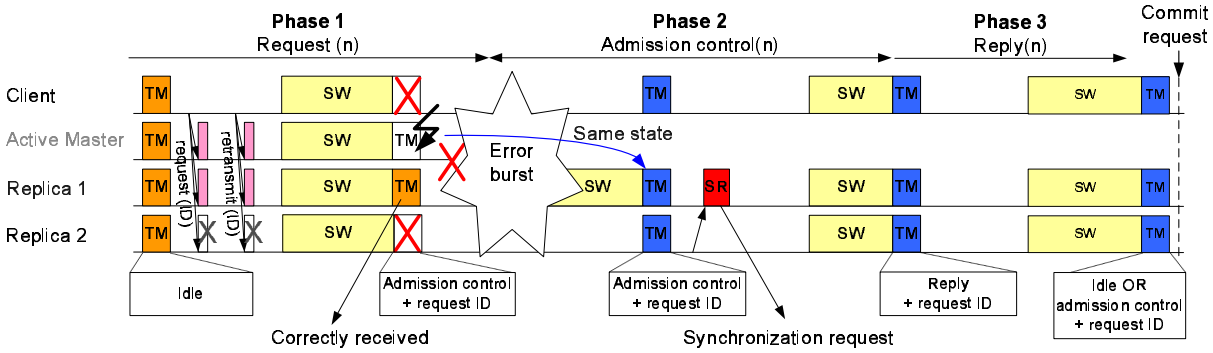


Figure 6.10: Active master crashes while processing an SRT update request. Master replica 2 does not receive the request and must issue a synchronization request. Notice that the error burst does not allow the TM retransmission during the TMTW by a master replica.

to compete in becoming active by using a longer replacement delay. The consequence of this rule is that a delayed backup master cannot become active master unless every other backup master is also delayed.

In the last part of the protocol, the request is committed to the SRT as soon as the state transition from *reply* to *admission control* is detected. This means that the commit is carried out as soon as a TM is received indicating such transition. If there are inconsistent TMs in this part of the protocol, some replicas may commit before others (Figure6.11). This could cause a transitory discrepancy in the EC-schedules of replicas and active master. Therefore, all replicas switch off the policing mechanism, that compares their own EC-schedule with the one conveyed within the TM, between the reply phase, if the request is accepted, and the request commit. This prevents replicas from issuing costly re-synchronization requests, when they are not in fact unsynchronized. Notice that the active master will not start processing other request while the reply from the previous was not successfully transmitted and thus, successfully received by all nodes.

6.4.4 Automatons of the entities involved in the protocol

This section presents the automatons that model the behavior of the slave, the active and the backup master. Notice that these automatons preserve all relevant properties of the protocol, however, for the sake of simplicity and to abstract the fundamental behaviors adopted in the protocol modeling, some non fundamental aspects are not included. AppendixA presents some low level details of the SRT update protocol that are closer to the implementation.

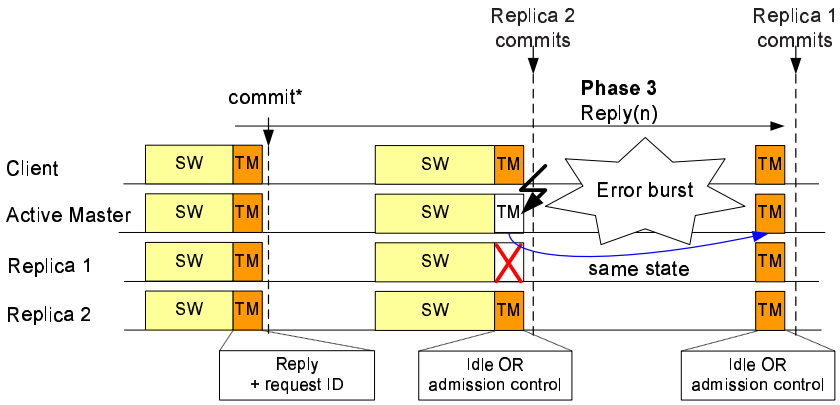


Figure 6.11: Backup masters committing SRT update request at different instants due to inconsistent TM transmission and an error burst.

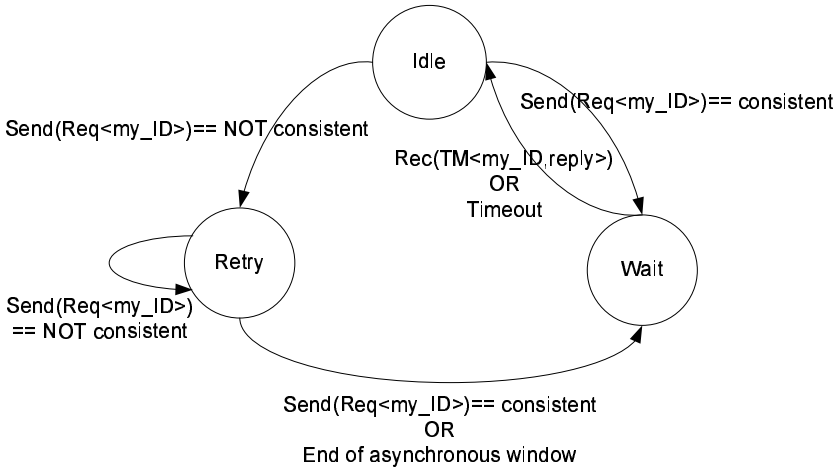


Figure 6.12: Slave's automaton.

Slave automaton

As depicted in Figure 6.12, each slave may be in one of the following states: *idle*, *retry* and *wait*.

During the *idle* state, the slave can request a change of the SRT. This is done by sending an update request within the asynchronous window. Depending on the result of this transmission the slave steps either into the *retry* state, if the transmission was inconsistent, i.e., not successfully accomplished, or to the *wait* state, if the transmission was consistent.

The slave reaches the *retry* state only if the transmission of the update request within the previous EC has been inconsistent. Therefore, and in order to recover from this inconsistency, the slave retries sending the same update request. After successfully transmission or after reaching the end of the following asynchronous window, the slaves steps into the*wait* state.

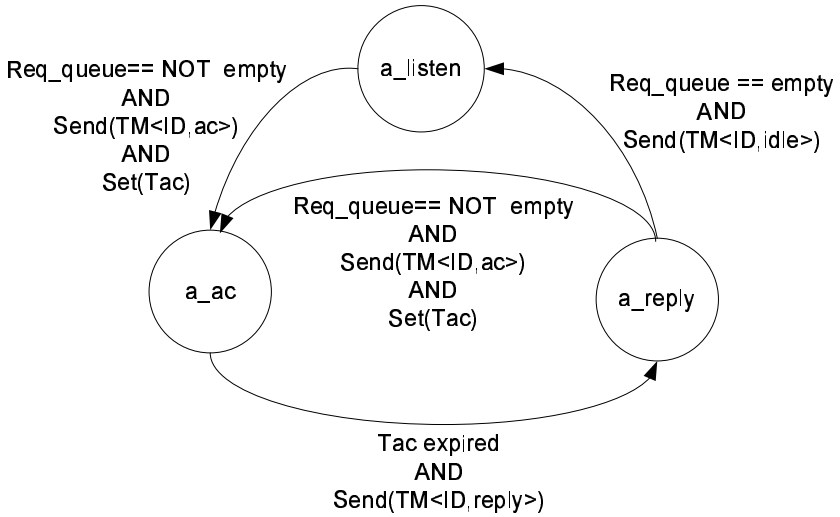


Figure 6.13: Active master's automaton.

In the *wait* state, the slave waits for the reply from the active master. This reply, embedded in the TM, is recognized because it contains the same request identifier as the original update request issued by the slave. As soon as the TM containing the reply is received, the slave leaves this state and goes back to the *idle* state. However, if after a given timeout the reply TM is not received, then the slave considers its request as being rejected and steps back to the *idle* state.

Active master automaton

The automaton executed by the active master (Figure6.13) has three states: *listen*, *admission control* and *reply*. Notice that the active master's transitions are allowed only when the TM is consistently transmitted, i.e., no transmission errors are detected. This is represented in Figure 6.13 with the condition `Send(TM<...>)`. In reality the `Send(TM<...>)` operation is more complex, because it involves a bounded retransmission interval (TMTW) in case of transmission errors, as described in section 5.2.

The active master stays in the *listen* (`a_listen` in Figure 6.13) state as long as an update request is not being processed. Once a request is received, the active master sends a TM which indicates the transition to the *admission control* state (`a_ac` in Figure 6.13) and informs which request is going to be processed. However, the active master itself does not step to the *admission control* state until the TM is successfully transmitted. If this condition is not verified then the active master re-tries to send the TM until it is successfully transmitted. The selection of the request to be processed is non-preemptive: once one request has been chosen, the active master does not change its decision.

In the *admission control* state, the active master processes the selected request and sets the

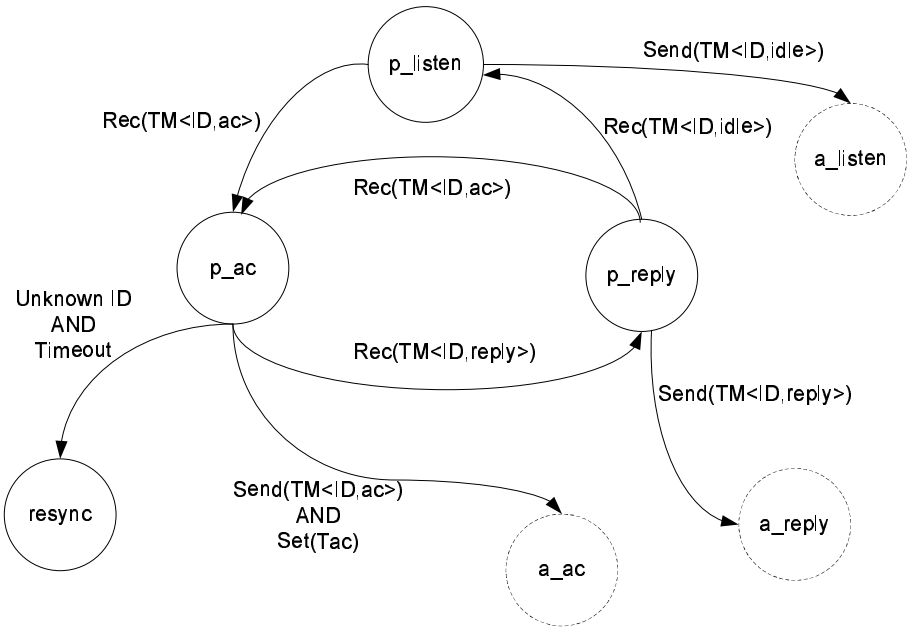


Figure 6.14: Backup master's automaton.

timer `Tac`. This timer indicates the time which the slowest backup master requires to carry out the admission control. Therefore, it ensures that the active master does not step to the next state before any of the backup masters have finished. Once the `Tac` timer expires, the *active* master sends a TM conveying the admission control result as well as the indication of state transition to the reply state (`a_reply` in Figure 6.13). The active master keeps transmitting this message until it is consistently transmitted. When this happens, it moves to the *reply* state.

The active master can leave the *reply* state and move either to the *idle* state, if no more update requests are pending, or to the *admission control* state, if at least one update request is pending. In both cases, the active master moves to the selected state only if the TM, which indicates that transition, is successfully transmitted. If this second condition is not verified then the active master re-tries to send the TM until it is consistently transmitted. If this process extends beyond one EC, i.e., one TMTW, the state of the update protocol is kept.

Backup master automaton

Figure 6.14 shows the possible states of the backup master. Notice that, since any backup master could become the active one upon active master crash, all the states described in the previous section are also included in this state machine. Such states are represented with a dotted circle.

The initial state of the backup master is the *passive listen* state (`p_listen` in Figure 6.14).

During this state, each backup master waits for the reception of a TM indicating the transition to the *passive admission control* state (`p_ac` in Figure 6.14) as well as the identifier of the request that is going to be processed.

Once in the *passive admission control* state, each backup master checks its request queue in order to determine whether the selected request has been received or not. If the selected request has been received then it is processed. Otherwise, the backup master waits for the request during one EC. If after that time the request is not received, then the backup master considers itself as being unsynchronized, and starts the re-synchronization process (`resync` state in Figure 6.14) referred to in sections 6.2 or 6.3.

Two conditions have to be verified in order for a backup master to leave the *passive admission control* state and move to the *passive reply* state (`p_reply` in Figure 6.14). In the first place, the admission control must have been concluded and, in the second place, a TM indicating the result of the admission control must have been received. If the TM is received before the admission control finishes or if the result from the active master does not match its own result, then the backup master considers itself as being unsynchronized.

A backup master leaves the *passive reply* state as soon as a TM which indicates the transition to the next state is received. The next state can be *passive listen* or *passive admission control*, depending on the decision made by the active master.

Whenever a backup master succeeds in transmitting its own TM, it becomes an active master. Note that this is equivalent to jumping from one state of the backup master automaton to the same state of the active master automaton. For the sake of clarity, Figure 6.14 does not include the transitions among the active states (*a_ listen*, *a_ ac* and *a_ reply*) because they correspond exactly to the transitions depicted in Figure 6.13.

6.4.5 Protocol verification

A protocol is basically a set of rules which indicates how entities interact in order to provide a given service. Regardless of the service provided, this set of rules must fulfill two requirements: it must be *complete* and *logically consistent* (i.e. without contradictory rules). Writing a validation model of the protocol in a formal description language, helps guaranteeing that these requirements are fulfilled. The model writing process forces the designer to make explicit assumptions about the environment as well as to specify the protocol rules in an unambiguous way. This process is a key step in the early detection of protocol specification mistakes. In addition, a validation model allows the use of tools which *automatically* verify the logical consistency of the protocol rules and the observance of the correctness requirements. However, to build an effective model, all relevant properties of the protocol should be preserved during the *abstraction process*.

The SRT update protocol has been partially verified by means of *model checking*. According to Clark *et al.* [CGP01], model checking is a formal verification technique by which:

> "*a desired behavioral property of a reactive system is verified over a given system (the model) through exhaustive enumeration of all the states reachable by the system and the behaviors that traverse through them*"

Applying model checking to a design consists of three main tasks. The first task is the modeling of the system in a formalism accepted by a model checking tool. In this work, the system has been modeled in PROMELA (PROtocol MEta LAnguage), with SPIN (Simple PROMELA Interpreter) as the model checker [Hol91][Hol04].

The second task is the specification of the properties that the system must verify. These properties are usually specified in temporal logic, such as Linear Temporal Logic (LTL) [Pnu79].

The third task is the verification of these properties over the model. This task is supposed to be fully automatic, but it actually requires some human intervention. The result of the verification is either 'yes', if the system satisfies the property specified, or a *counterexample* that shows a trace to the state where the property is not verified.

The following sections are devoted to explaining how each one of these tasks has been carried out.

6.4.6 Modeling

The main challenge in model checking is to deal with the state explosion arising either from over-specification or from a poor abstraction process. Due to this, a verification model must be as simple as possible, yet sufficiently powerful to represent all types of coordination problems that can occur in the system. All relevant properties of the protocol must be represented whereas other irrelevant aspects must be abstracted away.

When modeling a distributed system, one has to consider if nodes execute either in a *synchronous* way or in an *asynchronous* way. FTT-CAN nodes execute in an asynchronous way, even though the existence of temporal windows within the Elementary Cycle introduces a sort of synchronism among the nodes. For instance, slaves are not allowed to transmit within the TM window. The transmission of requests within the asynchronous window is interleaved so that slave requests can be received in any order.

Nevertheless, for the purpose of this verification, a synchronous behavior can be assumed for the nodes. This assumption is justified by the fact that even though the slave requests can be received in any possible order, the masters' request queues are sorted by priority. Therefore, the order in which the requests are received within a single asynchronous window is not relevant, and can be abstracted away. Owing to this, this model serializes the transmission of requests.

PROMELA is a C-like language. It allows defining a protocol in terms of three objects: processes, message channels and state variables. In the validation model, FTT-CAN masters and slaves are modeled as processes which exchange information with each other through

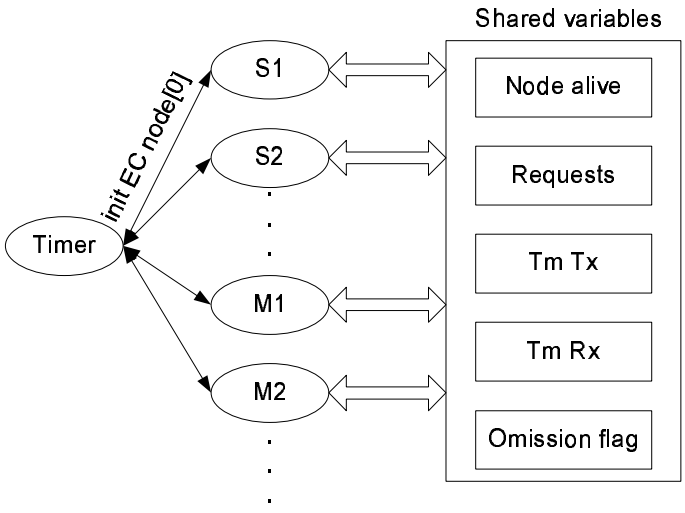


Figure 6.15: PROMELA model scheme.

message channels. It should be stressed that the validation model, presented in this section, is not aimed at verifying the basic services of FTT-CAN but only the SRT update protocol.

A PROMELA process is defined for each node. Moreover, an additional process, called `Timer`, is introduced in the model in order to synchronize the execution of the other processes. In the model, every process is connected to `Timer` through a synchronous channel, as described in Figure 6.15. These synchronous channels, defined in the following vector, are intended to activate and deactivate processes.

```
chan init_EC_node[NNODES]= [0] of {bit};
```

On the other hand, masters and slaves communicate by means of shared variables, instead of using PROMELA channels. In particular, the following variables are used:

```
typedef type_tm {
   byte id_master;
   bit id_req;
   mtype status;
};

bool node_alive[NNODES];
bool requests[NSLAVES_X_NMASTERS];
type_tm Tm_Rx;
bool omission_flag;
```

The function of these shared variable is explained next. Vector `init EC` node can be interpreted as a vector of semaphores, which `Timer` uses in order to guarantee that only one

process is active at a time. The granularity provided by this mechanism is one EC. Moreover, `Timer` follows an activation order which represents the time division of the EC. It first activates one of the slaves; which represents the beginning of the asynchronous window. The activated slave executes one step of its automata (which was described in Section 5) and waits until the next activation. Once the activated slave has finished its computation step,`Timer` activates the next slave, which proceeds like the previous one. The order in which the slaves are activated is not relevant for the verification process, because the request queues of the masters are sorted by priority, and not by reception order.

Therefore, without losing generality, a fixed activation order is used to reduce the complexity of the model. Once the last slave has finished its computation step,`Timer` activates the first master; which represents the beginning of the TM window. The procedure for activating the masters in the TM window is equivalent to the one explained for the slaves in the asynchronous window. Each master executes one computation step as soon as it is activated and then waits until the next activation. Once the activated master has finished its computation, `Timer` activates the next one. After the last master has finished,`Timer` activates the first slave again and the entire cycle is restarted. In contrast to slaves, the activation order of the masters is relevant in order to guarantee some properties which are fundamental for the proper modeling of the system. This is better explained later on, after introduction of the variable `Tm_Rx`.

`Requests` is a vector of booleans of size `#SLAVES`×`#MASTERS`, which is used to model the transmission of the slave requests. Each entry of this vector represents a request from one slave to one master. In order to model inconsistent transmissions of the requests, i.e. properties CAN.p1 and CAN.p2 (see section 4.6.1), a slave process may set the entry of only some masters.

The structured variables `Tm_Rx` and `Tm_Tx` model the mechanisms for transmission of the TM (properties FTT-CAN.p2 and FTT-CAN.p3). `Tm Rx` contains the TM of the current EC, whereas `Tm_Tx` is mainly used to implement the arbitration process within the TM window. At the end of its computation step, the activated master decides the contents of the TM it wish to send. Then, this master checks the values kept in`Tm_Tx`. If the activated master wants to send a TM of higher priority than the TM currently kept in`Tm_Tx` then it overwrites `Tm_Tx` with the values of its own TM. This mechanism ensures that at the end of the TM window, after activation of every master, `Tm_Tx` contains the TM of the master of highest priority in the system (property FTT-CAN.p1). At that precise instant, before activating the first slave and thus starting a new EC, `Timer` copies the values from `Tm_Tx` to `Tm_Rx`.

In order to model the inconsistent transmission of the TM, each process has the choice, in any computation step, of not reading `Tm_Rx`. Whenever this happens, the process which did not read `Tm_Rx` sets the variable `omission_flag` to true. This ensures that the transmitter of the TM knows that the transmission has been inconsistent (i.e. at least one process has not read `Tm_Rx`), as stated in property CAN.p1. However, in order to properly model this

property, `Timer` must guarantee that the master which transmitted the TM in one EC is the last one to be activated in the next EC. So the activation order of the masters needs be dynamic. `Node_alive` is a vector of booleans which indicates whether a node is non-faulty or has crashed. Every entry of this vector is assigned to a unique node process, which is allowed to set the entry to false at the end of any of its computation steps. `Timer` only activates those processes whose node alive entry is true. This models the crash failure semantics of the nodes.

6.4.7 Property specification and verification

In order to assess the correctness of the protocol designed in this work, three properties have to be verified. These properties are discussed next. They have been only verified for a system with four nodes (three masters and one slave). Notice that although only a slave node is included in the verified model, it is unlikely that any practical system will need more than three masters. Verification with a higher number of nodes has not been possible due to the state explosion. Formalizing these properties required definition of a number of additional boolean variables, which are mainly used to indicate internal states of the nodes.

The properties were specified in Linear Temporal Logic (LTL) [Pnu79]. Temporal logic allows one to succinctly describe many interesting temporal properties of systems. Informally, in LTL the symbol `[]` means that the related property is always verified, while the symbol`<>` means that the related property will be eventually verified.

Property 1 (Termination) – If a slave consistently sends a request r then a master eventually processes r.

```
[]( ( r_tx_c ) -> <> (reply[0] || reply[1] || reply[2]) )
```

`r_tx_c` is a boolean which is set by the slave after consistently sending a request and `reply[i]` is a boolean which master `i` sets as soon as it steps to the reply state.

Property 2 (Integrity) – If a master has processed a request r, then some slave must have sent request r.

```
[]( (reply[0] || reply[1] || reply[2]) -> r_tx))
```

r_tx is a boolean which is set by the slave after sending a request.

Property 3 (Agreement) – If a correct master has processed a request r, then all correct masters eventually process request r. The specification of this property requires three expressions like the one below, one for each master.

```
[]( reply[0] -> <> ( (node_alive[1] -> reply[1])
&& (node_alive[2] -> reply[2]) ) )
```

The expressions for master 1 and master 2 are omitted since they are analogous to this one.

6.5 Conclusion

This Chapter addressed the problems that arise from replicating FTT-CAN masters to avoid the single point of failure.

The first problem is synchronizing FTT-CAN masters to ensure that backups remain synchronized with the active master, even if they temporarily loose synchrony, e.g., due to asynchronous start/restart. Two solutions were proposed for this problem: one taking advantage of the planning scheduler capabilities and other based in more powerful computing hardware such as a custom scheduling co-processor.

The second problem is accepting runtime requests to update the communication requirements, while guaranteeing the consistent processing of the requests at all masters. The protocol proposed to handle SRT update requests, was designed taking into account the constrains imposed by the application domain where it is targeted to, i.e., dependable distributed real-time embedded systems. In particular, it takes advantage of some specific properties of CAN and FTT-CAN in order to reduce the protocol complexity as well as the computation and communication overheads.

The proposed protocol is a semi-active one in the sense that all requests are processed in parallel by every replica. However, in order to eliminate inconsistencies between masters, the active master is prioritized to the other masters. The active master rules all the process, assuming the role of the protocol *leader*, while the backup masters assume the role of *followers*. Possible local inconsistencies arising from lack of atomic broadcast protocol, are consistently cleared during the protocol execution by the active master in a *leader-followers* approach.

The computational and bandwidth overheads of this protocol are very low when compared to other solutions, since it only uses one extra message, the request message. The rest of the protocol messages are piggybacked into existing FTT-CAN messages. Also, the computational complexity is independent from the number of replicas, making the protocol scalable.

This protocol was also partially validated using a PROMELA model and the SPIN model checker for the case of a system with four nodes (three masters and one slave). Three basic protocol properties have been verified: termination - consistent delivered requests will be eventually processed by a master; integrity - all processed requests were issued by some slave; agreement - if a correct master has processed a request, then all correct masters eventually process the same request.

The method to synchronize unsynchronized masters together with the SRT update request protocol allow enforcing the required level of replica determinism so that FTT-CAN backup masters can successfully replace the active master without giving away the high level of operational flexibility.

Chapter 7

Enforcement of Fail Silence Behavior in FTT-CAN nodes

7.1 Introduction

A faulty FTT-CAN node that sends unsolicited messages at arbitrary points in time (*babbling idiot* failure mode [Kop97]) without respecting the bus access rules imposed by the master node can disable nodes with legitimate messages to access the network. However, this failure mode can only occur if a node fails in an uncontrolled way. Network topologies that support the operation of fail uncontrolled nodes are costly [Pow91]. Thus a node should only exhibit simple failure modes and ideally it should have just a single failure mode, the fail silent failure mode [Tem98], i.e., it produces correct results or no results at all. In this matter, a node can be fail-silent in the time domain, i.e., transmissions occur at the right instants, only, or in the value domain, i.e., messages contain correct values, only. With fail silence behavior, an error inside a node cannot affect other nodes and thus each node becomes a different fault confinement region [Tem98]. Furthermore, if k failures of a functional unit in a system must be tolerated, then $k+1$ replicas of that unit are needed as long as they are fail silent. If the replicas are fail uncontrolled, then $3k+1$ will be required. Thus, the use of fail silent nodes also reduces the complexity of designing fault-tolerant systems. Usually, fail silence is enforced by bus guardians, which are autonomous devices with respect to the node network controller and host processor, which act as failure mode converters, i.e., the failure modes of the component are, at the interface to other components, replaced by the failure modes of the guardian.

In order to be fail-independent with respect to the interface it monitors the bus guardian must belong to a separate fault confinement region. A guardian would be of no use if it failed whenever the node that it is guarding also failed. Some potential sources of common mode failures are: clocks, CPU/hardware, power supply, protocol implementation, operating system, etc. Designing the bus guardian with independent hardware, with no common components and design diversity can help to avoid common failure modes. Despite the possible

design compromises made between independence, fault coverage and simplicity/cost in any bus guardian architecture it is mandatory for the guardian to have some *a priori* knowledge of the timing behavior of the node it is policing. In time-triggered networks this implies that each bus guardian needs to have its own copy of the schedule and an independent knowledge of the time.

Despite CAN efficient error detection capabilities and automatic fail-silence enforcement, referred in Chapter 4, a CAN node only reaches the bus off state (fail silence) after a relatively long period of time (when the transmission error counter reaches 255). For example, in the case of an erratic transmitter in a 32 node CAN network at 1 Mbps, the worst-case time to bus off is 2.48 ms [RV97]. Moreover, a CAN node running an erroneous application can also compromise most of the legitimate traffic scheduled according to a higher layer protocol implemented in software in a standard CAN controller, simply by accessing the network at arbitrary points in time. Notice that a faulty application running in a node with a CAN controller may transmit at any time without causing any network errors, and consequently unable of leading the CAN controller to the bus-off state, simply by using the highest message priorities. An uncontrolled application transmitting at arbitrary points in time via a non-faulty CAN controller is a much severe situation than a faulty CAN controller also transmitting at arbitrary points in time because, in the first case, a non faulty CAN controller has no means to detect an erroneous application. In the second case the CAN controller would eventually reach the bus-off state.

This Chapter presents two different approaches to enforce fail silent behavior both in the master and in the slave nodes. Fail silence in the slave nodes is enforced using either dynamic bus guardians or internal replication and temporized agreement. The latter mechanism enforce fail silent both in the time and in the value domain and it was designed to be used primarily on the master nodes, given their central role.

7.2 Slave nodes fail silence enforcement

As it was discussed in Chapter 3, there are several possible approaches to enforce fail silence behavior in a distributed system node. Bus guardians are usually adopted in this task because they are simpler than replicating the whole node and using some sort of voting mechanism. Apart from the cost issue, simplicity is also important because a simpler component is often more dependable than a complex one. These were basically the reasons behind the option to design FTT-CAN compliant bus guardians to protect the bus against slave node's unsolicited access. Being FTT-CAN a two-phase protocol with event- and time-triggered parts, an FTT-CAN compliant bus guardian should police both protocol phases.

Concerning the event-triggered phase of the FTT-CAN, it is not possible to exactly specify nor anticipate when a message will be transmitted. This adds extra difficulty in *babbling idiot* detection and hence, in the design of a bus guardian. To be feasible, a bus guardian for the event-triggered phase of the protocol requires a minimum of knowledge concerning the

message parameters produced by each node, e.g., messages IDs and minimum inter message transmission time. Conversely, in the time-triggered phase of the protocol, a node is only allowed to write to the bus during a predefined time window. Therefore it is possible to detect fail-silence violations just by watching when the node writes to the bus. In this way, a bus guardian protecting the bus from a *babbling* node needs to have *a priori* knowledge of the messages the node is suppose to transmit including the transmission instants (traffic schedule). This work is primarily focused on designing a bus guardian capable of policing the time-triggered phase of FTT-CAN. For the event-triggered phase we propose adapting some mechanisms proposed by Broster and Burns [BB03], specifically the minimum message inter transmission time is used to block a node consuming excessive bandwidth.

The peculiarities of CAN do not help or simplify the task of designing a bus guardian for FTT-CAN. CAN protocol works in a bit-by-bit basis and at any instant any node may transmit (e.g. start an error frame) depending of its local view of the bus. That is, the transmission and reception signals are closely related at the bit level and not at the frame level as in other fieldbuses, e.g. TTP/C and FlexRay. Thus the typical technique used in bus guardians for the last two networks, i.e. isolating the respective node from the network outside its allowed transmission/reception windows, is not possible in CAN because, even outside such windows, CAN nodes must answer to every frame with the acknowledge bit and must participate in the re-synchronization process upon errors by transmitting error frames.

Broster and Burns [BB03] proposed a generic bus guardian architecture targeted to event triggered communication protocols. The proposed solution uses an external guardian which is a completely independent node connected directly to the bus. Broster and Burns also showed that a generic bus guardian architecture could be adapted for the specific case of the CAN bus. In this architecture the bus guardian does not police the node directly, instead it monitors the bus traffic to detect *babbling* nodes. In this way this bus guardian is only able to isolate a node that sends correct messages more often than it should and with the correct IDs. A node sending consecutive active error frames, or with a stuck at dominant failure mode or sending correct messages with unknown IDs, could not be located and isolated from the network. With this solution and apart from performance and topological issues, there is no reason to have a bus guardian policing each node since in fact it is not capable of doing that job. A single bus guardian with a direct control line to each node could do the same job.

Broster and Burns proposal assumes a static message set and does not consider online modification of the communication requirements, limiting system's flexibility with respect to online message parameter modifications. In this sense those bus guardians are static and could be programmed at pre-run time or startup with the message parameters. We believe that it is feasible to allow online modifications to the event-triggered communication requirements. This could be accomplished by an entity responsible for online admission control of requests to change the communication requirements and for notifying bus guardians of possible changes in the message parameters. The online admission control is necessary to guarantee that requests

to change a given message property do not violate the overall traffic schedulability. The bus guardian notification, upon accepting a request, is necessary to re-program the bus guardians' filters at runtime.

In our approach the CAN controller of a non-faulty node participates in every network transaction. It is isolated from the network, by the bus guardian, only if it attempts to corrupt the bus schedule. The bus guardians proposed for the slave nodes are initially programmed, at reset time, by the node host processor with the parameters of the set of messages that the node is responsible to transmit, both asynchronous and synchronous. At run time the bus guardian is programmed, every elementary cycle, by the network master, by means of the trigger message. Notice that this programming refers only to the messages to be transmitted during the synchronous window of one EC. Possible modifications of synchronous messages parameters are handled by the master, only. Slave nodes (and their bus guardians) are notified of the modifications by means of the EC-schedule conveyed in the trigger messages. Both the bus guardian and the node CAN controller, connected to the node host processor, receive messages in parallel. However, the node host processor is unable to interfere with the bus guardian except during initialization time.

7.2.1 Bus guardians requirements

Given the peculiarities and restrictions imposed by FTT-CAN, a list of the requirements that should be supported by the bus guardians protecting the bus against*babbling* slave nodes can be enumerated.

1. The bus guardian must protect both the event- and time-triggered phases of FTT-CAN.
2. The bus guardian should enforce temporal isolation among FTT-CAN phases, including the interval for trigger message transmission where all slave's traffic is disabled.
3. Event-triggered traffic within the asynchronous windows should be policed according to message's minimum inter transmission time. Asynchronous messages not listed in the bus guardian's bus access list are not allowed to be transmitted.
4. Time-triggered traffic within the synchronous windows should be monitored to guarantee that a node only transmits messages listed in its bus access list.
5. The bus access list describes the message set the node is allowed to transmit, both synchronous and asynchronous. Each list entry contains the message type (synchronous or asynchronous), identification and size. For asynchronous messages the list also contains the minimum inter transmission time (MIT).
6. The bus guardian only allows transmission of the messages present in its bus access list. This prevents masquerading faults, e.g., a slave node assuming the role of master.

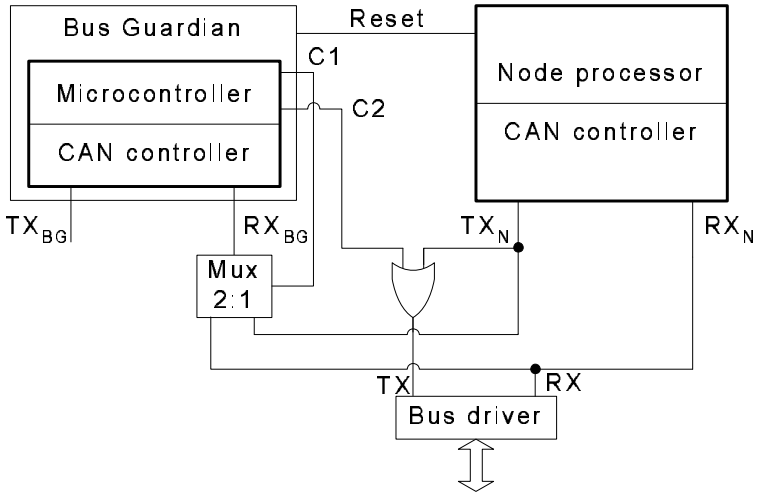


Figure 7.1: Implementing the bus guardian based on an off-the-shelf CAN controller.

7. The bus guardian should detect and isolate a node sending either consecutive active error frames (possibly due to a faulty receiver) or sending consecutive dominant bits (stuck at dominant failure mode).

8. Initial bus configuration (writing the bus access list) could be done by the node host processor, but only during system startup. At run-time the node host processor could not configure the bus guardian.

9. Run-time bus guardian reconfiguration should be possible via the network only. This is necessary to accommodate modifications of message parameters, issued by the network master.

Given these requirements, two design options were considered for implementing the bus guardians. One is based on an off-the-shelf CAN controller and the other one is based on specialized hardware that implements a subset of the CAN protocol, only.

7.2.2 COTS-based bus guardian

The first solution considered the implementation of the FTT-CAN bus guardian using a second standard CAN controller, that supports silent mode operation[1], attached to a dedicated microcontroller. In the silent mode operation the CAN controller does not interfere with the bus, i.e., it does not transmit ACK bits neither active error flags. This ensemble is connected in parallel with the node host processor CAN controller, hereafter also called main controller, and before the bus driver, to monitor and control the operation of the node (Figure7.1).

[1]The Philips SJA1000 provides this operational mode.

Initially, the bus guardian receives the network traffic, i.e. $RX_{BG} = RX_N = RX$, in parallel with the main CAN controller. After receiving and decoding a trigger message the bus guardian enters the policing mode. In this mode, the reception of its CAN controller is switched to the output of the node main CAN controller, i.e. $RX_{BG} = TX_N$, and starts policing the network traffic generated by the node main CAN controller until the end of the synchronous window. After that it goes back to the initial state and the whole process is repeated every cycle.

During the asynchronous window the bus guardian verifies whether the node is transmitting messages more often than it should, verifying message's minimum inter transmission time and size. If it is, the bus guardian blocks its access to the bus. During the synchronous window the bus guardian also monitors the messages transmitted by the node and verifies if they respect both the EC-schedule and declared size (DLC field).

One limitation of this approach comes from the fact that the bus guardian microcontroller only becomes aware of any message after it has been fully received by its CAN controller. Therefore, any potential damage to the system may have already be done and the error detection latency (Figure 7.2) is at least as long as the transmission time of one message plus the time taken by the microcontroller to check the message validity (receive interrupt latency and processing - RX ISR). If the RX ISR duration is shorter than the CAN inter-frame space, i.e. 3 bit times, the node will be isolated before any subsequent transmission can start. Thus, the worst-case fail silence enforcement latency will equal 136 bit times corresponding to 133 bit times for a maximum CAN normal frame format (formerly CAN 2.0A) message plus 3 bit times for the inter-frame space.

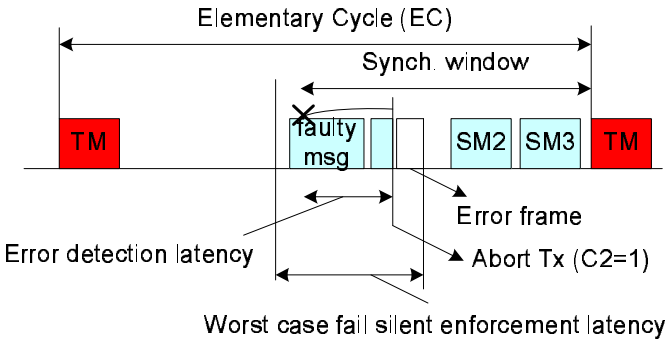


Figure 7.2: Fail silence enforcement using a bus guardian based on a standard CAN controller.

However, this latency can be longer if the RX ISR duration extends beyond the CAN inter-frame space. In this case, the node may start another transmission before the bus guardian takes action and isolates it from the network. Then, the bus guardian aborts the on-going transmission immediately and an error is generated in the network causing interference from error frames between 14 and 20 bit times. The worst-case fail silence enforcement latency

(WCFSL) will thus be 153 bit times (133 + 20) plus the RX ISR duration ($\Delta_{RX_{ISR}}$). The resulting value can be obtained by expression (7.1).

$$WCFSL = \lceil (133 + 20 + \Delta_{RX_{ISR}}) \rceil \times \tau_{bit} \quad (7.1)$$

Table 7.1 presents some worst-case scenarios for several durations of FTT-CAN synchronous windows and for several transmission rates of the CAN bus, considering the RX ISR duration shorter than 3 bit times.

For small sized synchronous windows and low bus transmission rate, a faulty node can corrupt a significant part of, or even all, the synchronous window. However, it only corrupts a single synchronous window since the node is then isolated from the network by the bus guardian. After disabling the bus access, the bus guardian may also cause a hard reset to the node host processor and its respective CAN controller, if programmed to do so at initialization time. This might be helpful for the case of a non-permanent failure, allowing the node to recover autonomously.

Sync. window duration (ms)	125 Kbps	250 Kbps	500 Kbps	1 Mbps
1	0	1	3	6
2	1	3	6	13
4	3	6	13	26
6	4	9	19	39
8	6	13	26	52
10	8	16	32	65
20	16	32	65	130

Table 7.1: Maximum number of maximum sized synchronous messages that fit in a given EC, considering the impact of a node failure and error confinement latency of 1 message time.

A tricky issue in the implementation of this solution is related with the switching of the bus guardian CAN controller input so that it can either *listen* to the bus or *listen* to the node CAN controller output. The bus guardian must always receive the trigger message despite possible trigger message retransmissions during the TMTW. In this way, the signal C1 (Figure 7.1) selects the bus input after the last synchronous message of an EC is transmitted and it selects the node input after receiving the trigger message, only. Notice that the bus guardian must implement a set of timers similar to the ones described in section 5.2.1 to account for possible TM retransmissions during the TMTW and still being able to enforce temporal isolation between protocol phases.

A particular negative aspect of these bus guardians is that when they are receiving data from the bus they are unable to police the nodes and, thus, preventing them to transmit

unsolicited traffic. Furthermore, they are also unable to block the access of a node sending consecutive active error flags or dominant bits (stuck at dominant failure mode). This is so because the bus guardian does not decouple the transmission from the node it policies from the traffic produced remotely and, thus, it is unable to decide whether a particular error source is local or remote. Notice that when C2=0 the traffic received by the bus guardian CAN controller is the result of the logical AND of the traffic locally transmitted by the node with the traffic transmitted by all other network nodes.

Summarizing, this bus guardian does not protect the network against all possible failure modes of the slaves it policies, e.g., the stuck at dominant failure mode. However, in this case, it is expectable that CAN native fault containment mechanisms will eventually drive the faulty node to an error passive or bus off state. Nevertheless, this bus guardian architecture is still capable of enforcing temporal isolation between protocol phases and policing synchronous message transmission, provided the failure modes are less severe.

7.2.3 Specialized hardware-based bus guardian

Another approach to design FTT-CAN bus guardians is to use specialized hardware that implements a subset of the CAN protocol and is able to examine the validity of an ongoing message transmission, right after the transmission of the respective ID and DLC message fields (Figure 7.3). In this approach, since the bus guardian can detect an erroneous message earlier, it can also take action sooner and thus, its worst-case fail silence enforcement latency is shorter.

The operation of the bus guardian is based on its capacity to observe the network state (Rx_{bus}) and the node transmission state (Tx_{node}), independently and bit by bit. In this way, the bus guardian needs to decode the node transmission, that accounts for the node contribution to the overall bus traffic in parallel with the decoding of the bus traffic. In the proposed bus guardian architecture, this is the responsibility of the frame sequencer unit and the control logic unit. The frame sequencer unit provides bit level bus guardian synchronization with the bitstream flowing in the bus and generates the sampling control signal that is used by the control logic to drive two shift registers that store the frames transmitted by the node (Tx_{node}) and the frames flowing in the bus. Notice that when the node does not transmit to the bus, then $\mathrm{Tx}_{node} = recessive$ and Rx_{bus} conveys the bus traffic. Conversely, when the node is transmitting to the bus, then $\mathrm{Tx}_{node} = \mathrm{Rx}_{bus}$ if there are no local errors or $\mathrm{Tx}_{node} \neq \mathrm{Rx}_{bus}$, in the cases when the view of the node differs from the bus state. A mismatch in the bitstream transmitted by the node and the one observed in the bus, depending on its position in the CAN frame, may indicate a possible error in the local node.

Two classes of errors could be detected by this bus guardian: the logical errors related with the FTT-CAN protocol and the errors at CAN bitstream level. Examples of logical protocol errors are unsolicited synchronous message transmission and asynchronous message

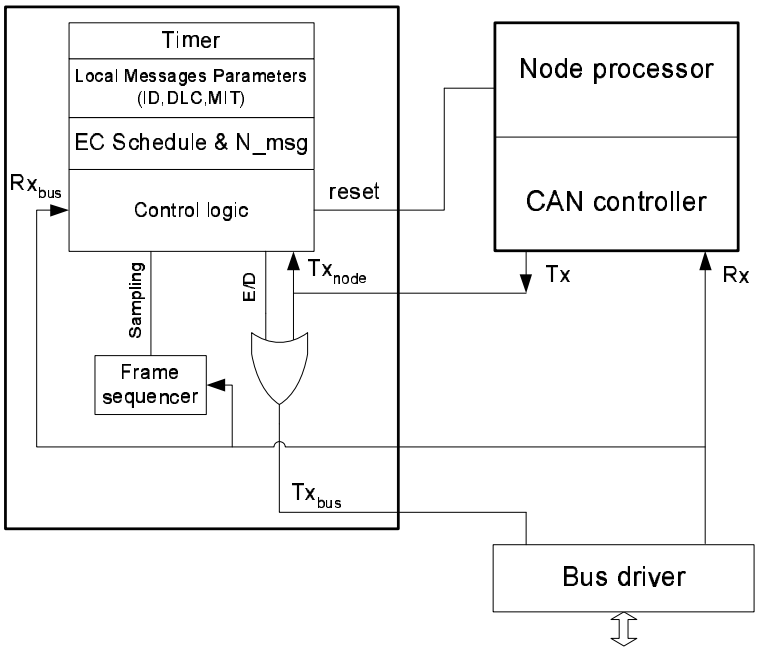


Figure 7.3: Bus guardian architecture based in specialized hardware and its integration in the slave node.

transmissions not respecting the minimum inter transmission time. Examples of bitstream errors include a node constantly sending active error frames or permanent dominant bits (stuck at dominant failure mode). Several mechanisms have already been proposed by Barranco*et al.* [BRNPA04] to handle bitstream errors. In particular Barranco*et al.* generalize the bitstream error concept to account for bit-flipping faults that occur whenever a node exhibits a fail uncontrolled behavior and starts sending erroneous and random bits with no restrictions in the value domain, destroying every correct contribution from non faulty nodes. The proposed bus guardian architecture relies in these mechanisms to detect and isolate $(\!(E/D = 1)\!)$ every node exhibiting this failure mode.

Figure 7.4 depicts the main functional blocks of the bus guardian. The frame sequencer receives the input signal, resulting from the contributions of all nodes including the local node when it is transmitting, and uses it to synchronize the bus guardian with the bitstream flowing in the bus. Thus, the frame sequencer extracts the clocking information from the bitstream and feeds it (`sampling` signal in Figure 7.4) to the bitstream error processing unit and to the shift registers A and B. The bitstream error processing unit makes a bitwise comparison of the most recent bits received in the sift registers to detect any possible mismatch. In case the local node transmits, for example, an active error frame that is not present in the bus, possibly meaning that the local CAN controller is faulty, the bitstream error processing unit isolates the node from the bus. Notice that the bitstream error processing unit can evaluate

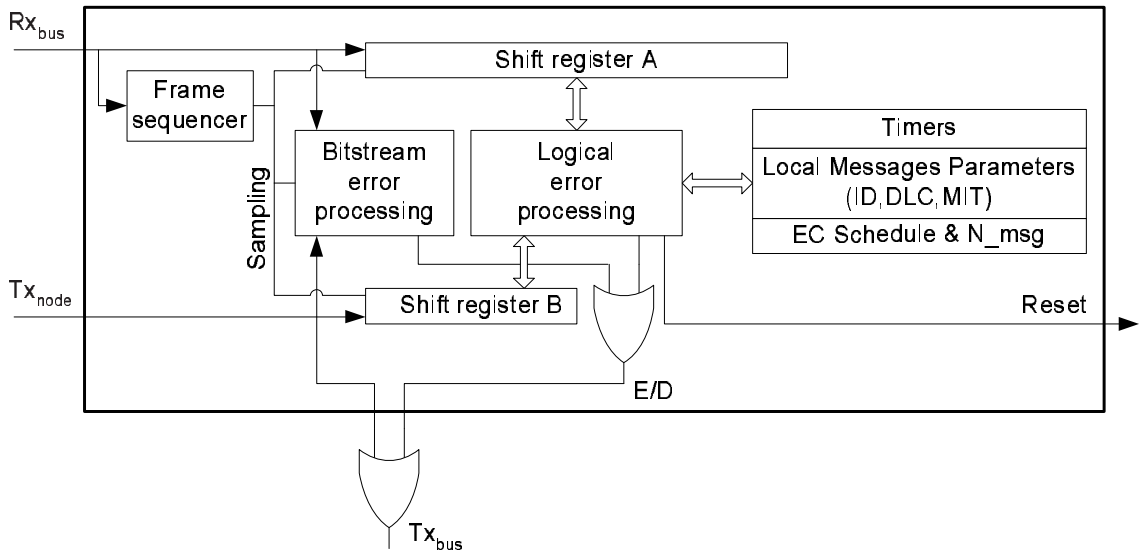


Figure 7.4: Main building blocks of the bus guardian.

the correctness of every transmission issued by the local node, because it receives the same bitstream that is also received by the local node CAN controller.

Shift register B stores the ID and the DLC of each message transmitted by the local CAN controller (15 bits). Shift register A stores the trigger message ID and DATA fields (75 bits). After receiving each TM and storing its content in the shift register A, the logical error processing unit extracts the node schedule for that EC, derives the number of messages to transmit (N_{msg}) in that EC and sets the timer to enforce the temporal isolation between protocol phases.

During the asynchronous window, the logical error processing unit analyzes the ID and DLC of the messages transmitted by the local node (shift register B) and blocks message transmissions that violate the pre-defined minimum inter-transmission time or ID or size. To validate the minimum inter transmission time property of each message, a timer is set each time an asynchronous message is transmitted and reset after the minimum inter transmission time elapses. A transmission is allowed when the timer is reset, only.

During the synchronous window and with the knowledge ofN_{msg}, the bus guardian drives the bus transmission line (Tx_{bus}) according to the following rules:

1. If $N_{msg} > 0$ and the bus is idle than $Tx_{bus} = Tx_{node}$ $(E/D = 0)$. This allows the node to start a message transmission. N_{msg} is decremented for each successful transmission.

2. If $N_{msg} = 0$ and the bus is idle than $Tx_{bus} = recessive$ $(E/D = 1)$, i.e., the node cannot start a message transmission.

3. If a transmission from other node is in progress than $Tx_{bus} = Tx_{node}$, but only to transmit

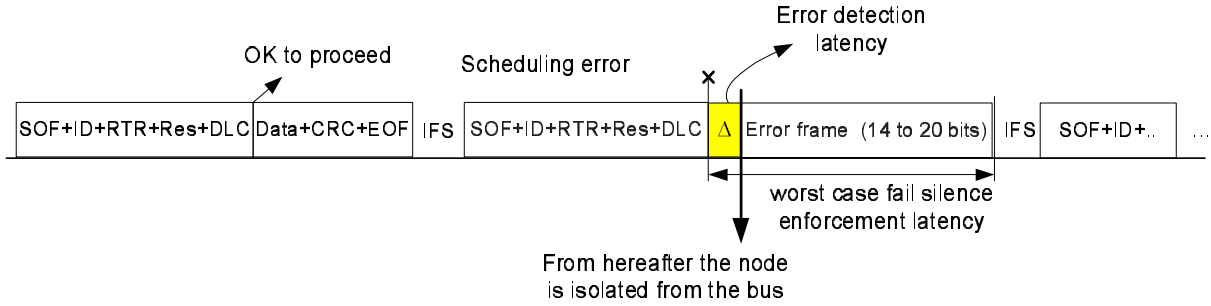


Figure 7.5: Enforcing fail silence with bus guardians based on specialized hardware.

acknowledge bits, error frames or overload frames. At all other times $Tx_{bus} = recessive$ $(E/D = 1)$.

Whenever the node starts a transmission during the synchronous window with $N_{msg} > 0$, the bus guardian policies the transmission bit by bit until the end of the message ID and the DLC fields. If one of these fields is incorrect, the bus guardian immediately aborts the on-going transmission, inducing an error in the bus and isolating the node from the network, possibly generating a hard reset to the host node processor and CAN controller may also be generated.

In this approach, the worst-case fail silence enforcement latency (WCFSL) corresponds to the error detection latency, i.e. 22 bit times (SOF to DLC plus stuff bits) plus the hardware error detection delay ($\Delta_{Hardware}$), plus the time taken by the transmission of error frames induced by the bus guardian when the on-going transmission is aborted, i.e. between 14 and 20 bit times (Figure 7.5). The resulting value can be obtained by expression (7.2).

$$WCFSL = \lceil (22 + 20 + \Delta_{Hardware}) \rceil \times \tau_{bit} \tag{7.2}$$

If $\Delta_{Hardware}$ is of the order of magnitude of a few bit times, then, WCFSL will be close to 52 bit times, which is the transmission time of a minimum-sized CAN frame. Thus, the interference caused by aborting an erroneous transmission is close (less than $10\tau_{bit}$) to the time allocated by the master to a correct message. Since a node issuing an erroneous message must have $N_{msg} > 0$, this means that the interference uses the bus time allocated to that node, with practically no interference on the bus schedule, during the time-triggered phase of FTT-CAN. Notice, also, that the erroneous message is destroyed by the bus guardian, not being delivered.

7.3 Internal replication and temporized agreement

For the case of the master node, and possibly some slave nodes, it is necessary to enforce fail-silence both in the time and value domains. This is mandatory in the master, to guarantee

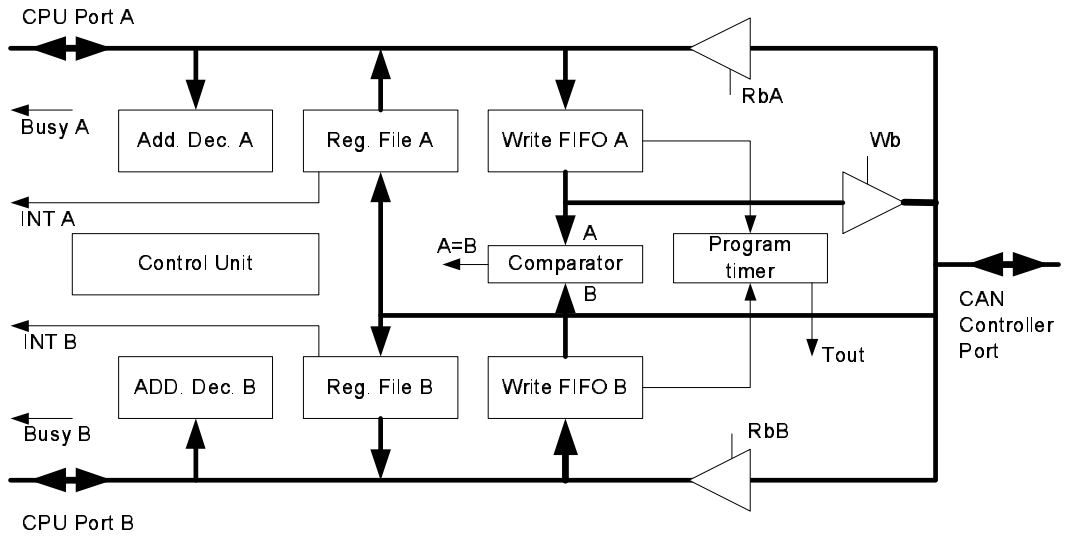


Figure 7.6: Interfacing a pair of processors with a CAN controller in a master node to enforce fail-silent behavior.

the correctness of the EC-schedule that is broadcast to the network. A bus guardian filtering functionality cannot be used in FTT-CAN master nodes because of the causal relations between the master node computed schedule and the bus guardian operation. Fail silence enforcement in FTT-CAN master nodes requires a replicated processing/voting scheme.

Several schemes have been reported over the years to control failure modes of distributed systems nodes. Notable examples of implementations of two processor fail-silent nodes are Stratus [WB91] and Sequoia [Ber88]. In these systems, a reliable common clock source is used for driving a pair of processors which execute in lock-step. Access to the bus is controlled by a reliable comparator circuit which only enables access to the bus if the signals generated by the two processors are the same. Our approach is somewhat similar to this one, but we do not require lock step operation of the processors. Also no restrictions are placed to the processors as long as they produce their outputs correctly and within a narrow time window, thus favoring design diversity.

The proposed approach is based on a specific network interface that supports internal duplication of the node in a transparent way. Basically, this interface enforces an agreement both in the time and in the value domain between all the messages generated by both internal replicas. In case of disagreement, no message is actually sent to the network. Figure7.6 depicts such a network interface called Dual Processor CAN controller Interface (DPCI) which can be used within the FTT-CAN masters. The CPUs run from independent clocks and each of them is connected to a dedicated DPCI port (on the left side) through which it sees the CAN controller programming interface as if it was physically connected to it. Synchronization between DPCI and each CPU is enforced by semi-synchronous port interfaces.

The CAN controller is connected to the right side port. This port is customized according to the interfacing requirements of the specific controller being used. Apart from that, the datapath shown in Figure 7.6 is standard for any CAN controller type. The bus transactions initiated by the CPUs are translated by the DPCI in read or write CAN controller accesses. All CPU writes directed from each port to any controller register, go through a separate Write FIFO memory. These memories decouple the two CPUs, allowing them to run at their own pace. The Comparator unit compares at all times the contents at the head of the two Write FIFOs. A write access to the CAN controller is generated only if a match is detected (correct output in the value domain).

The Programmable Timer (PT) allows this validation to be extended to the time domain as required for example with the generation of FTT-CAN trigger messages. In this case, when a CPU writes the transmission request on its Write FIFO, the PT is started. The other CPU must issue its request before the programmed time-out interval (correct output in the time domain) for the transmission request to be effectively passed to the CAN controller. If a time-out is reached or a mismatch is detected on the Comparator, an external line signals the fault, allowing the node to be disabled, possible generating a hard reset. The temporal validation is selected on a per transaction basis according to hardwired DPCI configuration. Notice that there must be some kind of synchronization among the internal replicas. Otherwise, even a small clock drift would lead both replicas to diverge and eventually isolating the node from the network. Therefore, the required synchronization is achieved upon a write operation, which generates an interrupt upon successful completion. This interrupt is raised simultaneously in both ports, allowing both replicas to synchronize. CPU replicas located in backup masters do not transmit regularly, thus they are synchronized by the reception of the trigger message.

Associated with each CPU port there is also a Register File, which replicates most CAN controller registers. These include the interrupt, status, control and command registers, as well as the receive buffer. All the registers have the same functionalities as their original counterparts within the CAN controller. This replication of the controller registers inside the DPCI is necessary for two reasons. Firstly, because in most controllers some registers change after being read (e.g. interrupt and status registers). Secondly, because this allows the CPUs to run decoupled (for example, when they both read the receive buffer).

The DPCI maintains both register files up to date according to the internal state of the CAN controller. This is achieved by a combination of polling periodically the controller registers and using interrupts to signal changing conditions in the controller. For example when the interrupt line from the CAN controller is asserted indicating a message reception, the control unit copies the contents of the controller interrupt register, status register and read buffer to their respective positions in both Register Files inside the DPCI. Both CPU interrupt lines are then activated and the CPUs now respond to these interrupts as if they were servicing directly the CAN controller.

7.4 Conclusion

This Chapter presented solutions to impose fail silent behavior in FTT-CAN nodes both slaves and masters. Two solutions are addressed, both based on hardware components that are attached to the node's network interface. One solution relies on bus guardians that allow enforcing fail-silence in the time domain, i.e. they inhibit faulty nodes from transmitting messages at arbitrary instants wasting bandwidth and disturbing legitimate traffic. These bus guardians are adapted to support dynamic traffic scheduling and are fit for use in the slave nodes, only. The other solution relies on a special network interface, with duplicated microprocessor interface, that supports internal replication of the node, transparently. In this case, fail-silence can be assured both in the time and value domain since transmissions are carried out only if both internal nodes agree on the transmission instant and message contents. This solution is well adapted for use in the master node but can also be used, if desired, in slave nodes.

Two possible implementations for the bus guardians are presented and discussed, which have different costs and performance in terms of latency to enforce fail silence upon detection of an erroneous transmission and thus, on the interference over legitimate traffic. One is based on COTS components while the other one relies on specialized hardware. The former one, despite easier to implement is substantially more limited in terms of the failure modes supported and in the latency required to enforce fail silence.

Chapter 8

Conclusions and Future Work

The central proposition of the thesis supported by the present dissertation, claims that it is possible to provide a bounded degree of flexibility without compromising dependability. The case-study adopted to validate this claim was the Flexible Time-Triggered CAN protocol: a protocol that combines the predictability of time-triggered systems, favoring the design of fault-tolerance mechanisms, and the flexibility of CAN, favoring the adaptation to evolving conditions.

The FTT-CAN protocol was originally developed to fulfill three basic requirements: timeliness, flexibility and efficiency. This was achieved by combining the advantages of time- and event-triggered paradigms and providing flexibility to the time-triggered traffic. FTT-CAN goes a bit further than other protocols that also combine event- and time-triggered traffic, as TTCAN and FlexRay, by allowing time-triggered messages to be scheduled dynamically and online, in contrast with those protocols where time-triggered messages are static and scheduled at pre-runtime. This fact makes the development of fault-tolerant mechanisms to TTCAN and FlexRay quite straightforward because there is an *a priori* common knowledge of the time-triggered message schedule by all nodes. This is not the case in FTT-CAN. However, as it was demonstrated in this work, it is possible to build fault-tolerant mechanisms for FTT-CAN that preserve the protocol inherent flexibility particularly in the time-triggered traffic.

Allowing flexible communication requirements in a real-time distributed system brings up some concerns regarding safety, since a change in communication requirements can possibly lead to a network overload and consequent timing failures. Furthermore, if the communication requirements can change online and unboundedly, it is not possible to use *a priori* knowledge to distinguish correct transmissions from wrong ones. The use of *a priori* knowledge is of utmost importance in fault-tolerance techniques, to distinguish between what is correct and what is wrong. However, if the on-line requests to change the communication requirements are admissible only within strict boundaries, both temporally and in value, then it becomes possible to guarantee the continued safe and timely behavior of the network. This requires the

filtering of the update requests in order to accept only those that conform to specifications. In this way, the system will still be flexible with respect to the communication requirements although the flexibility is limited to an extent up to which safety is not jeopardized.

The first step towards the definition of a fault tolerant FTT-CAN architecture was the identification of the impairments to dependability of the original FTT-CAN architecture as well as of native CAN. In this context, those impairments have been identified and some new findings, based in experimental data, related with the probability of inconsistent message omissions, were presented. Experimental data indicates that the probability of inconsistent message omissions, which depends on the channel bit error rate, appears to be substantially lower than previously assumed. In fact, in the experiments it is below the10^{-9} occurrences per hour, the commonly accepted threshold for safety-critical applications. This fact enables the direct use of native CAN in safety critical applications without requiring sophisticated and bandwidth inefficient atomic broadcast algorithms. Since the asynchronous messaging system of FTT-CAN preserves all native CAN properties, we conjecture that FTT-CAN asynchronous messages do not require an atomic broadcast algorithm to achieve consensus among FTT-CAN nodes. However, the same experimental results have also shown that the CAN bit error rate is high enough to make FTT-CAN synchronous messages very susceptible to inconsistent message omissions. The level of susceptibility of the asynchronous messages also raises substantially if the retransmission process upon error is interrupted before an unsuccessful transmission.

After having identified the generic impairments to FTT-CAN dependability, the impact of transmission errors in FTT-CAN, causing message omissions, was addressed next. A message omission could occur either due to transient electromagnetic interference or permanent node failure. Techniques based in temporal replication (message retransmission) were proposed to recover messages affected by transient errors, while techniques based in spatial replication (node replication) were proposed to cope with permanent node failure.

Other impairment to FTT-CAN dependability is related with the failure semantics of the master and the slave nodes. The fault hypothesis assumes that nodes exhibit a crash failure semantics, i.e., nodes can only fail by not issuing any message to the network (fail-silence failure mode). Unfortunately this does not match standard CAN and FTT-CAN nodes, thus, mechanisms to enforce such behavior were developed. Two solutions were addressed, both based on hardware components that are attached to the node's network interface. One solution relies on bus guardians that allow enforcing fail-silence in the time domain, i.e. they inhibit faulty nodes from transmitting messages at arbitrary instants wasting bandwidth and disturbing legitimate traffic. These bus guardians support dynamic traffic scheduling and are fit for use in the slave nodes, only. The other solution relies on a special network interface, with duplicated microprocessor interface, that supports internal replication of the node, transparently. In this case, fail-silence can be assured both in the time and value domain since transmissions are carried out only if both internal nodes agree on the transmission instant and

message contents. This solution is suited for use in the master node but can also be used, if desired, in slave nodes.

Two possible implementations for the bus guardians are presented and discussed, which have different costs and performance in terms of latency to enforce fail silence upon detection of an erroneous transmission and in the failure modes supported. One is based on COTS components while the other one relies on specialized hardware (for example, using FPGA technology).

Besides the inconsistent message transmission and the failure semantics of FTT-CAN nodes, another obvious problem with FTT-CAN is the single point of failure nature of the master node. If the master node fails to transmit trigger messages, transmit them out of time or with erroneous contents, then all network activity could be seriously compromised or even disrupted. This impairment was circumvented through master replication, with one or more similar nodes acting as master backups. In this way, as soon as a missing trigger message is detected, a backup master comes into the foreground and transmits it, maintaining the communication. Two problems emerge from replicating FTT-CAN masters to avoid the single point of failure. The first problem is synchronizing FTT-CAN masters to ensure that backups remain synchronized with the active master, even if they temporarily lose synchrony, e.g., due to asynchronous start/restart. Two solutions were proposed for this problem: one taking advantage of the planning scheduler capabilities and other based in more powerful computing hardware such as a custom scheduling co-processor. The second problem of replicating FTT-CAN masters and accepting runtime requests to update the communication requirements, is guaranteeing the consistent processing of the requests at all masters. The protocol proposed to handle SRT update requests, was designed taking into account the constrains imposed by the application domain where it is targeted to, i.e., dependable distributed real-time embedded systems. In particular, it takes advantage of some specific properties of CAN and FTT-CAN in order to reduce the protocol complexity as well as the computation and communication overheads.

8.1 Thesis validation

The thesis stated in Chapter 1, arguing that it is possible to provide a high degree of operational flexibility in distributed embedded systems without compromising their dependability, was validated throughout this dissertation for the specific case of FTT-CAN. In fact, it has been shown, mainly with the work presented in Chapters 5, 6 and 7, that it is possible to build flexible fault-tolerant mechanisms capable of adapting online to evolving requirements.

Several mechanisms and protocols proposed to enforce a correct and consistent behavior despite possible errors and evolving operational scenarios and requirements, were specified, analyzed and validated. The validation was done via of the implementation of the protocol to synchronize masters upon asynchronous start/restart and the master replacement scheme. The

protocol to enforce consistent updates of the SRT was partially validated via model checking, while the components to enforce fail-silent behavior in FTT-CAN nodes were functionally specified, designed and analyzed.

8.2 Future research

Adapting the fault-tolerant architecture to FTT-Ethernet

The FTT paradigm, presented by Pedreiras [Ped03], was implemented also over Ethernet, leading to the FTT-Ethernet protocol [PAG02]. However, FTT-Ethernet fails to provide the level of dependability required for use it safety-critical applications. Concerning the architectural aspect, FTT-Ethernet impairments to dependability are similar to the ones initially exhibited by FTT-CAN, e.g., master single point of failure and fail-uncontrolled behavior of the nodes, both master and slaves. This requires a master replication scheme and components to enforce fail silence behavior of FTT-Ethernet nodes. These components could be physically located within the Ethernet switches in a star topology favoring, thus, failure confinement.

Concerning the impairments to FTT-Ethernet dependability arising from the Ethernet message transmission nature, e.g., the necessity of an explicit acknowledge from all receiver nodes to validate a transmission, the mechanisms proposed for FTT-CAN are not directly applicable to FTT-Ethernet. There are many proposed solutions for this problem of reliable delivery in Ethernet, but it is necessary to assess their impact on FTT-Ethernet timeliness and complexity.

The work of converting FTT-Ethernet into a dependable architecture opens many application domains in the area of real-time adaptive distributed systems.

It is interesting to say that preliminary work on the field of fault-tolerant scheduling has already been carried out in FTT-Ethernet, following the proposals presented in Chapter 5. Namely, a mechanism to detect slave synchronous omissions and to reschedule them has already been developed [AF04].

A holistic approach to dependable adaptive real-time distributed systems

Most of the work presented in this dissertation deals with the communication sub-system only. However, tasks running in a distributed real-time system also need to be taken into account to make that system adaptable to evolving scenarios in a dependable way.

Recent work by Calha and Fonseca [CF04] explores the issue of centralized holistic scheduling of messages and tasks according to the FTT paradigm. This work could be extended to allow the dynamic adaptation to distributed real-time systems by allowing task migration between nodes and reconfiguring the corresponding message scheduling. The FTT paradigm allows a centralized management of a distributed system by keeping the system requirements in a monolithic data structure and disseminating commands to the whole system by means of

trigger messages.

The main challenge of this work is to provide additional services required to secure the centralized on-line management of the whole distributed system. These services include a membership service, to maintain information about which nodes are functioning and which have failed, a clock synchronization protocol, to provide an absolute global time reference, and a reliable, possibly certifiable, resource management service, to allocate and deallocate system resources according to evolving operational requirements. Thus, provided the centralized on-line management is correct and both the communication infrastructure and the operating system running in every node are timely and reliable, it will be possible to adapt to evolving operational scenarios, keeping the system within safety boundaries, without having to analyze all possible operational modes off-line.

We conjecture that the implementation of a membership service is facilitated, in the context of FTT, by the centralized nature of some data structures. The master node could detect a slave node that failed to transmit a synchronous message, thus suspecting it entered the fail silent failure mode. Conversely, a slave node transmitting asynchronous messages only, could send heartbeat messages to report its state. These solutions need to be further investigated according to the scenarios described in Chapter 4.

The resource management service is the cornerstone of the FTT architecture and needs to be formally validated since it is responsible for the global bandwidth management and process activation.

Inserting the master functionalities within a hub or a switch

In distributed embedded systems based in bus topologies, the communication medium is a single point of failure. In applications with safety-critical requirements, the bus is usually replicated to circumvent this impairment to dependability. However, bus replication alone may not prevent the occurrence of spatial proximity faults when node replicas are located close to each other as, for example, when an accident damages the left front wheel of a car or a small fire affects an airplane.

Star topologies are adopted when node replicas cannot be physically separated. Besides favoring the resilience to spatial proximity faults, the use of star topologies also favors the centralized management and isolation of faulty nodes. This approach has been followed both in TTP/C and in FlexRay, where the bus guardian functionalities are integrated within the star. Thus, despite exhibiting a distributed nature with a TDMA medium access protocol (in the case of FlexRay the TDMA scheme refers to the time-triggered phase of the protocol, only), these protocols are topologically centralized.

The FTT architecture is based in broadcast message transmission and is logically centralized with the master node controlling the medium access during the synchronous windows. Thus, the idea of inserting master functionality in a centralized component, either a hub or a switch, is a natural consequence resulting from the logically centralized nature of the medium

access control strategy. In this way, and besides being responsible for admission control and message scheduling, a hub/switch version of the master node could also monitor the contribution of each node to the overall network traffic using the active hub concept proposed by Barranco *et al.* [BRNPA04].

Analyzing the applicability of existing approaches to fault tolerant scheduling for use in FTT-CAN

As it was referred in Chapter 5, fault tolerant scheduling techniques have been substantially developed, both concerning the execution of tasks in uniprocessor systems or multiprocessor systems. Other possible taxonomy divides these techniques in two categories, the static off-line fault tolerant scheduling and the on-line fault tolerant scheduling. Both techniques could be adapted to FTT-CAN, however, FTT-CAN architecture is better suited for on-line fault-tolerant scheduling.

A first step in this direction was already made, as reported in Chapter 5. However several fault tolerant scheduling techniques need to be analyzed and evaluated in terms of complexity versus performance. After electing a fault tolerant scheduling algorithm, which by itself is not a trivial task, the next step would be the joint fault tolerant scheduling of messages and tasks.

Generalization to other protocols of the mechanisms proposed in this work

Despite being specific to FTT-CAN, the components and mechanisms proposed in this work seem to be adaptable to other master-slave protocols, such as WorldFIP [IEC00] (with dynamic bus arbitrator table) or Foundation Fieldbus-H1 [IEC00].

The bus guardian architectures designed for FTT-CAN slave nodes could be adapted to master-slave protocols or to event-triggered protocols. In the first case the bus guardian would police the node in order to prevent it to transmit any unsolicited message and in the second case the bus guardian would prevent the node to transmit consecutive messages before elapsing the minimum inter transmission time.

The dual port controller interface concept used to enforce fail silence behavior both in time and in the value domain within the FTT-CAN masters, is already a generic component that can be equally applied to other protocols, not necessarily master-slave.

Bibliography

[AAR01] J.-C. Laprie A. Avizienis and B. Randell. Fundamental concepts of dependability. Technical Report 739, University of Newcastle upon Tyne, School of Computing Science, 2001. {18,19}

[ADGFT04] Marcos Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Consensus with byzantine failures and little system synchrony. In Rachid Guerraoui, editor, *Distributed algorithms*, volume 3274/2004 of *Lecture Nodes in Computer Science*, pages –, Oct 2004. {25}

[AF01] Luís Almeida and José Alberto Fonseca. Analysis of a simple model for non-preemptive blocking-free scheduling. *Proceedings of the* 13^{th} *Euromicro Conference on Real-Time Systems (ECRTS 2001), 13-15 June 2001, Delft, The Netherlands*, 2001. {124}

[AF04] Jorge Andrade and Nuno Ferreira. FTT em Redes não Fiávies. Technical report, Projecto final de curso, Departamento de Electrónica e Telecomunicações da Universidade de Aveiro, 2004. {168}

[AFF98] L. Almeida, J. A. Fonseca, and P. Fonseca. Flexible time-triggered communication on a controller area network. *Proceedings of Work-In-Progress Session of RTSS'98 (*19^{th} *IEEE Real-Time Systems Symposium)*, 1998. {56,83,101,102}

[AG97] N. C. Audsley and A. Grigg. Timing analysis of the ARINC 629 databus for real-time applications. *Microprocessors and Microsystems*, 21:55–61, 1997. {71,72}

[Alm99] L. Almeida. *Flexibility and Timeliness in Fieldbus-based Real-time Systems* PhD thesis, University of Aveiro, Portugal, Nov 1999. {18,129,130}

[Alm04] Luís Almeida. Real-time networks (for distributed embedded systems). Handouts of the ARTIST Summer School on Real-tme Scheduling and Resource Management, Piazza Armerina, Sicily, Italy, July 5th - 9th, 2004. {v,13}

[AO03] N. Arqueiro and A. Oliveira. Design, Implementation and Test of an FPGA based CAN Controller. *Technical Report, Universidade de Aveiro/IEETA*, March 2003. {88}

[APF99] L. Almeida, R. Pasadas, and J. A. Fonseca. Using a planning scheduler to improve flexibility in real-time fieldbus networks. *Control Engineering Practice*, 7:101–108, 1999. {59,127}

[APF02] L. Almeida, P. Pedreiras, and J. A. Fonseca. The FTT-CAN Protocol: Why and How. *IEEE Transactions on Industrial Electronics*, 49(6), 2002. {5,14,101,102,127}

[ARI90] ARINC. *ARINC Specification 629 Multi-transmitter data bus* Aeronautical Radio INC, 2551 Riva Road Annapolis, Maryland, 1990. {16,71}

[ARI91] ARINC. Arinc specification 651: Design guidance for integrated modular avionics. Standard 651, Aeronautical Radio, Inc (ARINC)– Airlines Electronic Engineering Committee, 1991. {11}

[ASJS96] Tarek Abdelzaher, Anees Shaikh, Farnam Jahanian, and Kang Shin. RTCAST: Lightweight multicast for real-time process groups. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 250–259, 1996. {37}

[Asp03] James Aspnes. Randomized protocols for asynchronous consensus. *Distrib. Comput.*, 16(2-3):165–175, 2003. {23}

[BA04] I. Bate and N. Audsley. Flexible design of complex high-integrity systems using trade offs. In *8th IEEE International Symposium on High Assurance Systems Engineering*, pages 22–31, 2004. {45}

[BB03] I. Broster and A. Burns. An analysable bus-guardian for event-triggered communication. In *Proceedings of the 24th Real-time Systems Symposium*, pages 410–419, Cancun, Mexico, Dec 2003. IEEE. {52,153}

[BBRN02] I. Broster, A. Burns, and G. Rodríguez-Navas. Probabilistic analysis of CAN with faults. In *Proceedings of the 23rd Real-time Systems Symposium*, pages 269–278, Austin, Texas, 2002. {42,53,83,120}

[BBRN04] I. Broster, A. Burns, and G. Rodríguez-Navas. Comparing real-time communication under electromagnetic interference. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 04)*, Catania, Sicily, Italy, June 2004. Computer Society, IEEE. {40,42,47,76}

[BCC+99] Gordon S. Blair, Fábio M. Costa, Geoff Coulson, Hector A. Duran, Nikos Parlavantzas, Fabien Delpiano, Bruno Dumant, François Horn, and Jean-Bernard Stefani. The design of a resource-aware reflective middleware architecture. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 115–134. Springer-Verlag, 1999. {4}

[BDM93] Michael Barborak, Anton Dahbura, and Minoslaw Malek. The consensus problem in fault-tolerant computing. *ACM Comput. Surv.*, 25(2):171–220, 1993. {25}

[Bel02] Belschner, R. *et al.* FlexRay Requirements Specification, version 2.0.2. *FlexRay Consortium, http://www.flexray-group.com*, 2002. {v,68,69}

[Ber88] P. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 21(2):37–45, 1988. {35,162}

[BES+96] F. Brasileiro, P. Ezhilchelvan, S. Shrivastava, N. Speirs, and S. Tao. Implementing Fail-Silent Nodes for Distributed Systems. *IEEE Transactions on Computers*, 45(11):1226–1238, 1996. {35}

[BF93] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Trans. Softw. Eng.*, 19(1):3–12, 1993. {20}

[BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems* Addison-Wesley Longman Publishing Co., Inc., 1986. {31}

[BHRT03] Roberto Baldoni, Jean-Michel Hélary, Michel Raynal, and Lenaik Tangui. Consensus in byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210, 2003. {24,25}

[BKS03] Günther Bauer, Hermann Kopetz, and Wilfried Steiner. The central guardian approach to enforce fault isolation in a time-triggered system. *The 6th International Symposium on Autonomous Decentralized Systems (ISADS 2003)* April 2003. {35,36,65}

[BMST93] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary Backup Approach. *S. Mullander, ed.,chapter 8, second edition*, pages 199–216, 1993. {37}

[BOS91] Robert BOSCH. *CAN Specification Version 2.0.* Postfach 300240, D-7000 Stuttgart 30, 1991. {48,49,78,79,89}

[Bos04] Bosch TT CAN homepage. Time Triggered Communication on CAN. *http://www.can.bosch.com/content/TT_CAN.html*, 2004. {55}

[BPB+00] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The meaning and role of value in scheduling flexible real-time systems. *J. Syst. Archit.*, 46(4):305–325, 2000. {2}

[BPSW99a] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proceedings of the conference*

on Dependable Computing for Critical Applications, page 361. IEEE Computer Society, 1999. {41}

[BPSW99b] A. Burns, S. Punnekkat, L. Stringini, and D.R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Proceedings of the* 7^{th} *International Working Conference on Dependable Computing for Critical Applications*, pages 361 – 378. IEEE Society Press, 6 - 8 Jan 1999. {121}

[BRNPA04] M. Barranco, G. Rodríguez-Navas, J. Proenza, and L. Almeida. CANcentrate: An active star topology for CAN networks. *Proceedings of the* 5^{th} *IEEE International Workshop on Factory Communication Systems (WFCS 2004)*, 2004. {48,159,170}

[Bro03] I Broster. *Flexibility in Dependable Communication*. PhD thesis, Department of Computer Science, University of York, York, YO10 5DD, UK, Aug 2003. {11,41,47,60}

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985. {23}

[CASD85] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press. {122}

[CDV01] Gianluca Cena, Luca Durante, and Adriano Valenzano. A new can-like field network based on a star topology. *Comput. Stand. Interfaces*, 23(3):209–222, 2001. {48}

[CEN96] CENELEC. European standard EN 50170. Fieldbus: Vol.1: P-Net. *European Committee for Electrotechnical Standardisation*, 1996. {15}

[CF04] Mário Calha and José A. Fonseca. Approaches to the ftt-based scheduling of tasks and messages. In *Proceedings. 2004 IEEE International Workshop on Factory Communication Systems*, pages 3–11. IEEE Computer Society, 2004. {168}

[CFP03] Francisco Carreiro, José A. Fonseca, and Paulo Pedreiras. Virtual Token-Passing Ethernet - VTPE. *Proceedings of FeT'2003* 5^{th} *IFAC Conference on Fieldbus Technology*, 2003. {15}

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. {20}

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 2001. {145}

[CHTCB96] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, New York, USA, 1996. ACM. {39}

[CiA99] CiA. CAN physical layer. *http://www.can-cia.de/CANdll.pdf*, 1999. {116}

[CiA02] CiA. *CANopen*. CiA (CAN in Automation), 2002. EN 50325-4 Standard. {48}

[Con04a] FlexRay Consortium. FlexRay Communications System - Electrical Physical Layer Specification, v2.0. Technical report, FlexRay Consortium, 2004. {70}

[Con04b] FlexRay Consortium. FlexRay Communications System - Protocol Specification, v2.0. Technical report, FlexRay Consortium, 2004. {16,36,68}

[Cor99] Echelon Corporation. *Introduction to the LONWORKS System, Version 2.0* Echelon Corporation, 1999. {18}

[Cri91] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–188, 1991. {39}

[CT91] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 325–340. ACM Press, 1991. {24}

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. {22,26}

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987. {26}

[DS98] Assia Doudou and André Schiper. Muteness detectors for consensus with byzantine processes. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, page 315. ACM Press, 1998. {25}

[DSS98] X. Defago, A. Schiper, and N. Sergent. Semi-Passive Replication. *Proceedings of Symposium on Reliable Distributed Systems*, pages 43–50, 1998. {37,137}

[DSU03] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. Research Report IS-RR-2003-009, Japan Advanced Institute of Science and Technology, Ishikawa, Japan, September 2003. {v,33}

[FAA88] FAA. Advisory circular ac 25.1309-1a. Technical report, Federal Aviation Administration, 1988. {20}

[FAF+03] J. Ferreira, L. Almeida, J. Fonseca, G. Rodriguez-Navas, and J. Proenza. Enforcing Consistency of Communication Requirements Updates in FTT-CAN. *Proceedings of the Workshop on Dependable Embedded Systems, held in conjunction with the* 22^{nd} *Symposium on Reliable Distributed Systems (SRDS 2003)*, pages 7–12, October 2003. {126}

[Fg02] FlexRay-group. Handouts of the international flexray workshop. *FlexRay Consortium at* `http://www.flexray-group.com`, April 2002. {67,68}

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982. {25}

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. {23}

[FMD+00] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time triggered communication on CAN. In *Proceedings of* 7^{th} *International CAN Conference*. CAN in Automation GmbH, Oct 2000. {v,55}

[FNP+98] P. Ferriol, F. Navio, J. Pons, J. Proenza, and J. Miro-Julia. A double CAN architecture for fault-tolerant control systems. In *5th International CAN Conference, ICC'98*, San Jose CA, Nov 1998. {78}

[FOFF04] J. Ferreira, A. Oliveira, P. Fonseca, and J. A. Fonseca. An experiment to assess bit error rate in CAN. *RTN 2004 -* 3^{rd} *Int. Workshop on Real-Time Networks satellite held in conjunction with the* 16^{th} *Euromicro Intl Conference on Real-Time Systems*, June 2004. {42,47,76}

[FPAF02] J. Ferreira, P. Pedreiras, L. Almeida, and J. Fonseca. Achieving fault tolerance in FTT-CAN. *Proceedings of the* 4^{th} *Workshop on Factory Communication Systems (WFCS 2002)*, 2002. {113,126}

[Fre95] L.-B. Fredriksson. A CAN kingdom can kingdom rev. 3.01. Technical report, KVASER AB, Sweden, 1995. {48}

[FSMF98] P. Fonseca, F. Santos, A. Mota, and J. A. Fonseca. A dynamically reconfigurable CAN system. *Proceedings of* 5^{th} *International CAN Conference*, 1998. {113,131}

[GMM97] Sunondo Ghosh, Rami Melhem, and Daniel Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(3):272–284, 1997. {121}

[Gro03a] Bluetooth Special Interest Group. Bluetooth core specification v1.2. `https://www.bluetooth.org/spec/`, 2003. {14}

[Gro03b] ETHERNET Powerlink Standardization Group. Ethernet powerlink presentation. *http://www.can-cia.de/CANdll.pdf*, 2003. {14}

[GT91] Ajei Gopal and Sam Toueg. Inconsistency and contamination (preliminary version). In *Proceedings of the* 10^{th} *annual ACM symposium on Principles of distributed computing*, pages 257–272. ACM Press, 1991. {33,34}

[Hav86] N. Haverty. MIL-STD 1553 - a standard for data communications. *Communication & Broadcasting*, 10(1):29–33, 1986. {14}

[HK03] Günther Bauer Hermann Kopetz. The time-triggered architecture,. In *Proceedings of the IEEE*, 2003. {63}

[HKD97] H. Hilmer, H.-D. Kochs, and E. Dittmar. A fault-tolerant communication architecture for real-time control systems. In *Proc. IEEE Int. Workshop on Factory Communication Systems*, Barcelona, Spain, Oct 1997. {78}

[HM90] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, Jul 1990. {12}

[HNP00] H. Hansson, C. Norström, and S. Punnekkatt. Integrating reliability and timing analysis of CAN-based systems. *Proceedings of IEEE Workshop on Factory Communications Systems (WFCS-2000)*, 2000. {83,94}

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols* Prentice-Hall, Englewood Cliffs, New Jersey, 1991. {146}

[Hol04] G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual* Addison-Wesley, Reading, Massachusetts, 2004. {146}

[HS95] M. Hiltunen and R. Schlichting. Properties of membership services. In *Proceedings of the Second International Symposium on Autonomous Decentralized Systems*, pages 200–207, 1995. {39}

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994. {*v*,26,27,28,29,30,32,33}

[iA] CAN in Automation. Can history. `http://www.can-cia.de/can/protocol/history/history.html`; accessed February 18, 2005. {47}

[IEC00] IEC. IEC International Standard 61158: Fieldbus standard for use in industrial control systems - Type 1: Foundation Fieldbus H1; Type 3: PROFIBUS; Type 7: WorldFIP. *International Electrotechnical Committee*, 2000. {14,15,170}

[IEE00] IEE. EMC and functional safety. IEE guidance document, IEE, Sept 2000. Available from `http://www.iee.org.uk/PAB/EMC/core.htm` {40}

[ISO93] International Standards Organisation. *ISO 11898. Road Vehicles—Interchange of digital information—Controller area network (CAN) for high speed communication*, 1993. {5,48,49,78}

[ISO00] ISO. *ISO 11898-4, Road Vehicles—Interchange of digital information—Controller area network (CAN) part 4: Time triggered Communication* International Standards Organisation, 2000. Working Draft. {16}

[ISO01] ISO. Road vehicles - controller area network (CAN) - part 4: Time triggered communication, 2001. {53,56}

[IXX05] IXXAT. FO-Star-Coupler. *http://www.ixxat.de/fo-star-coupler_en,7460,5873.html; accessed February 21, 2005.*, 2005. {48}

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, January 1994. {16,37,63}

[KL99] J. Kaiser and M. Livani. Achieving Fault-Tolerant Ordered Broadcasts in CAN. *Proceedings of the European Dependable Computing Conference*, pages 351–363, 1999. {122,123,137}

[KMMS97] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. Solving consensus in a byzantine environment using an unreliable fault detector. In *in Proceedings of the First Int. Symposium on Principles of Distributed Systems (OPODIS '97)*, pages 61–76, 1997. {25}

[KNH+98] Hermann Kopetz, Roman Nossal, René Hexel, Andreas Krüger, Dietmar Millinger, Roman Pallierer, Christopher Temple, and Markus Krug. Mode handling in the time-triggered architecture. *Control Engineering Practice*, 6(1):61–66, January 1998. {67}

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Press, 1997. {1,5,12,45,74,92,122,151}

[Kop98] H. Kopetz. A comparison of CAN and TTP. Technical Report 1998, Technishe Universitat Wien, Austria, 1998. {47}

[KS94] Hagbae Kim and Kang G. Shin. Modeling of externally-induced/common-cause faults in fault-tolerant systems. In *Proceedings of the 13th Digital Avionics Systems Conference*, pages 402–407. AIAA/IEEE, October 1994. {41}

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974. {72}

[Lap92] J. C. Laprie. *Dependability—Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-tolerant Systems* Springer-Verlag, 1992. IFIP WG 10.4. {18,19}

[Lat03] Elizabeth Latronico. Problems Facing Group Membership Specifications for X-by-Wire Protocols. *Proceedings of the IEEE International Conference on Dependable Systems and Networks, student paper*, 2003. {20}

[LB03] G. M. A. Lima and A. Burns. A consensus protocol for CAN-based systems. In *Proceedings of the 24th Real-time Systems Symposium*, pages 420–429, Cancun, Mexico, Dec 2003. Computer Society, IEEE. {122}

[LC86] A. L. Leistman and R. H. Campbell. A fault-tolerant scheduling problem. *IEEE Trans. Softw. Eng.*, 12(11):1088–1089, 1986. {121}

[Le 92] Gérard Le Lann. Designing real-time dependable distributed systems. *Comput. Commun.*, 15(4):225–234, 1992. {46}

[Lev00] Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. Softw. Eng.*, 26(1):15–35, 2000. {20}

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. {24,25}

[LSTS02] C. Lu, J. Stankovic, G. Tao, and S. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Special issue of RT Systems Journal on Control-Theoretic Approaches to Real-Time Computing* 23(1/2):85–126, Jul/Sept 2002. {2}

[LSZ+01] Joseph P. Loyall, Richard E. Schantz, John A. Zinky, Partha Pratim Pal, Richard Shapiro, Craig Rodrigues, Michael Atighetchi, David A. Karr, Jeanna Gossett, and Christopher D. Gill. Comparing and contrasting adaptive middleware support in wide-area and embedded distributed object applications. In*ICDCS*, pages 625–634, 2001. {3}

[M. 00] M. Peller and J. Berwanger and R. Griebach. ByteFight - A New High-Performance Data Bus System for Safety-Related Applications. *BMW AG, EE-211 Development Safety Systems Electronics*, 2000. {16,69}

[MAF05] Ernesto Martins, Luís Almeida, and José Alberto Fonseca. An FPGA-based Coprocessor for Real-Time Fieldbus Traffic Scheduling - Architecture and Implementation. *Journal of Systems Architecture*, 51:29–44, January 2005. {132}

[MF01] E. Martins and J. Fonseca. Improving flexibility and responsiveness in FTT-CAN with a scheduling coprocessor. *Proceedings of FeT'2001 - 4^{th} FeT IFAC Conference Fieldbus Technology*, 2001. {60}

[MFA+02] E. Martins, J. Ferreira, L. Almeida, P. Pedreiras, and J. Fonseca. An Approach to the Synchronization of Backup Masters in Dynamic Master-Slave Systems. *Proceedings of the WiP Session of 23^{rd} IEEE International Real-Time Systems Symposium (RTSS)*, 2002. {126}

[MFH+02] B. Müller, T. Führer, F. Hartwich, R. Hugel, and H. Weiler. Fault tolerant ttcan networks. In *Proceedings of 8^{th} International CAN Conference*. CAN in Automation GmbH, Oct 2002. {53}

[MIS95] MISRA. Integrity, MISRA Report 2. Technical report, The Motor Industry Software Reliability Association, 1995. {21}

[MR97] Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *CSFW '97: Proceedings of the 10th Computer Security Foundations Workshop (CSFW '97)*, page 116. IEEE Computer Society, 1997. {25}

[MRJ97] Ashish Mehra, Jennifer Rexford, and Farnam Jahanian. Design and evaluation of a window-consistent replication service. *IEEE Trans. Comput.*, 46(9):986–996, 1997. {37,38}

[MSCV04] Pedro Martins, Paulo Sousa, Antonio Casimiro, and Paulo Verissimo. Dependable adaptive real-time applications in wormhole-based systems. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)* page 567. IEEE Computer Society, 2004. {4}

[NFG+02] Roman Nossal, Emmerich Fuchs, Thomas M. Galla, Roland Lang, Dietmar Millinger, and Michael Sprachmann. How to Build Automotive Applications Based on the FlexRay Communication System. *Proceedings of the SAE World Congress*, March 2002. {68}

[NSS00] N. Navet, Y.-Q. Song, and F. Simonot. Worst-case deadline failure probability in real-time applications distributed over CAN (controller area network). *Journal of Systems Architecture*, 46(7), 2000. {83}

[NYQS00] 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(1):607–617, 2000. {41,78,94}

[Obe02] R. Obermaisser. Can emulation in a time-triggered environment. In *Proceedings of the 2002 IEEE International Symposium on Industrial Electronics (ISIE)* volume 1. IEEE, January 2002. {67}

[OFSF03] Arnaldo Oliveira, Pedro Fonseca, Valery Sklyarov, and António Ferrari. An object-oriented framework for CAN protocol modeling and simulation. In *FET2003: 5th IFAC International Conference on Fieldbus Systems and their Applications*, pages 243–248, Aveiro, Portugal, July 2003. {88}

[Oli03] Arnaldo Oliveira. MoiCAN Project. *http://www.ieeta.pt/~arnaldo/projects/MoiCAN/MoiCAN.htm*, 2003. {87}

[PA00] P. Pedreiras and L. Almeida. Combining event-triggered and time-triggered traffic in FTT-CAN: Analysis of the asynchronous messaging system. *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 67–75, 2000. {129}

[PAF00] P. Pedreiras, L. Almeida, and J. Fonseca. Improving the responsiveness of synchronous messaging system in FTT-CAN. *Proceedings of DCCS'2000, IFAC Workshop on Distributed Computer Control Systems*, 2000. {59}

[PAG02] Paulo Pedreiras, Luís Almeida, and Paolo Gai. The ftt-ethernet protocol: Merging flexibility,timeliness and efficiency. In *ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems* IEEE Computer Society, 2002. {168}

[PBA03] D. Prasad, A. Burns, and M. Atkins. The valid use of utility in adaptive real-time systems. *Real-Time Syst.*, 25(2-3):277–296, 2003. {2}

[PBG99] M. Peller, J. Berwanger, and R. Grießbach. *byteflight specification*. BWM AG, draft edition, Nov 1999. Available From `www.byteflight.com`. {15,16}

[PD95] M. Peraldi and J. Decotignie. Combining real-time features of local area networks FIP and CAN. *Proc. of ICC'95 (2^{nd} International CAN Conference)*, 1995. {56}

[Ped03] Paulo Pedreiras. *Supporting Flexible Real-Time Communication on Distributed Systems*. PhD thesis, University of Aveiro, Portugal, July 2003. {5,58,168}

[PF04] Juan R. Pimentel and José Alberto Fonseca. Flexcan: A flexible architecture for highly dependable embedded applications. *RTN 2004 - 3^{rd} Int. Workshop on Real-Time Networks sattelite held in conjunction with the 16^{th} Euromicro Intl Conference on Real-Time Systems*, June 2004. {v,47,61,62,74}

[PHN00] S. Punnekkat, H. Hansson, and C. Norström. Response time analysis under errors for CAN. *Real-Time Technology and Applications Symposium (RTAS'2000)*, 2000. {40}

[Pim04] Juan R. Pimentel. An architecture for a safety-critical steer-by-wire system. *Proceedings of the SAE World Congress 2004*, 2004. {61}

[PK04] Juan R. Pimentel and John Kaniarz. A can-based application level error detection and fault containment protocol. *Proceedings of the INCOM'04,* 11^{th} *IFAC Symp. on Information Control Problems in Manufacturing*, 2004. {61}

[PMJ00] J. Proenza and J. Miro-Julia. MajorCAN: A modification to the Controller Area Network to achieve Atomic Broadcast. *IEEE Int. Workshop on Group Communication and Computations. Taipei, Taiwan*, 2000. {74,78,79,81,99,122,123}

[Pnu79] Amir Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 1–20. Springer-Verlag, 1979. {146,149}

[Pow91] D. Powell. *Delta-4 - A generic Architecture for Dependable Distributed Computing.* 1991. {151}

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. {22,25}

[PV03] Luís Miguel Pinho and Francisco Vasques. Reliable real-time communication in can networks. *IEEE Trans. Comput.*, 52(12):1594–1607, 2003. {122,137}

[RGS95] R. Rajkumar, M. Gagliardi, and Lui Sha. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. In *Proceedings of the Real-Time Technology and Applications Symposium*, page 66. IEEE Computer Society, 1995. {37,38}

[RNPR+04] G. Rodríguez-Navas, J. Proenza, J. Rigo, J. Ferreira, L. Almeida, and J. Fonseca. Design and Modeling of a Protocol to Enforce Consistency among Replicated Masters in FTT-CAN. *Proceedings of the* 5^{th} *Workshop on Factory Communication Systems (WFCS 2004)*, pages 229–238, 2004. {126}

[RP03] Guillermo Rodríguez-Navas and Julián Proenza. Analyzing Atomic Broadcast in TTCAN Networks. In *Proceedings of the* 5^{th} *IFAC International Conference on Fieldbus Systems and their Applications (FET 2003), Aveiro, Portugal*, pages 153–156, 2003. {v,47,76,80,84,85}

[RTC00] RTCA/EUROCAE. Requirements Specification for Avionics Computer Resource (ACR). Technical report, Radio Technical Commission for Aeronautics/European Organisation for Civil Aviation Equipment, 2000. {1,45}

[Ruc94] M. Rucks. Optical layer for CAN. In *Proceeding of the* 1^{st} *International CAN Conference*, volume 2, pages 11–18, Mainz, Germany, 1994. CiA. {48}

[Rus99] John Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA. {11}

[Rus01] J. Rushby. Bus Architectures For Safety-Critical Embedded Systems.*Proceedings of the First Workshop on Embedded Software (EMSOFT 2001)*, 2211:306–323, 2001. {11,36}

[RV97] J. Rufino and P. Veríssimo. Hard real-time operation of CAN. *CSTC Technical Report RT-97-02*, 1997. {83,152}

[RVA+98] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodigues. Fault-tolerant broacast in CAN. *Digest of Papers,* 28_{th} *International Symposium on Fault Tolerant Computer Systems*, pages 150–159, 1998. {40,78,79,80,81,84,85,92,122,137}

[RVA99a] J. Rufino, P. Veríssimo, and G. Arroz. A Columbus' egg idea for CAN media redundancy. In *Digest of Papers, The 29th International Symposium on Fault-Tolerant Computing Systems*, pages 286–293, Madison, Wisconsin, USA, Jun 1999. IEEE. {48}

[RVA99b] J. Rufino, P. Veríssimo, and G. Arroz. A Columbus' egg idea for CAN media redundancy. In *Digest of Papers, The 29th International Symposium on Fault-Tolerant Computing Systems*, pages 286–293, Madison, Wisconsin, USA, June 1999. IEEE. {100}

[RWS01] Binoy Ravindran, Lonnie Welch, and Behrooz Shirazi. Resource management middleware for dynamic, dependable real-time systems. *Real-Time Systems*, 20(2):183–196, 2001. {4}

[Sch90] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4), December 1990. {30,37,137}

[SES+92] S. Shrivastava, P. Ezhilchelvan, N. Speirs, S. Tao, and A. Tully. Principal Features of the Voltan Family of Reliable Node Architectures for Distributed Systems. *IEEE Transactions on Computers (Special Issue on Fault-Tolerant Computing)* 41(5):542–549, 1992. {35}

[She03] C. Shelton. *Scalable Graceful Degradation for Distributed Embedded Systems* PhD thesis, Carnegie Mellon University, 2003. {3}

[SK04] Charles P. Shelton and Philip Koopman. Improving system dependability with functional alternatives. In *Proceedings of the 2004 International Conference on*

Dependable Systems and Networks (DSN'04), page 295. IEEE Computer Society, 2004. {3}

[SLST99] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao. The case for feedback control real-time scheduling. *Proceedings of Euromicro Conference on Real-Time Systems*, pages 11–20, 1999. {2}

[SR89] John A. Stankovic and K. Ramamritham. *Tutorial: hard real-time systems*. IEEE Computer Society Press, 1989. {12}

[SRG94] L. Sha, R. Rajkumar, and M. Gagliardi. The simplex architecture: An approach to building evolving industrial computing systems. In *Proceedings of the International Conference on Reliability and Quality in Design*, pages 122–126. ISSAT Press, 1994. {2}

[SSM+] D. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems, CrossTalk, November. 2001. Available from: `http://www.cs.wustl.edu/~schmidt/PDF/crosstalk.pdf`; accessed February 21, 2005. {3}

[TBW95] K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995. {18,40,83}

[Tem98] C. Temple. Avoiding the babbling-idiot failure in a time-triggered communication system. *Fault Tolerant Computing Symposium*, pages 218–227, 1998. {34,65,151}

[TH95] K. Tindell and H. Hansson. Babbling idiots, the dual-priority protocol, and smart CAN controllers. In *Proceedings of the 2nd International CAN Conference*, pages 7.22–28, 1995. {52}

[Tho93] Jean-Pierre Thomesse. Time and industrial local area networks. *Proceedings of COMPEURO'93, Paris, France*, 1993. {14}

[Tho98] J. P. Thomesse. A review of the fieldbuses. *Annual Reviews in Control*, 22:35–45, 1998. {13}

[TTT02] TTTech. Time-Triggered Protocol TTP/C High-Level Specification Document (edition 1.0). *http://www.ttagroup.org*, 2002. {16,62}

[VC02] P. Veríssimo and A. Casimiro. The Timely Computing Base Model and Architecture. *IEEE Transactions on Computers - Special Issue on Asynchronous Real-Time Systems*, 51(8), aug 2002. {4}

[Veg96] L. Vega. *Modèles de Coopération et de Communication entre Processus Temps Réel Répartis. Expression de Contraintes de Temps pour la Vérification de Propriétés Temporelles dans la Communication* PhD thesis, CRIN - Institut National Polytechnique de Lorraine, Nancy, France, 1996. {12}

[Ver03] Paulo Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, pages –. Springer-Verlag LNCS 2584, May 2003. {4}

[WB91] S. Webber and J. Beirne. The Stratus Architecture. *Digest of Papers FTCS-21*, pages 79–85, 1991. {35,162}

[WBDP98] A. J. Wellings, L. Beus-Dukic, and D. Powell. Real-time scheduling in a generic fault-tolerant architecture. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*, page 390. IEEE Computer Society, 1998. {121}

[Wen78] Wensley, J. *et al.* SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control. *Proceedings of IEEE*, 66(10):1240–1255, 1978. {35}

[WPS+00] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C., April 2000. IEEE Computer Society Technical Commitee on Distributed Processing. {37,138}

[zBT93] Özalp Babaoglu and Sam Toueg. Non-blocking atomic commitment. In *Distributed systems (2nd Ed.)*, pages 147–168. ACM Press/Addison-Wesley Publishing Co., 1993. {31}

[ZJ98] H. Zou and F. Jahanian. Real time primary-backup (rtpb) replication with temporal consistency guarantees. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 48–56. IEEE Computer Society, 1998. {37}

Appendix A

Low level details of the SRT update protocol

This Appendix presents some low level details of the SRT update protocol, in the form of flowcharts, that are closer to the implementation.

Figure A.1 shows the flowchart of a slave node and highlights two phases, one in which the slave waits for a successful request transmission and another in which the slave waits for the reply. Both phases are time bounded.

Figure A.2 depicts the flowcharts of the active and backup masters. Their functionality is similar, however, the backup master uses the protocol information conveyed in the trigger message sent by the active master to synchronize its internal state. The backup masters also declares itself unsynchronized when it receives a trigger message with a request that is not in its queue or the result of the admission control differs from the one achieved by the active master.

Figure A.3 shows the flowcharts of the interrupt handlers related with the transmission of the trigger message by the active master and with is reception by the backup master. The commit operation in case of a successful reply to an update request is made during the execution of these interrupt handlers using a global variable (commit*) to synchronize this operation. This variable is set in the active and backup masters request handling tasks and it is reset in these interrupt handlers.

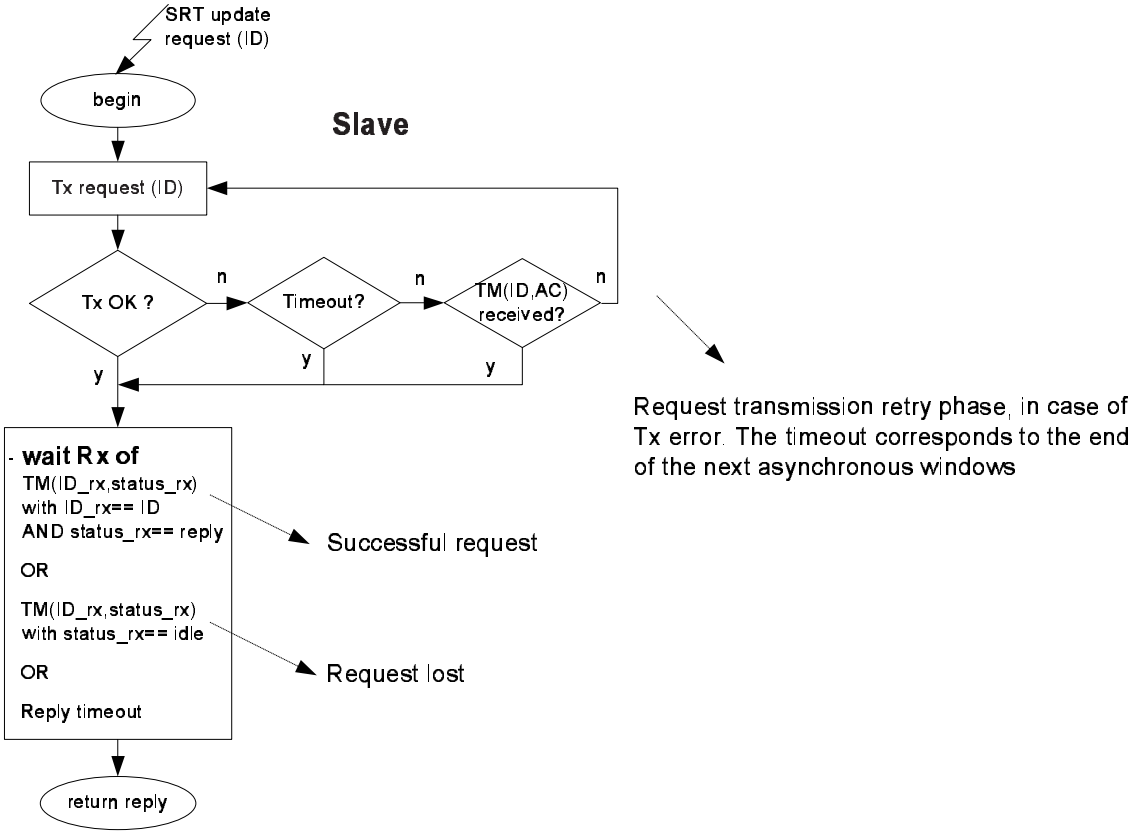


Figure A.1: Slave's flowchart.

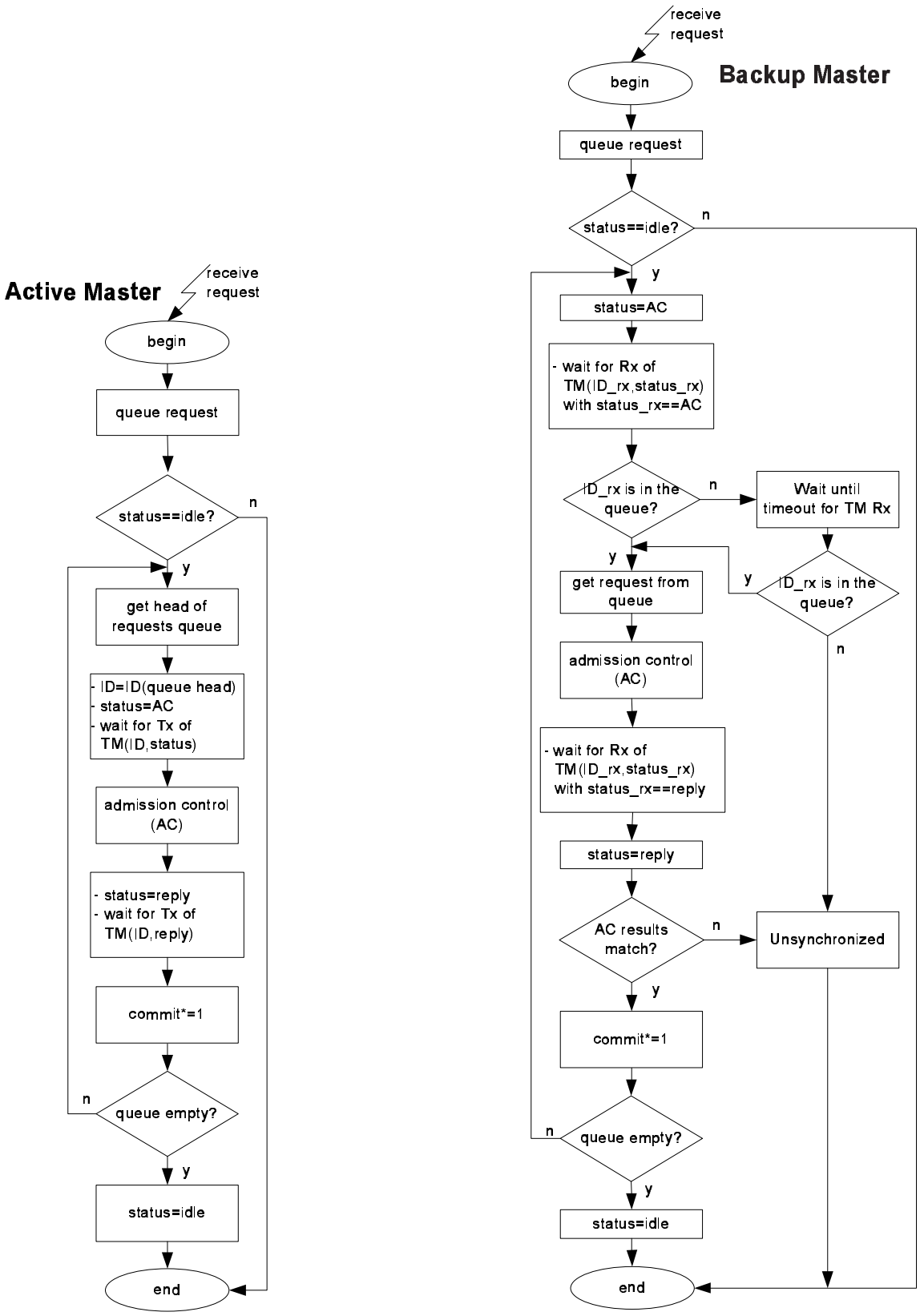


Figure A.2: Active (left) and backup (right) master flowcharts.

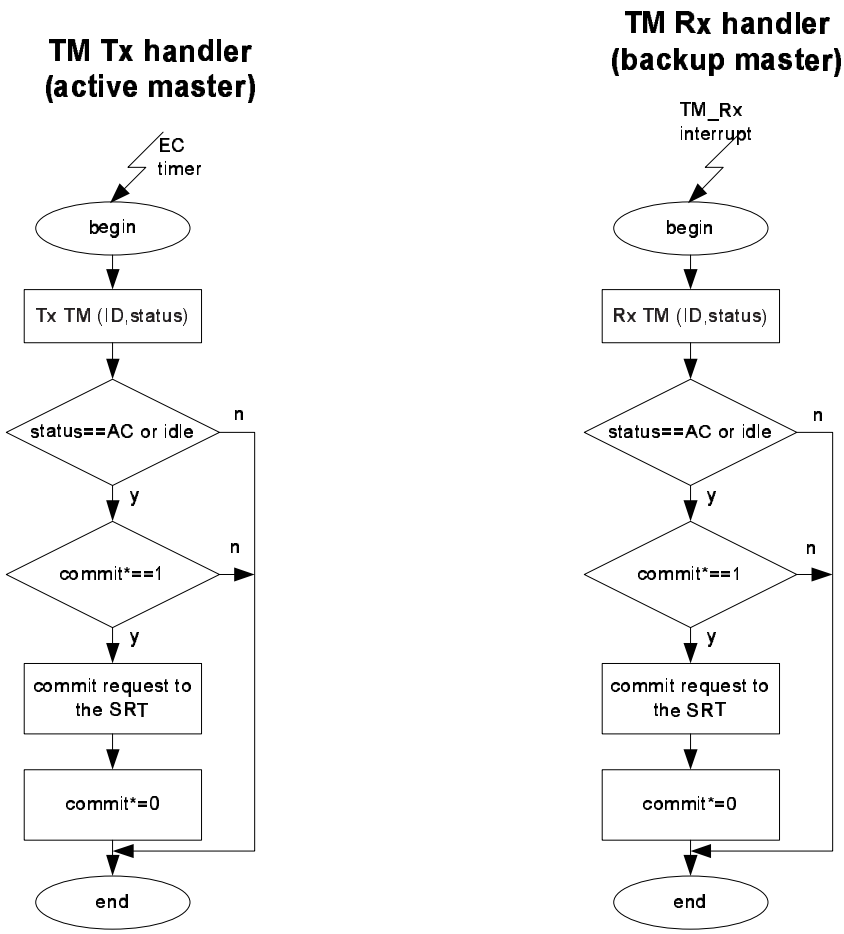


Figure A.3: Trigger message transmission handler (left) and rigger message reception handler (right) flowcharts.

Appendix B

Table of Abbreviations

Abbreviation	Meaning
BER	Bit Error Rate
CAN	Controller Area Network
COTS	Commercial off-the-shelf (COTS)
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA	Carrier-Sense Multiple Access
CSMA-CA	CSMA - Collision Avoidance
CSMA-CD	CSMA - Collision Detection
CSMA-DCR	CSMA - Deterministic Collision Resolution
CSMA-BA	CSMA - Bitwise Arbitration
DES	Distributed Embedded Systems
EC	Elementary Cycle
EMI	Electromagnetic Interference
IMD	Inconsistent Message Duplicate
IMO	Inconsistent Message Omission
ISO	International Standards Organization
ET	Event-Triggered
FPGA	Field-Programmable Gate Array
FTT	Flexible Time-Triggered protocol
FTT-CAN	Flexible Time-Triggered protocol on CAN
FTT-Ethernet	Flexible Time-Triggered protocol on Ethernet
IMA	Integrated Modular Avionics

continues ...

continued ...

Abbreviation	**Meaning**
LAW	Minimum Length of the Asynchronous Window
law(i)	Length of the Asynchronous Window of EC i
LSW	Upper bound for the Length of the Synchronous Window
lsw(i)	Length of the Synchronous Window of EC i
MAC	Medium Access Control
MEDL	Message Descriptor List
OSI	Open Systems Interconnection
QoS	Quality of Service
SRDB	Synchronous Requirements Database
TCAN	Timely CAN
TDMA	Time-Division Multiple Access
TM	Trigger Message
TT	Time-Triggered
TTCAN	Time-Triggered CAN
TTP	Time-Triggered Protocol

Appendix C

List of publications

Papers with the candidate as first author:

- J. Ferreira, L. Almeida, J. Fonseca, P. Pedreiras, M. Mauro. On the dependability and flexibility of CAN and CAN based protocols. *VII Workshop de Tempo Real*, 13 May 2005, Fortaleza, Brazil.
- J. Ferreira, L. Almeida, J. A. Fonseca, P. Pedreiras, E. Martins, G. Rodríguez-Navas, J. Rigo, J. Proenza. Combining Operational Flexibility and Dependability in FTT-CAN. Submitted to IEEE Transactions on Industrial Informatics.
- J. Ferreira, A. Oliveira, P. Fonseca, J. Fonseca. An Experiment to Assess Bit Error Rate in CAN. In *Proceedings of* 3^{rd} *International Workshop on Real-Time Networks*, 29 July 2004, Catania, Italy.
- J. Ferreira, L. Almeida, J. Fonseca, G. Rodriguez-Navas, J. Proenza. Enforcing Consistency of Communication Requirements Updates in FTT-CAN. In *Proceedings of Workshop on Dependable Embedded Systems workshop satellite of the* 22^{nd} *Symposium on Reliable Distributed Systems (SRDS 2003)*, October 2003, Florence, Italy.
- J. Ferreira, L. Almeida, E. Martins, P. Pedreiras, J. A. Fonseca. Components to Enforce Fail-Silence Behavior in Dynamic Master-Slave Systems. In *Proceedings of* 5^{th} *IFAC International Symposium on Intelligent Components and Instruments for Control Applications (SICICA 2003)*, June 2003, Aveiro, Portugal.
- J. Ferreira, P. Pedreiras, L. Almeida, J. Fonseca, The FTT-CAN protocol: improving flexibility in safety-critical systems. IEEE Micro (special issue on Critical Embedded Automotive Networks), volume 22, number 4, July/August 2002, pp 46-55.
- J. Ferreira, P. Pedreiras, L. Almeida, J. A. Fonseca. Achieving Fault Tolerance in FTT-CAN. In *Proceedings of* 4^{th} *IEEE International Workshop on Factory Communication Systems (WFCS'02)*, 2002, Västerås, Sweden.

- J. Ferreira, P. Pedreiras, L. Almeida, J. A. Fonseca. FTT CAN Error Confinement. In Proceedings of 4^{th} IFAC Conference Fieldbus Technology (FeT'2001), 2001 Nancy, France.

Other papers co-authored by the candidate:

- J. Fonseca, J. Ferreira, E. Martins, "Future Trends in the Hardware of Embedded Systems", Embedded Real-Time Systems Implementation (ERST 2004) Workshop, 2004, Lisboa, Portugal.

- G. Rodríguez-Navas, J. Rigo, J. Proenza, J. Ferreira, L. Almeida, J. Fonseca. Design and Modeling of a Protocol to Enforce Consistency of the Replicated Table of Communication Requirements in FTT-CAN", In *Proceedings of* 5^{th} *IEEE International Workshop on Factory Communication Systems*, September 2004, Vienna, Austria.

- E. Martins, J. Ferreira, L. Almeida, P. Pedreiras, J. Fonseca. An Approach to the Synchronization of Backup Masters in Dynamic Master-Slave Systems. *In Proceedings of RTSS'02, IEEE Real-Time Systems Symposium, Work in Progress session*, December 2003, Austin, USA.

- J. Fonseca, J. Ferreira, M. Calha, P. Pedreiras, L. Almeida. Issues on Task Dispatching and Master Replication in FTT-CAN. In *Proceedings of IEEE AFRICON'02*, 2003, George, South Africa.

- L. Almeida, José A. Fonseca, A. Mota, P. Fonseca, E. Martins, P. Pedreiras, J. Ferreira. Flexibility, Timeliness and Efficiency in Fieldbus Systems: The DISCO Project Approach. Proceedings of 8^{th} IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2001), 2001, Antibes, France.