Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Electrotécnica e de Computadores

A Framework for the Transparent Replication of Real-Time Applications

By

Luís Miguel Rosário da Silva Pinho

A dissertation submitted in partial fulfilment of the requirements for the degree of Doctor in Electrical and Computer Engineering

September 2001

Doctoral Committee:

Prof. Joaquim Silva GOMES	Chairman
Prof. Andy WELLINGS Prof. Paulo VERÍSSIMO Prof. Eduardo TOVAR	External Examiners
Prof. Adriano Silva CARVALHO Prof. Pedro SOUTO Prof. José Magalhães CRUZ	Internal Examiners
Prof. Francisco VASQUES	Supervisor

A Framework for the Transparent Replication of Real-Time Applications

Abstract

Computer control systems are used in a wide range of application domains, such as factory automation, process control, robotics, automotive systems, etc. The development of such applications is a complex task, often requiring the integration of fault tolerance and real-time properties. The use of Commercial-Off-The-Shelf (COTS) components presents a significant new challenge, since these components do not usually support fault tolerance mechanisms. Moreover, the use of the pre-emptive fixed priority computational model in these applications presents significant problems, due to the increased difficulty in managing the determinism of replicated application components. Therefore, current computer control applications are becoming more complex to develop and maintain, since they are required to implement the mechanisms needed to support replication and distribution.

The main research objective of this thesis is to develop a transparent and generic framework to support the replication of multitasking applications, considering the use of COTS components. The target of such framework is to allow the development of applications focusing on the requirements of the controlled system, and abstracting from the low-level details of replication and distribution mechanisms.

In this thesis, a framework for the development of fault-tolerant real-time applications is proposed, based on the transparent replication of application components. The main focus is given to the support of Ada 95 applications conforming to the Ravenscar profile. The proposed framework provides a set of generic task interaction objects, which are used as the basic building blocks of the application. These objects provide the usual task interaction mechanisms used in hard real-time applications, and allow applications to be developed without considering replication and distribution issues.

The communication support for the replication of software components is provided by a set of atomic multicast and consolidation protocols, guaranteeing fault-tolerant real-time communication in CAN networks. These protocols maintain the predictability of CAN message transfers in spite of the CAN inconsistent message transfer, considering the possible occurrence of either bus or nodes' network interface errors.

A prototype was also developed to assess the expressiveness of the Ravenscar profile for the development of fault-tolerant real-time systems, considering the proposed generic and transparent approach.

Keywords: *Real-Time Systems, Fault-Tolerant Systems, COTS, Ada 95, CAN Networks.*

Uma Infra-Estrutura para a Replicação Transparente de Aplicações de Tempo-Real

Resumo

Os sistemas de controlo por computador são utilizados num largo espectro de aplicações, tais como automação industrial, controlo de processos, robótica, sistemas automóveis, etc. O desenvolvimento destas aplicações é uma tarefa complexa, devido à necessidade de integrar os requisitos de tempo-real e tolerância a falhas. A utilização de componentes de uso genérico apresenta um novo desafio, pela ausência de suporte a mecanismos de tolerância a falhas. Também a utilização do modelo computacional preemptivo por prioridades fixas, introduz problemas adicionais, devido à dificuldade acrescida de gerir o determinismo dos componentes replicados das aplicações. É assim que as aplicações de controlo por computador são cada vez mais complexas para desenvolver e manter, porque é necessário que implementem directamente os mecanismos necessários ao suporte de replicação e distribuição.

O principal objectivo desta tese é o de desenvolver uma infra-estrutura transparente e genérica para suportar a replicação de aplicações multitarefa, considerando a utilização de componentes de uso genérico. O objectivo desta infra-estrutura é permitir que o desenvolvimento das aplicações seja focalizado nos requisitos apresentados pelo sistema controlado, abstraindo-se dos mecanismos de baixo nível de suporte à replicação e distribuição.

Neste tese, é proposta uma infra-estrutura para o desenvolvimento de aplicações de tempo-real e tolerantes a falhas, baseado na replicação transparente de componentes da aplicação. O foco principal é dado ao suporte a aplicações Ada 95, em conformidade com o perfil Ravenscar. A infra-estrutura proposta disponibiliza um conjunto de objectos para interacção entre tarefas, que são usados como blocos básicos para o desenvolvimento das aplicações. Estes objectos implementam os mecanismos de interacção entre tarefas normalmente utilizados em aplicações de tempo-real crítico, e permitem um desenvolvimento das aplicações sem a necessidade de serem considerados os detalhes de replicação e distribuição.

O suporte de comunicações para a replicação dos componentes da aplicação é disponibilizado por um conjunto de protocolos para difusão atómica e para consolidação de réplicas, garantindo comunicação de tempo-real e tolerante a falhas em redes CAN. Estes protocolos mantêm a previsibilidade das comunicações em CAN, apesar das inconsistências na transmissão de mensagens, considerando a possível ocorrência de erros tanto no barramento como nas interfaces de rede dos nós.

Um protótipo foi também desenvolvido para avaliar a expressividade do perfil Ravenscar para o desenvolvimento de sistemas de tempo-real e tolerantes a falhas, considerando a abordagem genérica e transparente proposta.

Palavras-chave: *Sistemas de Tempo-Real, Sistemas Tolerantes a Falhas, Componentes de uso Genérico, Ada 95, Redes CAN.*

Une Infrastructure Logicielle pour la Réplication Transparente d'Applications Temps-Réel

Résumé

Les systèmes de contrôle-commande sont utilisés dans une grande plage de domaines d'application, tels que l'Automatique Industrielle, le Contrôle de Processus, la Robotique, etc. Le développement de ce type d'applications est une tâche complexe, conséquence du besoin d'intégration des propriétés à la fois de tolérance aux fautes et de temps-réel. L'utilisation de matériel sur étagère, c'est-à-dire du matériel à large diffusion non conçu pour un domaine d'applications particulier, présente un défi supplémentaire, car ce type de matériel ne possède pas de mécanismes particuliers pour la tolérance aux fautes. En plus, l'utilisation du modèle préemptif à priorités fixes pose des problèmes supplémentaires, dus aux difficultés de gestion de la réplication de ses composants et de son déterminisme. En conséquence, le développement de ce type d'applications devient de plus en plus complexe, car des mécanismes de distribution et de réplication doivent être intégrés dans le logiciel applicatif.

Le principal objectif de cette thèse est celui de développer une infrastructure logicielle générique pour la réplication transparente d'applications multitâche, en considérant l'utilisation du matériel sur étagère. La cible majeure de cette infrastructure est de permettre la focalisation du développement sur les besoins du système contrôlé, en créant une abstraction sur les détails concernant la réplication et la distribution.

Dans cette thèse, une infrastructure logicielle, basée sur la réplication transparente des composants applicatifs, est proposée pour le développement d'applications temps-réel tolérantes aux fautes. La cible principale de cette infrastructure est le développement d'applications Ada 95, en accord avec le profil Ravenscar. L'infrastructure proposée fournie un ensemble d'objets génériques utilisés pour la construction d'applications. Ces objets fournissent les mécanismes traditionnels pour l'interaction de tâches dans des systèmes temps-réel strict, permettant le développement d'applications, sans que les questions de réplication et de distribution soient prises en compte.

La réplication de composants est supportée par un ensemble de protocoles de diffusion atomique et de consolidation de données, qui garantissent la communication temps-réel et tolérante aux fautes. Ces protocoles garantissent la prévisibilité du transfert de messages, malgré les inconsistances du réseau CAN.

Un prototype a été développé pour évaluer l'aptitude du profil Ravenscar pour le développement des systèmes temps-réel tolérants aux fautes, en considérant l'approche transparente et générique proposée.

Mots-clés: *Systèmes Temps-Réel, Systèmes Tolérants aux Fautes, Matériel sur Etagère, Ada 95, Réseaux CAN.*

para a Ana e
para os meus Pais

Acknowledgements

First, I would like to thank my supervisor, Francisco Vasques, for his permanent support. From the very beginning his dedication was invaluable and without his suggestions and reviews the thesis would never come to a worthwhile end.

I would also like to thank the jury elements for accepting to be members of the doctoral committee, and for the time invested in the evaluation of this thesis.

I benefited greatly from fruitful discussions with many other people. I want to thank them for their comments and suggestions that significantly increased the quality of this work. Particularly, I would like to thank Ian Broster, Alan Burns, António Casimiro, Luis Rodrigues, José Rufino, Paulo Veríssimo and Andy Wellings.

My colleagues and friends in the IPP-HURRAY! research group. Eduardo, Mário, Luis, Berta, Filipe, Ana, Bertil, Sandra e Veríssimo: thanks for the valuable discussions and for accepting extra work. Without your support and friendship I would come across much more difficulties.

I also acknowledge the support provided by my numerous colleagues in the Computer Engineering Department. Special thanks go to Dulce, Helena and Miguel for their continuous encouragement.

Thanks also to the Polytechnic Institute of Porto (IPP) and to its School of Engineering (ISEP) for the support provided, which was important to the conclusion of this work.

I would also like to acknowledge the financial support provided by the Ministério da Educação, through the PRODEP program, by FCT (Fundação para a Ciência e Tecnologia), by FLAD (Fundação Luso-Americana para o Desenvolvimento) and by DEMEGI-FEUP.

Special thanks go to my family for always being present. Particularly, I would like to thank my parents, for giving me all their support during all the different periods of my life.

Finally, I want to emphasise the support provided by my wife, Ana. Without her wonderful patience, her permanent encouragement and her will to sacrifice our weekends, I am sure that I would have not finished this thesis. Her presence has made everything possible.

Porto, 21 September 2001

Luís Miguel Pinho

Table of Contents

Chapter 1

Overview

1.1. Introduction

Computer control systems are used in a wide range of application domains. They can be found in areas such as factory automation, process control, robotics, automotive systems, avionics and space applications. In all of these applications, computers are used to control the surrounding environment, and they are expected to react to external *stimuli* according to the requirements of the controlled environment. As the majority of the targeted application domains present timing requirements, their correct behaviour is expected both in the value and timing domains. Therefore, these systems are also considered as being real-time systems, where the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced (Stankovic, 1988).

Furthermore, computer systems are increasingly expected to perform correctly even in the presence of malfunctioning components. They are required to provide a service in accordance with the specified behaviour in spite of faults, in order to provide fault tolerance to the supported applications (Laprie, 1992). It is thus essential the integration of both the fault tolerance and the real-time requirements of the supported applications in the development of computer control systems.

Currently, Commercial Off-The-Shelf (COTS) components are progressively being considered for the development of computer control systems. Using COTS components as the systems' building blocks provides a cost-effective solution, and at the same time allows for an easy upgrade and maintenance of the system. However, the use of COTS components implies that specialised hardware will not be used to guarantee fault-tolerant and real-time behaviour. As COTS hardware and software do not usually provide the confidence level required by fault-tolerant real-time applications, these requirements must be guaranteed by a software-based fault tolerance approach. This implies that the application of COTS technology to computer control problems is not a simple engineering task.

The recent trend for incorporating pre-emptive multitasking applications in application areas with the above mentioned requirements, increases the complexity of application development. Traditionally, fault-tolerant real-time applications were supported by cyclic executives, providing determinism guarantees (concerning real-time and fault tolerance properties). Nevertheless, using a pre-emptive model increases the system flexibility and decreases development costs (Locke, 1992). Therefore, it is

essential to provide applications with the necessary support for the development of pre-emptive multitasking computer control applications, whilst guaranteeing the required fault tolerance and real-time properties of the controlled system.

These requirements can be guaranteed by providing application redundancy through the replication of application software in a distributed context. Consequently, this requires replication management mechanisms, supported by appropriate communication protocols providing the consistent multicast of information and consolidation of replicated components. However, these mechanisms usually increase the complexity of the application development, since they are directly implemented within the application. A transparent and generic programming model must be devised, allowing applications to be developed without simultaneously considering both the system requirements and the distribution and replication issues.

1.2. Research Context

A computer control system (Figure 1.1) is usually constituted by three subsystems (Kopetz, 1997): the computer system, the controlled system, and a supervision and management system (a human operator or some higher-level computer system). The computer system interacts with the controlled system through input/output devices, which allow the computer system to acquire the state of the controlled environment (sensors) and to change its state (actuators). It interacts with the supervision and management system either through a local console, or through some network interface.

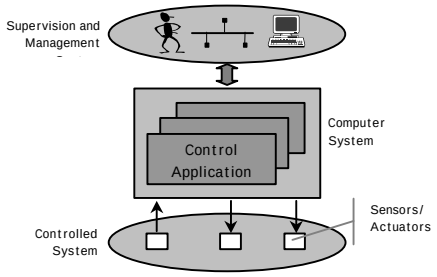

Figure 1.1. Computer control system

The computer system can be constituted by several applications, which control the required behaviour of the controlled system. These applications must also provide the fault tolerance and the real-time properties required by the controlled system.

Often the computer system is required to perform multiple concurrent actions, since the controlled system is inherently concurrent. Therefore, applications are constituted by several tasks (processing units), which may communicate with each other using some form of inter-task interaction mechanisms.

Figure 1.2 presents the example of a simple computer control application, constituted by four tasks, implementing a simple control loop between a sensor and an actuator. The *Sensor* task is responsible for reading the value of the sensor and passing it to the *Controller* task, which in turn performs the control algorithm. An *Actuator* task is then

responsible for the actual writing of the output. The *Alarm* task is responsible for some type of notification in case the *Controller* task signals an abnormal condition.

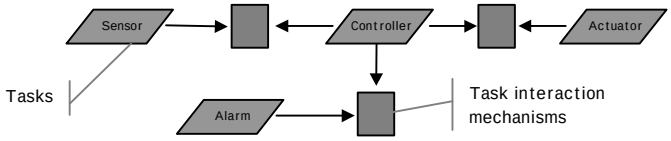

Figure 1.2. Computer control application

A common characteristic among computer control applications is their real-time behaviour. This behaviour is specified in compliance with the system timing requirements. The computer system must process the inputs from the environment and provide the adequate output within an upper-bounded time interval, which is dictated by the requirements of the controlled system. For instance, in Figure 1.2, it is expected that the *Actuator* task outputs the result of the control algorithm, within a time interval related to the arrival of the *Sensor* input. This time interval is imposed by the controlled system.

The simplest architecture that can be considered for a computer control system is represented in Figure 1.3. In this type of architecture (centralised), there is only one single computer unit, which has all the necessary capabilities to interact with the environment (the controlled system), thus all input/output capabilities. It also supports all the applications required for the correct behaviour of the system.

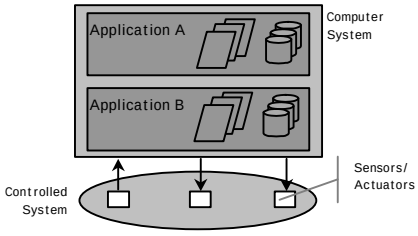

Figure 1.3. Centralised computer system

However, there are many advantages in using a distributed system instead of a centralised one. In such distributed architecture (Figure 1.4), a broadcast communication network is used as a replacement for the point-to-point links (between the sensors/actuators and the computer system). Additionally, application processing is no longer performed by a single computer unit, but by several units (nodes), interconnected via the same network.

By providing a distributed architecture to the development of computer systems, the overall throughput of the system can be increased, as more processing power is available for the applications. Furthermore, specialised hardware (*e.g.* specific sensor/actuator interfaces, human-machine interfaces, etc) can be used to interface the real-time computer system with the environment, while general-purpose nodes can be used for the processing activities.

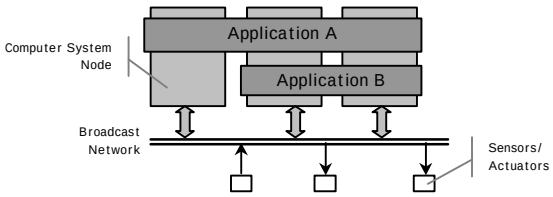

Figure 1.4. Distributed computer system

By using a distributed environment, it is also possible to provide redundancy to the computer control application, increasing the reliability of the system, since application components can be replicated and thus single points of failure can be avoided in the system (Figure 1.5). Using this approach, the broadcast network is also responsible for supporting the communication mechanisms related to the redundancy management.

Centralised computer control systems can also benefit from distributed architectures, since redundancy can be provided to the computer control application. One or more nodes can execute replicas of all (or some) of the tasks of the original node. Either all or some of the nodes are responsible for the interaction with the controlled system.

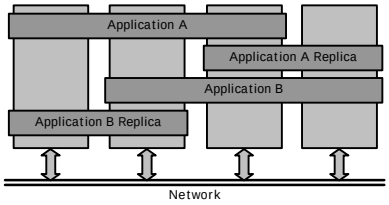

Figure 1.5. Redundancy in a distributed system

Whichever the means to achieve redundancy, a real-time communication network must be used to provide communication between the nodes, in order that consensus between them can be achieved within an upper-bounded time interval. The Controller Area Network (CAN) (ISO, 1993) is a field-level broadcast network, suitable for computer control systems. Studies are available on how to guarantee real-time requirements of CAN messages, thus providing pre-run-time schedulability conditions to guarantee the real-time requirements of the system. However, temporary network inaccessibility (Rufino and Veríssimo, 1995) and inconsistent message delivery (Rufino *et al.*, 1998) present impairments to guarantee fault-tolerant communication. Therefore, communication services providing fault tolerance properties must be developed, to guarantee that replicated components of the application observe the same consistent state of the controlled system, whilst guaranteeing predictability of message transfers.

Concerning the development of computer control applications, it is necessary to consider both the real-time and fault tolerance requirements of the controlled system, and, at the same time, the complexity of the mechanisms required to support replication and distribution. Such applications would become easier to develop and maintain if a

transparent and generic development approach were provided, hiding from the application the details of replication and distribution.

The use of the pre-emptive fixed priority computational model increases the complexity associated to the development of computer control applications. Traditionally, a scheduling table comprising the sequence of task executions is determined off-line (cyclic-executive model). Therefore, the programming model for the development of these applications is simpler, not requiring complex task interaction mechanisms, because it fixes the sequence of tasks' executions. By using the pre-emptive fixed priority model of computation, the system flexibility is increased, and the design effort is decreased. However, mechanisms for task interaction become more complex, since there is no fixed pattern for the sequence of tasks' executions.

The Ada 95 (ISO/IEC, 1995) language allows the development of applications using the pre-emptive fixed priority model, and is widely used in the domain of fault-tolerant real-time systems. Nevertheless, its multitasking mechanisms are rarely used, since they are considered to be too complex to be analysable and introduce inefficiencies and non-determinism in the supported applications. The Ada 95 Ravenscar profile (Burns, 1997) defines a subset of the language's multitasking mechanisms, considered suitable for the development of efficient and deterministic real-time applications. It allows multitasking pre-emptive applications to be considered for the development of fault-tolerant real-time systems, whilst providing efficient and deterministic applications. Nevertheless, it is considered that further studies are necessary for its use in replicated and distributed systems (Wellings, 2000). The interaction between multitasking pre-emptive software and replication introduces new problems that must be considered, particularly for the case of a transparent and generic approach.

1.3. Research Objective

Considering the presented context, it is important to provide a generic and transparent programming model for the development of computer control applications. The goal is to decrease the complexity of application development, by precluding the need for the simultaneous consideration of system requirements and interaction between multitasking and replication/distribution. This programming model must be supported by the appropriate communication mechanisms, guaranteeing that messages are consistently delivered to the application replicas, and also that replicated outputs are consolidated according to pre-defined rules.

Therefore, the main objective of this thesis is to propose such a programming model for the development of pre-emptive multitasking applications on top of Commercial Off-The-Shelf components. The central proposition of this thesis is that *pre-emptive fixed priority applications can be built using a generic and transparent programming model, without having to simultaneously consider the system requirements and the interaction between multitasking and replication/distribution issues*.

This can be accomplished by means of providing a transparent support for the replication of software components, allowing Ravenscar applications with different structures and configurations to be developed, and considering a close integration between the programming mechanisms and the underlying communication infrastructure.

1.4. Research Contributions

Considering the above mentioned research objective, the main contributions of this thesis are:

- *A transparent framework for the development of replicated Ravenscar applications.*
  This thesis proposes a generic and transparent framework for the development of replicated software components (Pinho and Vasques, 2000), based on the use of generic inter-task interaction objects (Pinho *et al*., 2001a; Pinho *et al*., 2001b). The use of these objects allows applications to be developed without considering replication and distribution issues in the programming phase. Afterwards, these generic objects can be instantiated with application-specific configuration issues, only introducing the replication and distribution mechanisms in a later configuration phase.
- *Fault-tolerant real-time communication mechanisms in CAN networks*.
  In this thesis the communication support for the replication of the software components is also considered. Therefore, this thesis proposes a set of atomic multicast and consolidation protocols for CAN networks (Pinho *et al*., 2000b; Pinho and Vasques, 2001b; Pinho and Vasques, 2001d). In order to guarantee the real-time behaviour of the supported applications, a set of pre-run-time schedulability conditions is devised (Pinho and Vasques, 2001a; Pinho and Vasques, 2001c), enabling the off-line timing analysis of the network, even in the presence of errors (either caused by the bus or the nodes' network interface) (Pinho *et al*., 2000a).

The prototype implementation of the proposed framework is also described. This prototype was used to assess the expressiveness of the Ravenscar profile for the development of fault-tolerant real-time systems, considering the proposed generic and transparent approach.

1.5. Thesis Organisation

This thesis is structured as follows. Chapter 2 provides an overview of the system architecture used as the support for the development of the replication management and communication mechanisms. Some basic definitions are presented, together with the fault tolerance and real-time requirements imposed by the targeted application domains.

Chapter 3 presents a survey of relevant related work in the field of fault-tolerant real-time systems. It presents an overview of relevant system architectures developed to support these systems, and also some previous relevant work in the areas of software-based fault tolerance and real-time schedulability analysis. This chapter also provides a survey of the technologies addressed in this thesis, namely the Ravenscar profile of the Ada 95 language and the Controller Area Network.

Chapters 4 and 5 present the main research contributions of this thesis. Chapter 4 proposes a framework supporting the replication of software components. It explains how replication is achieved in applications, and how these applications can be configured to address their real-time and fault tolerance requirements. Afterwards, the

repository of generic objects for task interaction available to applications is presented, together with the underlying software layer, intended to support these objects.

Chapter 5 presents how fault-tolerant real-time communication in CAN networks is achieved. The problem of inconsistency in CAN message deliveries is addressed through a set of atomic multicast and consolidate protocols for fault-tolerant real-time communication in CAN. A set of pre-run-time schedulability conditions is also presented, enabling the timing analysis of the supported real-time communication streams.

Finally, Chapter 6 presents the prototype implementation of the proposed framework, based on the mechanisms and protocols proposed in Chapters 4 and 5. The goal of this implementation was the assessment of the expressiveness of the Ravenscar profile for the implementation of the proposed approach.

This thesis concludes with Chapter 7, which summarises the presented contributions and identifies topics for further research. An Annex is also provided, presenting a study addressing the behaviour of CAN networks in the presence of either bus or nodes' network interface errors.

Chapter 2

Definition of the System Architecture

2.1. Introduction

The main purpose of the DEAR-COTS (Distributed Embedded ARchitecture using Commercial Off-The-Shelf components) project[1] is the specification of an architecture based on the use of COTS components, intended for the development of computer control systems. The project addresses several issues, at the communication and programming levels, such as: the impact of real-time and fault tolerance requirements on the communication architecture, distributed fault-tolerant concurrent applications and the real-time support environment.

The generic DEAR-COTS architecture (Veríssimo *et al*., 2000b), allows the integration in the same system of applications with different real-time and fault tolerance requirements, whilst guaranteeing the requirements imposed by the more stringent applications.

The research presented in this thesis was performed within the DEAR-COTS Hard Real-Time Subsystem (Pinho and Vasques, 2000). The goal was to provide DEAR-COTS with a generic and transparent framework, intended for the development of fault-tolerant real-time applications.

This chapter presents the DEAR-COTS architecture, and how it can be used to develop fault-tolerant real-time applications. The remainder of the chapter is structured as follows. Section 2.2 provides the basic concepts and definitions in the areas of real-time and fault tolerance, which are of relevance for the definition of the system architecture.

Section 2.3 presents some of the requirements commonly found in computer control systems, which were considered in the development of the DEAR-COTS architecture. The architecture itself is presented in Section 2.4. Finally, Section 2.5 presents the main guidelines for the development of fault-tolerant real-time applications, using the DEAR-COTS Hard Real-Time Subsystem.

---

[1] Project DEAR-COTS (Leader: Paulo Veríssimo) is funded by the FCT as project PRAXIS/P/EEI/14187/1998. The project members are: the University of Lisbon, the University of Porto, the Polytechnic Institute of Porto, and the Technical University of Lisbon.

2.2. Definitions

2.2.1. Real-Time Definitions

When the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced, the system is classified as a real-time system (Stankovic, 1988). Such kind of systems must process inputs from the environment and provide the adequate outputs within an upper-bounded time interval (relative deadline), which is dictated by the requirements of the controlled system.

It is therefore necessary to analyse its timing behaviour, comparing the worst-case response time (WCRT) of the application tasks with the relative deadline required by the controlled system. The worst-case response time of an application task is defined as the time interval between the arrival of a request from the controlled system and the completion of the required processing (Joseph and Pandya, 1986). The relative deadline can be defined as the maximum time interval between the arrival of the request and the completion of the related processing. It is obvious that, in order to guarantee the timing requirements of the controlled system, the worst-case response time of the application tasks must be smaller or equal to the associated relative deadline. In this case, the system is considered to be schedulable. Schedulability analysis is thus defined as the process to assess if the responsiveness of the system is sufficient to guarantee the required timing bounds.

Applications tasks may have different types of timing requirements, depending on the consequences of not being completed before their deadlines (Burns, 1991). When the benefit of the action to be performed by the task is zero or negative if it is performed after the deadline, the task is defined as hard real-time. If missing the deadline does not imply compromising the integrity of the system, the task can be defined as a soft real-time task. It is important to note that applications may have a set of tasks with different timing requirements. Nevertheless, if the application contains at least one hard real-time task, the application is defined as being a hard real-time application.

2.2.2. Fault Tolerance Definitions

Fault tolerance is defined as the ability of a system to provide a service complying with the specification in spite of faults (Laprie, 1992). It is one of the means to achieve dependability in computer systems, that is, the property of a computer system such that reliance can justifiably be placed on the service it delivers.

A fault can be defined as a potential source of system malfunction. It can be caused by some external interference with the system (*e.g.* electro-magnetic interference), or it can exist in the system itself (*e.g.* a design fault in the application software). A fault by itself may not produce an incorrect behaviour of the system, since it may remain silent. An error only occurs when the effect of a fault is observed. If the error propagates through the boundaries of the system, it causes a failure. These notions are not self-contained, since a failure in a component produces a fault of the system that contains the component, or in other components that interact with it (Laprie, 1992).

Therefore, the objective of providing fault tolerance is to preclude the failure of the system in the presence of faults, caused by the failure of one of its components.

The simplest assumption that can be made on the failure modes of a component is that it only fails by stopping to produce results (Powell *et al.*, 1988). In such case, the component is assumed to be fail-silent. The less restrictive assumption is that a component can exhibit arbitrary (or Byzantine) failures. In this case, the component is assumed to be fail-uncontrolled, and can fail by not producing any result, by producing an incorrect result, by producing a result too early or too late, or by unexpectedly producing a result.

One of the approaches to guarantee the fault tolerance requirements of computer control applications is by error compensation. This is achieved by providing some form of redundancy, replicating some of the system components, in order to detect errors by some form of voting between replicas. A usual example is Triple Modular Redundancy (TMR), where a component is constituted by three replicas, and the output of the component is the result of the comparison of the individual replicas' outputs. However, this approach does not provide tolerance to software design faults, which must be addressed by means of design diversity.

2.3. Requirements

2.3.1. Real-Time Requirements

Since there is the need to guarantee that the timing requirements of the controlled system are met, it is necessary to analyse the response time of the application tasks, in order to compare them with the defined set of timing requirements. In hard real-time computer control applications it is necessary to guarantee *a priori* that deadlines are met. Therefore, some sort of off-line analysis must be performed, to determine the worst-case response time of application tasks and to make the comparison with the system imposed deadlines, guaranteeing the schedulability of the task set before execution. This pre-run-time schedulability analysis requires *a priori* knowledge of the tasks' characteristics, which fortunately is possible in most of computer control applications.

In the controlled system, different devices may require different application behaviours. Some devices require specific time intervals to be kept between consecutive sampling, while others may sporadically require immediate processing. Consequently, applications are required to support tasks with different behaviours: periodic or sporadic. A periodic task is cyclically released with a defined time interval between release requests, while a sporadic task is released in response to some change of state in the environment.

A periodic task is characterised by its period (time interval between consecutive arrivals), its worst-case execution time (maximum time to execute the task program per period, without considering the existence of other tasks) and its relative deadline (maximum duration for the response to be performed, related to the instant of the initial release request). For the case of a sporadic task, since their release is usually requested

by some event, the minimum time interval between requests is used instead of the period.

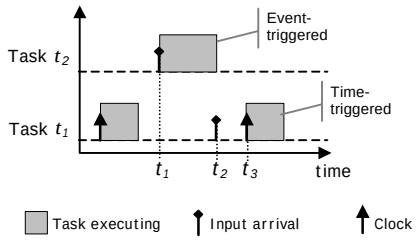

Figure 2.1. Time-triggered vs. event-triggered

Therefore, it is necessary to support both the event-triggered and the time-triggered model for task activation. In the event-triggered model, tasks are released when a state change of the system is detected (sporadic tasks). Conversely, time-triggered tasks are initiated at predefined points in time (periodic tasks). By allowing the simultaneous execution of both sporadic and periodic tasks, applications are not restricted to just a specific model. For instance, in Figure 2.1, task t_1 follows the time-triggered model, since although the external event occurs at instant t_2, the task only becomes ready for execution by the passage of time (clock) at instant t_3. On the other hand, task t_2 follows the event-triggered model, as it becomes ready for execution when the event occurs.

Since it is expected that tasks often need to share information and to synchronise, the application is also required to allow tasks to interact with each other. Thus, it is necessary to provide mechanisms allowing tasks to interact without compromising the integrity of the data.

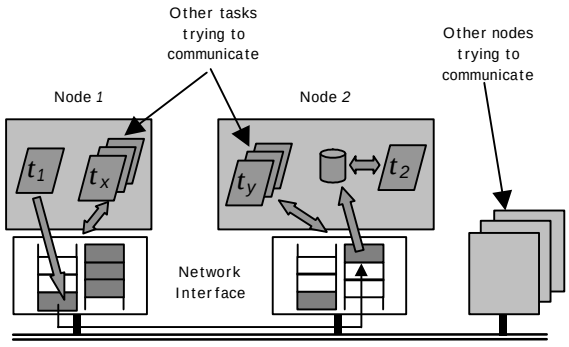

Figure 2.2. Distributed interaction

Additionally, as real-time applications can also be distributed over the system nodes, there will be real-time tasks interacting through the network (Figure 2.2). It is obvious that, in order to guarantee the real-time requirements of the application, a network with real-time characteristics must also be used. Furthermore, the response time of the

messages depends not only on the scheduling of tasks in its node, but also on the scheduling of messages in the network. As there could be several messages queued to be transferred, the response time analysis of the task set must consider the interference of the overall message scheduling.

In Figure 2.2, when task t_1 in node *1* sends a message to task t_2 in node *2*, the transfer of this message will suffer interference from other messages in the network, sent by other tasks in the system. Moreover, also the scheduling of task t_2 may suffer interference from the scheduling of messages in the network, if its release is dependent on the arrival of the message. Therefore, in order to allow the schedulability analysis of this distributed model the schedulability analysis of the communication network must be integrated with the schedulability analysis of the processing tasks.

2.3.2. Fault Tolerance Requirements

In computer control applications, unexpected failures of the system must be avoided, since value or timing requirements would not be met. It is clear that applications must rely on specific mechanisms to tolerate faults in its components, precluding the failure of the application. These mechanisms must allow applications to tailor their behaviour in the presence of faulty components. The controlled system may allow applications to be designed in order to provide a fail-safe behaviour, thus to correctly and gracefully shutdown when the required level of fault tolerance can no longer be provided. Or it may require that a functioning system is maintained (even if in a degraded mode), until the faulty components can be repaired (or replaced) to restore the required system capabilities.

This implies that it is necessary to provide continuous and adequate service to the controlled system, in order to increase the confidence level put in the controlling system. The use of COTS components presents new difficulties, since it generally implies fail-uncontrolled components, as they usually do not have the required self-checking mechanisms (Powell, 1994). The assumption of fail-silent components simplifies the implementation of the fault tolerance mechanisms, however, achieving fail-silent behaviour is only possible with the use of self-checking techniques, increasing the system cost and complexity. So, software-based fault tolerance mechanisms must also address components with fail-uncontrolled behaviour.

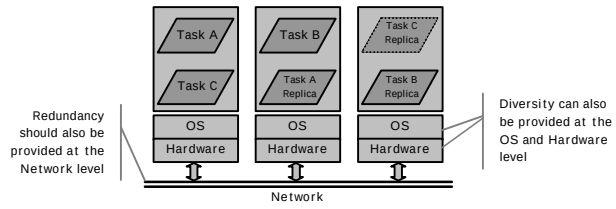

Figure 2.3. Redundancy with dissimilar task sets in a distributed system

These requirements can be guaranteed by providing redundancy through the replication of application software on different nodes within a redundant distributed system (Figure 2.3). Consequently, this requires support for replication of application

components, with the consistent dissemination of data to replicated components. It is necessary to manage the replicated components and to provide the appropriate communication protocols with consistent multicast of information and consolidation of replicated components. Fail-uncontrolled components require the use of active replication (Powell, 1994), since masking the failure of a component requires the replication of such component in other nodes, with some form of consolidation between the components' outputs, in order to give the illusion of a single component. Consequently, the computer system must be able to manage this component replication, guaranteeing that whenever a component fails, appropriate actions are performed in order to preclude the failure of the system.

A broadcast network must be provided for interconnection with the controlled system and between the processing nodes. As the network is a single point of failure, it should also be replicated. By distributing the system elements, tolerance to temporary and permanent external faults can be provided, due to some geographic distribution of the system. In order to tolerate common mode faults in the system, diversity in the COTS components may also be required (operating system and hardware platform).

Dissimilar replicated task sets can be provided in each node, thus providing different execution environments, tolerating temporary design faults (Powell, 1994). Furthermore, it also increases the system flexibility, as nodes are not just copies of each other, allowing for a more flexible design of real-time applications. Note that this approach embodies both distribution motivated fault tolerance (implementing fault tolerance in a distributed environment) and fault tolerance motivated distribution (implementing distribution to achieve fault tolerance), approaches that although similar present different requirements (Powell, 1994).

However, when replicated components are provided, it is necessary to guarantee the consistency of all replicas, that is, replicated components behave as a single fault-free component. It must be guaranteed that all replicas work with the same input values and that they all vote on the final output. Moreover, the different processing speed in replicated nodes can cause different replicas to respond to the same inputs in different order, providing inconsistent results if inputs are non-commutative. It is therefore necessary that replicas present a deterministic behaviour (Poledna, 1994).

There are a number of aspects related with fault tolerance that may interfere with the real-time performance of the system. By providing replicated software components, it is necessary to include replication management and fault-tolerant communication in the timing analysis models. It is also necessary to consider the intervals of time in which nodes can be disconnected of the network due to temporary periods of error recovery.

When distribution is used, there is also the need for fault-tolerant and time-bounded communication services. Messages must be correctly and orderly delivered according to their timing requirements. Therefore, the full integration of the communication infrastructure with the application fault tolerance mechanisms is required, in order to obtain the desired level of confidence in the system.

2.3.3. Genericity and Transparency Requirements

Although with similar requirements, computer control applications have several different structures and configurations. Thus, any solution for building fault-tolerant real-time

applications must be generic, in order to allow the development of these different kinds of applications. It is essential to allow tailoring applications to meet their specific real-time and fault tolerance requirements.

As presented, the targeted applications may require different type of computational models, encompassing both time-triggered and event-triggered requirements. Moreover, as the introduction of replication in the system also introduces overheads, applications may want to provide lower degrees of replication to less critical components, in order to increase the system's efficiency. These applications must not be restricted to a particular configuration, since it is necessary to encompass different structures, ranging from a redundant centralised system to a completely redundant distributed system.

Since developing computer control applications on top of COTS components is a difficult task (further complicated with the incorporation of the pre-emptive model), it is necessary to transparently deal with replication and distribution issues. Application development must abstract from the low-level implementation details of distribution and replication, focusing on the requirements of the controlled system.

Hence, a set of generic mechanisms must be provided, which can be parameterised with both application-specific data and application-specific configuration (distribution and replication). These mechanisms will be the basic building blocks of distributed/replicated real-time applications, providing a higher level of abstraction to developers and maximising the capability to reuse components and mechanisms in different applications.

2.3.4. Interconnectivity Requirements

Currently, computer control systems demand for more flexibility and interconnectivity capabilities, while guaranteeing the requirements of the supported real-time applications. The integration of hard real-time applications, whose requirements have to be guaranteed, with soft real-time applications, where a more flexible approach can be used, is also a current requirement in computer control applications. There is the need to integrate applications with less stringent fault tolerance and real-time requirements with the hard real-time computer control applications, in order to allow the interoperability of the computer control system with higher-level systems (*e.g.* supervision/management).

The interconnection mechanisms must be carefully designed, guaranteeing that failures in less critical components do not interfere with the guarantees provided to hard real-time applications. Thus, mechanisms for memory partitioning must be provided, and also the integrity of data transferred from the different applications must be guaranteed by appropriate inter-communication mechanisms.

2.4. The DEAR-COTS Architecture

A DEAR-COTS system (Figure 2.4) is built using distributed processing nodes, where distributed hard real-time and soft real-time applications may coexist. Each DEAR-COTS node can be constituted by several different subsystems, within which applications with different requirements will be executed. A DEAR-COTS node is

characterised by the subsystems it is composed of. There are essentially three basic node types: Hard real-time nodes (H), Soft real-time nodes (S) and Gateway nodes (H/S).

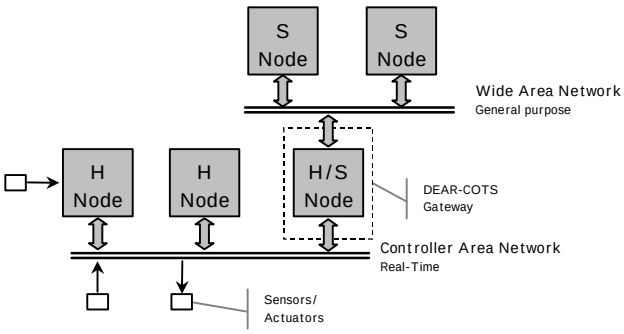

Figure 2.4. A generic DEAR-COTS system

Hard real-time nodes are those where only the Hard Real-Time Subsystem (HRTS) exists. Therefore, they will be exclusively used to support hard real-time applications, which are at the core of the computer control system. Soft real-time nodes only include the Soft Real-Time Subsystem (SRTS), providing the execution environment for the remote supervision and remote management of applications.

A Gateway node integrates both subsystems, with two distinct and well-defined execution environments. The idea is to allow hard real-time components, executing in the HRTS, to interact in a controlled manner with soft real-time components, executing in the SRTS.

In order to support distributed/replicated applications, a fault-tolerant and real-time communication infrastructure based on the Controller Area Network (CAN) (ISO, 1993) is provided to the set of H and H/S nodes. As there is the need to interconnect these nodes with the upper levels of the system (*e.g.* for remote access, remote supervision and/or remote management), there is a general-purpose network interconnecting H/S and S nodes.

In the DEAR-COTS architecture, the Timely Computing Base (TCB) model (Veríssimo *et al.*, 2000a) is used as a reference model to deal with the heterogeneity of system components and of the environment, with respect to timing properties. The TCB model deals with the problem of implementing applications with real-time requirements in environments that are unpredictable or unreliable.

This model requires systems to be constructed with a small control part, a TCB module, to protect resources with respect to timeliness and to provide basic time related services to applications. Applications can use the TCB to achieve different levels of timing guarantees, even in an environment with soft real-time behaviour as the Soft Real-Time Subsystem. Additionally, the supported hard real-time applications can use the TCB services to be aware of their timing behaviour, preserving the reliability of the system. The reasoning is that the TCB module is built as a small control module, therefore it can be built with greater coverage of failure assumptions (Veríssimo *et al.*, 2000a).

2.5. Fault-Tolerant Real-Time Applications in DEAR-COTS

In the set of H and H/S nodes, the Hard Real-Time Subsystem (HRTS) (Pinho and Vasques, 2000) is intended for the transparent distribution and replication of real-time applications. Since real-time guarantees must be provided, applications have guaranteed execution resources, including processing power, memory and communication infrastructure. This is the main reason for the need of a separated real-time communication network for the HRTS, where messages are transmitted and processed in a bounded time interval.

A multitasking environment is provided to support the real-time applications, with services for task communication and synchronisation (including replication and distribution support). Applications' timing requirements are guaranteed through the use of current off-line schedulability analysis techniques (*e.g.*, the well-known Response Time Analysis (Joseph and Pandya, 1986; Audsley *et al.*, 1993)).

To ensure the desired level of fault tolerance to the supported real-time applications, specific components of these applications may be replicated. This replication model supports the active replication of software (Figure 2.5) with dissimilar replicated task sets in each node. The goal is to tolerate faults in the COTS components underlying the application. In order to tolerate common mode faults in the system, COTS components diversity is also considered (operating system and hardware platform).

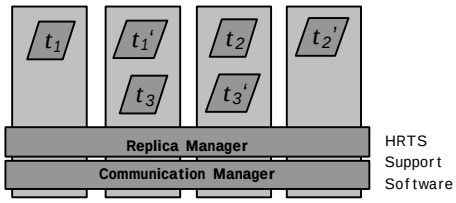

Figure 2.5. Replicated hard real-time application

However, using diverse operating systems has to be carefully considered, since in order to guarantee a transparent approach, the programming environment in each node must be the same. This can be achieved by using operating systems with a standard programming interface or by using a programming language that abstracts from the operating system details. DEAR-COTS considers the use of the Ada 95 (ISO/IEC, 1995) language in the replicated hard real-time applications, namely the Ravenscar profile (Burns, 1997), which is a restricted profile of the language tasking model suitable for the development of efficient and deterministic real-time applications. This solution provides the same programming model in all nodes, whilst diversity can be provided by using different compilers and runtimes (Yeh, 1995).

The DEAR-COTS architecture does not address tolerance to application design faults. Nevertheless, by providing different execution environments in each node, the tolerance to temporary design faults is increased. Temporary design faults can be tolerated due to the differences in the replicas' execution environment (Powell, 1994), since nodes are considered independent from the point of view of failures. Moreover, dissimilar replicated task sets in each node also increase the system flexibility, as nodes are not just copies of each other, allowing for a more flexible design of real-time applications.

The HRTS Support Software provides the distribution support (including both the application distribution itself and the replica management) to hard real-time applications. The goal of the Replica Manager layer is to support hard real-time application objects required for interaction between distributed and replicated tasks. The Communication Manager layer is responsible for the adequate communication services, providing a fault-tolerant and real-time transfer of data.

2.5.1. The Timely Computing Base in the HRTS

In the generic model of DEAR-COTS, the TCB can be used to guarantee the timing requirements of soft real-time applications, or to increase coverage of timing faults at the Hard Real-Time Subsystem (Figure 2.6). The different approaches can be combined in any form, in each particular instantiation of the architecture.

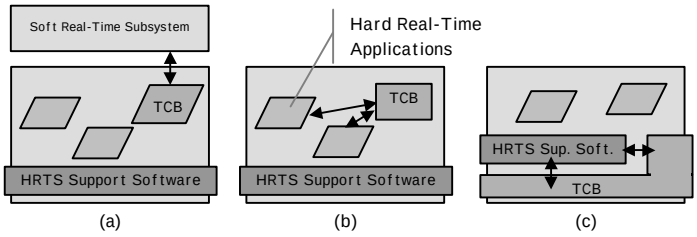

Figure 2.6. The Timely Computing Base in the HRTS

In the first approach (Figure 2.6a), the TCB is used as an additional hard real-time application, to deal with the timing requirements of soft real-time applications executing in the SRTS of the system. In this approach, the Support Software sees the TCB as any other hard real-time application.

The second approach (Figure 2.6b) is to use the TCB as a timing error detector at the hard real-time applications level. That is, tasks in the HRTS may use the services of a TCB to detect timing errors, thus increasing the failure assumption coverage of the application. In this approach, the TCB would serve as a second independent level of fault tolerance. However, to the Support Software, it would also appear as one hard real-time application.

The third approach (Figure 2.6c) is to use the TCB to increase the reliability of the Support Software itself. In this approach, Support Software tasks use the TCB to detect their own timing errors. This solution increases the system reliability, since it is possible to detect both Support Software tasks' overruns and incorrect communication requests.

2.5.2. Error Detection and Recovery

On the occurrence of faults, they will be masked by component replication, providing the required fault-tolerant behaviour. However, in the case of permanent (or intermittent) faults, the system can no longer provide the same level of fault tolerance as it was designed to. Therefore, recovery actions must be executed.

Three different approaches are considered in instantiated DEAR-COTS systems. First, the occurrence of faults may require that only the higher levels of the system are notified, and no action is taken. Second, it is possible to design applications in order to provide a fail-safe system, thus to correctly shutdown when the required level of fault tolerance can not be provided. Finally, applications may attempt to maintain a functioning system in a degraded mode, until the faulty components can be repaired (or replaced) to restore the required system capabilities.

Integrating new components in active systems is not an easy task, as these newly created components are in an "amnesia" state (Powell, 1994), *i.e.*, they have no knowledge of the system state. This implies that, prior to their activation, their state must be brought to be consistent with the replicas that continued execution. However, this state is very application specific, meaning that this transfer cannot be easily performed in a generic or transparent approach (Powell, 1991; Rushby, 1996; Bondavalli e*t al.*, 1998). Moreover, the efficient transfer of this internal state is highly dependent of the properties of applications, namely data-flow dependencies among tasks and the way that internal task state data is replaced by data external to the task (Rushby, 1996).

Error recovery and state restoration is still an open issue in the DEAR-COTS architecture. It is considered that an efficient and reliable mechanism can only be obtained with some knowledge of the semantics and data-flow properties of applications, therefore cannot be provided by the Support Software itself and application level mechanisms must be used (Figure 2.7). Nonetheless, it is considered that error detection and recovery actions must be supported.

Consequently, the Support Software of the HRTS provides mechanisms for applications to be notified of error detection (in order to take actions to recover from them) and component's shutdown or silence mechanisms can be used to prevent components that are performing incorrectly to contaminate the application.

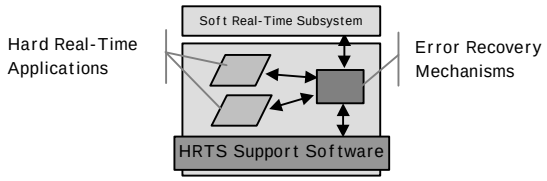

Figure 2.7. Error recovery mechanisms

Error detection in the HRTS may take two complementary forms: by detection of errors in the communication protocols and in the consolidation of replicated values by the Support Software, or by the use of the TCB. The Support Software may also be used to notify applications and/or to disseminate the error detection through the real-time network. The TCB can be used to detect the occurrence of timing errors in the HRTS (using approach *c)* of Figure 2.6), in order, for instance, to silence the node so that the network is not contaminated (Veríssimo *et al.*, 2000a).

2.5.3. Relationship with the Research Objectives

The goal of this thesis is to propose a generic and transparent programming model for the development of pre-emptive multitasking applications on top of Commercial Off-The-Shelf components. The DEAR-COTS Hard Real-Time Subsystem (HRTS) is considered to be suitable for the development of such programming model. Therefore, this goal is accomplished by providing the HRTS with:

- *A transparent framework for the development of replicated Ravenscar applications*.
  In the HRTS, replicated Ravenscar applications are supported through the active replication of their components. It is therefore necessary to provide a transparent programming model, abstracting the development of applications from the distribution/replication details, focusing on the requirements of the controlled system. This programming model must be generic, in order to allow applications with different structures and configurations to be developed.
- *Fault-tolerant real-time communication mechanisms in CAN networks*.
  In the HRTS, a CAN network is provided as the communication infrastructure, both for the interconnection with the controlled system and for the management of replicated application components. This communication infrastructure must guarantee the consistent state of replicated applications, while at the same time preserving CAN real-time characteristics. This infrastructure must provide the appropriate atomic multicast and consolidation protocols, in order to guarantee the fault tolerance and real-time properties of message streams.

2.6. Summary

This chapter presented an overview of the system architecture used to support the research presented in this thesis. Basic definitions of the real-time and fault tolerance issues addressed in this thesis are presented, in order to allow a better comprehension of the system architecture. The requirements that were considered in the development of the architecture are then presented. In addition to the real-time and fault tolerance requirements, the issues of genericity, transparency and interconnectivity are also considered, as a more flexible programming environment must be provided for the development of current applications.

The DEAR-COTS architecture is then briefly presented, focusing on how it is suitable to support the development of fault-tolerant real-time applications. In DEAR-COTS, the Hard Real-Time Subsystem is intended for the development of fault-tolerant real-time applications. Within this subsystem a replication model is provided, supporting the active replication of software with dissimilar replicated task sets in each node.

Finally, the relationship between the DEAR-COTS architecture and the research objective of this thesis is emphasised, clarifying how the proposed framework is accomplished within the DEAR-COTS Hard Real-Time Subsystem.

Chapter 3

Analysis of Previous Relevant Work

3.1. Introduction

The DEAR-COTS Hard Real-Time Subsystem (HRTS) is intended for the development of fault-tolerant real-time applications in a COTS-based architecture. It is thus essential to address the problems common to the development of such systems, and consider the existent methodologies and mechanisms for providing fault tolerance and real-time properties to computer control applications. Moreover, as the HRTS is based on replicating Ravenscar applications on top of a CAN network, it is also necessary to survey these technologies, and to study the impairments to their use.

The remaining of the chapter is structured as follows. Section 3.2 presents a survey of fault-tolerant real-time systems, presenting relevant architectures that provide non application-specific environments for the design of fault-tolerant real-time applications. A special focus is given to software-based fault tolerance techniques, since they are essential for providing fault tolerance to COTS-based architectures. Replica determinism is also given a special emphasis, since real-time applications inherently lead to timing non-determinism.

Afterwards, Section 3.3 presents relevant approaches for the schedulability analysis of real-time applications. The use of the response time analysis technique allows determining the worst-case response time of application tasks in the pre-emptive fixed priority computational model, thus providing a means to determine if the deadlines imposed by the controlled system can be guaranteed.

Section 3.4 presents an overview of the Controller Area Network (CAN) (ISO, 1993), which is used as the communication infrastructure for the replication and distribution of fault-tolerant real-time applications in the DEAR-COTS HRTS. The focus of the overview will be on the real-time behaviour of CAN message transfers, and on impairments that CAN presents for fault-tolerant communication. The problem of inconsistency in CAN message transfers is presented, and it is discussed how this problem precludes CAN from being used in a replication environment, without providing further mechanisms.

Finally, Section 3.5 provides a brief description of the Ada 95 language (ISO/IEC, 1995), with a special focus on its use to build fault-tolerant real-time applications. A small overview of its concurrency model is provided, being the main focus given to the Ravenscar profile (Burns, 1997), which is a restricted profile of the language suitable for the development of efficient and deterministic real-time applications.

3.2. Fault-Tolerant Real-Time Systems

A considerable research effort has been devoted to the design and validation of fault-tolerant real-time systems. The most significant examples are Delta-4 (Powell, 1991), MARS (Kopetz *et al.*, 1989) and GUARDS (Powell, 2001), as these architectures intend to provide non application-specific environments to the design of fault-tolerant real-time applications. However, the integration of COTS components with the required fault tolerance and real-time properties is considered difficult, since the efficient implementation of fault-tolerant communication and replication management mechanisms is not supported by most COTS components.

The Delta-4 project aimed to develop an open, dependable architecture for large distributed real-time systems. In Delta-4, nodes are split in two different subsystems: the host, which is a COTS component, and the Network Attachment Controller (NAC), which is a fail-silent component making use of specialised self-checking hardware. The need to target systems with more stringent timing requirements led to the specification of the eXtended Performance Architecture (XPA) (Barrett *et al*., 1990). XPA systems are constituted by a set of distributed homogeneous nodes, connected by a LAN network with real-time properties, where the host is also considered to be fail-silent. However, contrarily to the NAC, the fail-silent behaviour of hosts is achieved through the use of soft fail-silent techniques, where fail-silence behaviour is achieved through software management of replicated processors.

MARS is a fault-tolerant distributed real-time system intended to support process control applications. The architecture consists of one or more clusters, which are distributed systems composed of single board computers, called components, connected by a real-time network. All the components maintain a global time base, allowing them to synchronise their actions and to use a time-triggered approach. In MARS, both node and network schedules are determined off-line and stored in a static schedule table. Components are devised as fail-silent, through the use of self-checking hardware, running in dual active redundancy, and of two redundant real-time networks where messages are sent in duplicate. There was no intention to use COTS components, which consequently led to a very specialised and costly architecture.

The GUARDS project intended to develop a generic architecture, substantially based on the use of COTS components, in order to minimise the development time and costs associated with critical real-time applications. The architecture is based on software-based fault tolerance mechanisms, in order to cope with the unreliability in the underlying COTS components. This difficulty to provide fault tolerance mechanisms led to the development of a two level replication approach. The architecture is constituted by a set of channels, each one containing replicated hosts interconnected by a shared memory scheme. These channels are interconnected by the Interchannel Communication Network (ICN), which is based on unidirectional serial links interconnecting channels. Therefore, it is difficult to use this interconnection scheme when more than a few channels are involved. Moreover, the ICN has to be scheduled in a static off-line table-driven approach, leading to an increased burden in the analysis and to the difficulty of changes in the application design. Nevertheless, since channel replication is only motivated by fault tolerance, it is not foreseen the need for systems with more than 3 or 4 channels. Furthermore, this architecture is targeted to safety- or mission-critical systems

(in the domains of railway, nuclear and space applications), that require a greater level of dependability and a more restrictive set of failure assumptions (Laprie, 1992).

3.2.1. Software-Based Fault Tolerance

Fundamental for a COTS-based fault-tolerant architecture is the issue of software-based fault tolerance. As previously discussed, there is no specialised hardware with self-checking properties, thus it is up to the software to manage replication and fault tolerance. The group abstraction can be used to implement replica management (Guerraoui and Schiper, 1997). In this approach, a fault-tolerant service is implemented by co-ordinating a group of replicated software components (Figure 3.1). The idea is to manage the group in order to mask failures of some of its members. Inter-replica co-ordination gives the illusion to other software components that the group is a single (fault-free) software component (Powell, 1991).

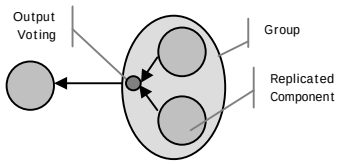

Figure 3.1. Active replication of software components

Three main replication approaches are addressed in the literature: active replication (presented in Figure 3.1), primary-backup (passive) replication and semi-active replication (Powell, 1991). In active replication, all replicas process the same inputs, keeping their internal state synchronised and voting all on the same outputs. In the primary-backup approach only one replica (the primary) is responsible for processing the inputs. In the semi-active replication, one of the replicas (the leader) co-ordinates the non-deterministic decisions. If fail-silent replicas are assumed, then any of the three approaches can be used. Otherwise, in the absence of the fail-silent assumption, incorrect service delivery can only be detected by active replication, because it is required that all replicas output some value, in order to perform some form of voting. Therefore, active replication is the most adequate technique when fail-uncontrolled components are considered (Powell, 1991). The use of COTS components generally implies fail-uncontrolled replicas (as these components usually do not have the required self-checking mechanisms), so it becomes necessary to use active replication techniques.

As real-time applications are based on time-dependent mechanisms, the different processing speed in replicated nodes can cause different task interleaving. Consequently, different replicas (even if correct) can process the same inputs in different order, providing inconsistent results if inputs are non-commutative. That is the problem of replica determinism in distributed real-time systems (Poledna, 1994).

For instance, in Figure 3.2, task t_1 is specified to send a message to task t_2. As both tasks are replicated in nodes *1* and *2*, task t_1 in node *1* will send a message to task t_2 in node *1*, while task t_1 in node *2* will send the message to task t_2 in node *2*. However, a slightly different execution pattern (*e.g.* caused by a small clock difference) causes task

t_1 to be slightly delayed in node *2*, and at instant t_2 (the arrival of the input that releases task t_2) task t_1 is pre-empted before being able to send the message. As in node *1* task t_1 has already sent the message (instant t_1), while in node *2* task t_1 will only send the message at instant t_4, when task t_2 receives the message (instant t_3) in node *2* it will receive a previous version of the message (or no message at all). Thus, even with both tasks executing correctly, replicated tasks t_2 will no longer be consistent.

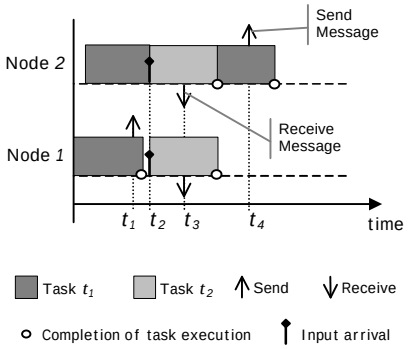

Figure 3.2. Timing non-determinism

Determinism can be achieved by forbidding the applications to use non-deterministic timing mechanisms. As a consequence, the use of multitasking would not be possible, since task synchronisation and communication mechanisms inherently lead to timing non-determinism. This is the approach taken by both MARS and Delta-4. The former by using a static time-driven scheduling that guarantees the same execution behaviour in every replica. The latter by restricting replicas to behave as state-machines (Schneider, 1990) when active replication is used.

Guaranteeing that replicas take the same scheduling decisions by performing an agreement in every scheduling decision, allows for the use of non-deterministic mechanisms. This imposes the modification of the underlying scheduling mechanisms, leading to a huge overhead in the system since agreement decisions must be made at every dispatching point. This is the approach followed by previous systems, such as SIFT (Melliar-Smith and Schwartz, 1982) or MAFT (Keickhafer *et al.*, 1988), both architectures for fault-tolerant real–time systems with restricted tasking models. However, the former incurred overheads up to 80% (Pradhan, 1996), while the latter was supported by dedicated replication management hardware (Keickhafer *et al.*, 1988).

3.2.2. The Timed Messages Concept

The use of the timed messages concept, independently developed in (Barrett *et al.*, 1995) and in (Poledna, 1998) and then integrated in (Poledna *et al.*, 2000), allows a restricted model of multitasking to be used, while at the same time minimises the need for agreement mechanisms. This approach is based on preventing replicated tasks from using different inputs, by delaying the use of a message until it can be proven (using global knowledge) that such message is available to all replicas.

This is the approach used in the GUARDS architecture in order to guarantee the deterministic behaviour of replicated real-time transactions (Wellings *et al.*, 1998). However, in the GUARDS approach this mechanism is explicitly used in the application design and implementation, thus forcing system developers to simultaneously deal with both system requirements and replication issues.

Timed messages are based on the global knowledge of the release time (the instant when the task becomes ready for execution) and the worst-case response times of tasks (if approximately synchronised clocks are used). Therefore, it is possible, using this global knowledge, for tasks to read the latest version of a value that it is known to be available in all replicas.

In this approach, messages are associated with a validity time. This validity time is defined as the instant where the message value becomes valid[2]. A value becomes valid when it is known that all replicas of the writer task have already written the value. Such validity time is defined as (Poledna *et al.*, 2000):

$$m_k(v) = \begin{cases} W & \text{intra-node messages} \\ W + \Delta + e & \text{inter-node messages} \end{cases} \tag{3.1}$$

where $m_k(v)$ is the validity time of message m_k, W is the maximum worst-case response time of all replicated writer tasks, D is the worst-case delivery time of the message and e is the maximum clock difference in the system. Note that all these values are known prior to execution.

In order to guarantee that replicated tasks read the same value, it is necessary to store several versions of the same message. Reader tasks must read the version that has the maximum validity time older than the task release time (Poledna *et al.*, 2000):

$$m_k.receive(tr_j) = \max_{i=0}^{n}(m_k : m_k[i](v) \le tr_j) \tag{3.2}$$

where tr_j is the release time of the reader task and n is the number of different versions received of message m_k. The release time of a task is not known prior to execution, but can be globally known in the system. For a periodic task, its release time is the same in all the replicas. For sporadic tasks released by other tasks, its release time can be determined as:

$$tr_j = tr_i + B_i \tag{3.3}$$

where tr_i and B_i are the release time and the best-case execution time of the releasing task, respectively. Sporadic tasks that are released by external events must also have a common release time. This common time must be agreed upon all the replicas.

Figure 3.3 presents the same example as in Figure 3.2, but using the timed messages concept. As it can be seen, since message versions have an associated valid time, when replicated tasks t_2 receive the message they will both receive version *[n-1]* of the message, thus will remain consistent. Version *[n]* of the message will only become valid after a delay equal to the worst-case response time of task t_1 ($W(t_1)$).

[2] Note that this is not the usual meaning of validity. In this case it is a *not to use before* rather than a *not to use after*.

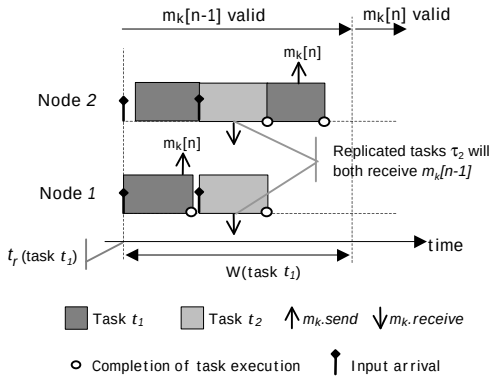

Figure 3.3. Execution of timed messages

As it is necessary to store several versions of the same message, there is the need to determine off-line the required number of versions to keep. This number depends on the relative ratio of reader tasks' response time and writer tasks' periods. In (Poledna *et al.*, 2000) the *non-deterministic send rate* (*NDSR*(m_k,t_i)) of message m_k is defined as the number of versions of m_k that become valid during the worst-case response time of reader task t_i:

$$NDSR(m_k, t_i) = \sum_{t_j \in TS(m_k)} \left(\left\lceil \frac{W_i}{P_j} \right\rceil + 1 \right) \tag{3.4}$$

where $TS(m_k)$ is the set of tasks that send message m_k, and P_j is the period of task t_j. Then, the number of message versions that must be available ($MVersions(m_k)$) can be determined as:

$$MVersions(m_k) = \max_{t_i \in TR(m_k)} NDSR(m_k, t_i) \tag{3.5}$$

where $TR(m_k)$ is the set of all tasks that read message m_k. Therefore, the maximum number of message versions that must be available can be upper bounded, since it is known which are the tasks that read or write each message, and which are their periods and their worst-case response times.

Poledna *et al.* (Poledna *et al.*, 2000) also provides optimisations to these equations, taking advantage of worst- and best-case response times of internal computations of a task. However, for clarity reasons only the simpler optimisations were presented here. The analysis can easily be extended to incorporate those optimisations if the required values are available in the application.

3.2.3. Replication Support

Even using the timed messages concept for replica determinism, the existence of replication and/or distribution implies that further mechanisms must be implemented in

order to support replica consistency. The active replication technique implies that replicas must process the same set of inputs, in the same order (Guerraoui and Schiper, 1997). Furthermore, replication management must support some form of output consolidation, in order to give the illusion to other software components that the group of replicas is a single component (Powell, 1991).

Therefore, active replication requires the communication infrastructure to provide atomic multicast protocols and mechanisms to consolidate replicated components. An atomic multicast has the following properties (Hadzilacos and Toueg, 1993):

- Validity: If a correct component broadcasts a message *m*, then all correct components deliver *m*.
- Agreement: If a correct component delivers a message *m*, then all correct components deliver *m*.
- Integrity: For any message *m*, every correct component delivers *m* at most once, and only if *m* was previously broadcast by *sender(m)*;
- Total Order: If correct components *p* and *q* both deliver message *m* and *m'*, then *p* delivers *m* before *m'* if and only if *q* delivers *m* before *m'*.

These properties guarantee that all messages sent by correct components are delivered only once to all of the intended recipients, and in the same order.

Consolidation of replicated outputs requires a mechanism to allow components to agree on a common value. If an underlying atomic multicast protocol is used to disseminate each of the outputs, the consolidation mechanisms needs just to guarantee the following properties (based on the consensus agreement (Hadzilacos and Toueg, 1993)):

- Validity: If all components that propose a value propose *v*, then all correct components decide *v*;
- Agreement: If a correct component decides *v*, then all correct components decide *v*.

The integrity property (Hadzilacos and Toueg, 1993) (which states that the value decided must be present in the set of proposed values) is not considered, as it precludes decisions on values different than those proposed, like average or median functions.

3.3. Schedulability Analysis of Real-Time Applications

In order to allow the comprehension of the difficulties associated with the development of applications conforming to the pre-emptive fixed-priority model, this section provides a brief overview of current state-of-the-art schedulability analysis for guaranteeing the real-time requirements of the system. In the considered analysis (the response time analysis approach (Joseph and Pandya, 1986; Audsley *et al.*, 1993)), these requirements are guaranteed by checking, before run-time, that the scheduling of the application task set is feasible. This is accomplished by calculating the worst-case response time of each task, verifying if it is smaller than the associated deadline. The advantages of response time analysis is its precision (often exact or nearly exact) and great flexibility in choice of process models. It is also applicable to other type of resources (such as CAN networks).

In the pre-emptive fixed-priority model, an application is constituted by a set of concurrent processing units (tasks) which may be periodic or sporadic. Each task can

only be released by one event, but may be released an unbounded number of times. A periodic task is released by the runtime (temporal invocation), while a sporadic task can be released either by another task or by the environment.

A periodic task is characterised by its period (time interval between consecutive requests), its worst-case execution time (maximum time that it takes to execute per period, without considering the existence of other tasks) and its relative deadline (maximum duration for the response to be performed, related to the instant of the release). For the case of a sporadic task, as it is usually released by some event, a minimum time interval between requests is usually considered to allow schedulability analysis to be performed.

In the fixed priority pre-emptive model, priorities are off-line allocated to the system tasks and, at any instant, the task with the greatest priority that is ready to run is assigned to the processor. Note that although a task may be released at a specific instant, its start of execution can be delayed, if the processor is currently executing a task with a priority greater or equal than the priority of the released task.

Several different priority assignment approaches exist. The Rate Monotonic (RM) approach (Liu and Layland, 1973) assigns to the tasks a priority level based on their periods (the smaller the period the higher the priority). However, when considering sporadic tasks (that can have a large minimum inter-arrival time with stringent deadline, *e.g.* an alarm), it is not reasonable to base their priorities in the minimum inter-arrival time. Therefore, the Deadline Monotonic (DM) priority assignment (Leung and Whitehead, 1982), which gives the tasks a priority level based on their deadlines (the smaller the deadline the higher the priority), can be used.

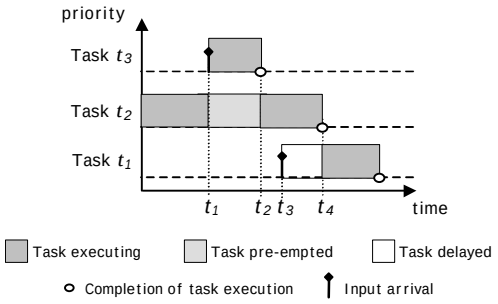

Figure 3.4. Pre-emptive fixed priority model

Figure 3.4 presents an example of a pre-emptive fixed priority model. In this example three tasks compete for the node processor. From instant zero until instant t_1 task t_2 is allocated to the processor, as it is the only task ready for execution. At instant t_1 some event (either interrupt or clock-based) causes task t_3 to be ready for execution. As this task has higher priority than task t_2, it will pre-empt this last task. At instant t_2, task t_3 finishes its execution, thus it allows task t_2 to resume execution. At instant t_3, an event causes task t_1 to be ready for execution. However, as this task has lower priority than the one executing, its execution will be delayed until instant t_4, when task t_2 finishes execution. This behaviour differs from the traditional cyclic executive model, where a

table of sequence of task executions is off-line determined, and execution is carried statically following the scheduling table.

3.3.1. Single Node Scheduling

The schedulability analysis considers the existence of a set of tasks, which may be periodic or sporadic. A task is defined as:

$$t_i = (C_i, T_i, D_i) \tag{3.6}$$

where t_i defines a task i, with a worst-case execution time of C_i and a periodicity of arrival defined by T_i (for the case of a sporadic task, T_i refers to the minimum inter-arrival time). D_i is the relative deadline of the task.

Joseph and Pandya (Joseph and Pandya, 1986) proved that the worst-case response time of a task occurs when all tasks are simultaneously released at their maximum rate. This simultaneous release is referred to as the critical instant. The response time is thus given by (Joseph and Pandya, 1986; Audsely *et al.*, 1993):

$$R_i = C_i + I_i \tag{3.7}$$

where I_i is the maximum interference that task i can experience from higher-priority tasks. During the interval $[0,R_i)$, that is the time interval from the instant when task i is released and the time instant of its worst-case response time, the number of releases of a task j is:

$$I = \left\lceil \frac{R_i}{T_j} \right\rceil \tag{3.8}$$

where the ceiling function ($\lceil\rceil$) produces the smallest integer greater than the result of its parameter. Each release of a higher-priority task j will interfere with task i by C_j. Hence:

$$I = \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{3.9}$$

The interference that task i will suffer from all higher-priority tasks can then be determined ($hp(i)$ is the set of tasks with higher priority than task i):

$$I_i = \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j \right) \tag{3.10}$$

Therefore, by replacing equation (3.10) in equation (3.7) the worst-case response time of task i is:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j \right) \tag{3.11}$$

This equation is mutually dependent, since R_i appears in both sides of the equation. In order to solve this dependency, a recurrence relationship may be used (Audsley *et al.*, 1993):

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \right) \tag{3.12}$$

This recursion ends when R_i^{n+1} equals R_i^n, or when it exceeds D_i (in this case the task is not schedulable), since it can be shown that the series is either convergent or monotonically increasing (Audsley *et al.*, 1993).

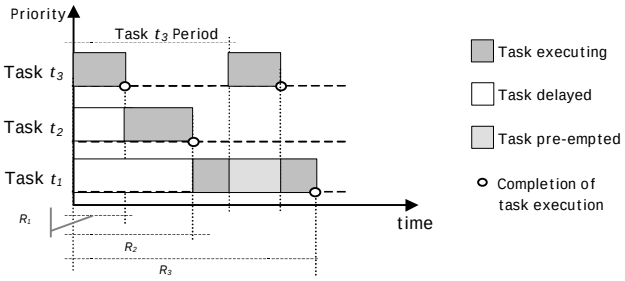

Figure 3.5. Task response time

Figure 3.5 presents an example of the task response time evaluation. In this example three tasks compete for the node processor. In order to determine the worst-case response time of the three tasks, it is considered that they are all released at the same time (critical instant).

Since task τ_3 is the higher priority task it will execute without being delayed, and its worst-case response time (WCRT) is equal to its worst-case execution time (WCET):

$$R_3 = C_3 \tag{3.13}$$

Task τ_2, on the other hand, will suffer the interference of one occurrence of task t_3, thus its WCRT will be equal to its WCET plus the WCET of task t_3:

$$R_2 = C_2 + \left\lceil \frac{R_2}{T_3} \right\rceil C_3 = C_2 + C_3 \tag{3.14}$$

Finally, task t_1 will suffer interference both form task t_2 and from task t_3. Since the period of task t_3 is inferior the WCRT of task t_1, it will execute twice during task t_1 execution.

Therefore, task t_1 WCRT becomes:

$$R_1 = C_1 + \left\lceil \frac{R_1}{T_3} \right\rceil C_3 + \left\lceil \frac{R_1}{T_2} \right\rceil C_2 = C_1 + 2C_3 + C_2 \tag{3.15}$$

In a real-time application is important to allow tasks to interact with each other. However, when a higher priority task pretends to access a resource that is currently being used by a lower priority task (for instance a semaphore), it will be blocked. In Figure 3.6, when task t_3 tries to access a shared resource it becomes blocked, since this resource is currently being used by task t_1. This means that, during this interval, the effective priority of the lower-priority task is greater than the priority of the higher-priority task (priority inversion). As the low priority task may be pre-empted by any number of medium priority tasks (task t_2 pre-empts task t_1, and further blocks task t_3), the duration of the priority inversion may be unbounded. It is thus necessary to upper bound the duration of the maximum blocking that each task may suffer.

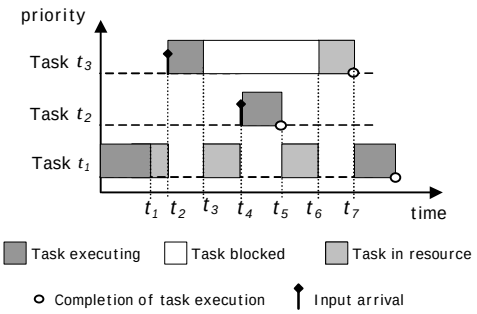

Figure 3.6. Blocking example

In (Audsley *et al.*, 1993) the response time analysis was also extended to incorporate blocking introduced by task interaction. Equation (3.11) is then updated by:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j \right) \tag{3.16}$$

where B_i represents the maximum blocking time that task i can suffer. Determining B_i depends on the particular protocol that is used for managing priority inversion. The Priority Ceiling Protocol (Sha *et al.*, 1990) is one of the protocols proposed to upper-bound priority inversion periods, which also precludes deadlocks and blocking chains. In this approach, resources are also assigned a priority (ceiling priority), which must be equal or higher than the priority of any task that can use the resource.

The maximum priority inversion period that task i can suffer is equal to the longest time interval of any lower priority task accessing a resource with a ceiling priority equal or higher than the priority of task i. This allows to bound the blocking time as (Burns and Wellings, 1995b):

$$B_i = \max_{j \in lp(i)} \left(\max_{r \in res(i)} (CS(t_i, r)) \right) \tag{3.17}$$

where $lp(i)$ is the set of tasks with a lower priority than task i, $res(i)$ is the set of resources with a ceiling priority greater than the priority of task i, and $CS(t_j, r)$ is the worst-case execution time of task j while blocking resource r.

3.3.2. Distributed Scheduling

Analysing tasks and messages' response times in distributed systems must also be considered. However, as the arrival patterns of tasks and messages are mutually dependent, an appropriate analysis must be used. Consider the example presented in Figure 3.7. The task in node *1* is a periodic task that always sends a message to the receiving task in node *2*. Consequently both the message and the receiving task in node *2* inherit the period of the sending task. The figure represents two consecutive executions of the transaction. In the first execution of the sending task, it suffers interference from higher-priority tasks (or blocking from lower-priority tasks) and thus it is delayed. The same happens with the message sent, since it has to wait for the completion of transfer of other messages. As in the second execution of the sending task, it does not suffer any delay, the message will inherit the jitter (variability in the release of the task) of the sending task. Furthermore, the receiving task in node *2* will inherit the jitter of the message (both caused by the sending task and by the interference in the message itself).

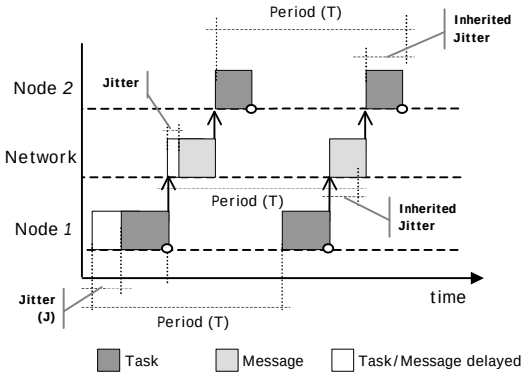

Figure 3.7. Distributed transaction

Therefore, it is possible to realise that the schedulability analysis of the messages in the network is dependent on the release jitter of the sending task, and the schedulability analysis of the receiving task is dependent on the message jitter. It is necessary to consider a holistic approach, where both analyses are integrated.

Tindell and Clark (Tindell and Clark, 1994) addressed this issue, considering a time division multiple access (TDMA) network with real-time properties. The devised solution is based on the fact that both schedulability analysis equations of the node and of the network are monotonic. A recurrent solution was provided (similar to the solution used to determine tasks response times). In the first iteration, the inherited release jitter of both the tasks and the messages is considered to be zero. In each iteration, the inherited release jitter is set according to the response time result of the previous iteration. This solution can thus be used to determine the overall response times of the system, if the network schedulability analysis equations are also monotonic. As it will be presented in the following section, the same reasoning can be applied to the CAN network, which is the real-time network used in the DEAR-COTS architecture.

3.4. The Controller Area Network

The Controller Area Network (CAN) (ISO, 1993) was originally developed to be used in road vehicles to interconnect microprocessor-based components. It is also considered for the automated manufacturing and distributed process control environments (Zuberi and Shin, 1997), and is being used as the communication interface in proprietary architectures, such as DeviceNet (Rockwell, 1997). Several studies on how to guarantee the real-time requirements of messages in CAN networks are available, *e.g.* (Tindell *et al.*, 1995), providing the necessary pre-run-time schedulability analysis equations for the timing analysis of the supported traffic.

The CAN protocol implements a priority-based bus, with a carrier sense multiple access with collision avoidance (CSMA/CA) medium access control (MAC), where bus signals can take two different states: *recessive bits* (idle bus), and *dominant bits* (which always overwrite recessive bits). The collision resolution mechanism works as follows: when the bus becomes idle, every node with pending messages will start to transmit. If a node transmitting a recessive bit reads a dominant one, it means that there was a collision with a higher-priority message, and consequently the transmission is aborted. The highest-priority message being transmitted (the one with most leading dominant bits) will proceed without perceiving any collision, and thus will be successfully transmitted. Nodes that loose the arbitration phase will automatically retry the transmission of requested messages.

There are 4 types of frames that can be transferred in a CAN network. Two are used during the normal operation of the CAN network: the Data Frame, which is used to send local data and the Remote Frame, which is used to request remote data. The other two are used to signal an abnormal state of the CAN network: the Error Frame signals the detection of an error and the Overload Frame signals that a node is not ready to transmit data.

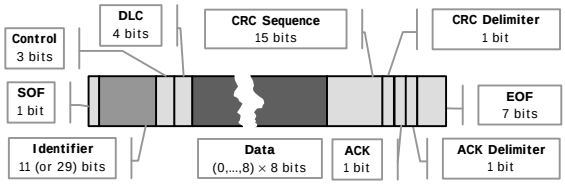

Figure 3.8. Structure of a CAN Data Frame

Figure 3.8 shows the structure of a Data Frame (specific fields: SOF, Identifier, Control, DLC, CRC and EOF are described in (ISO, 1993)). A Remote Frame has the same structure (without data field) and identifier of the remotely requested Data Frame. The structure of both the Error and the Overload Frames will be presented in the following subsection.

At the physical layer, frames are transmitted using the NRZ (Non Returning to Zero) coding technique, with the insertion of stuff bits. That is, whenever there are more than five equal consecutive bits (up to the end of the CRC Field), there is the insertion of an opposite bit in the frame. This opposite bit will be detected and removed by the physical

layer at the receiving side. This bit stuffing technique ensures that, in the normal behaviour, there will never be more than 5 consecutive equal bits on the bus.

3.4.1. Error Detection and Recovery Mechanisms

In the CAN protocol, all the nodes continuously monitor every frame being transmitted on the bus, to detect any transmission error. The node that first detects an error, starts the transmission of an Error Frame (which starts with 6 consecutive dominant bits). The transmission of an Error Frame is an efficient way for the CAN protocol to tolerate transient failures (*e.g.* due to electromagnetic interference).

This Error Frame transmission is immediate, pre-empting the ongoing transmission and avoiding the reception of invalid frames by the other nodes. As a consequence, all the receiving nodes know that the frame being transmitted has an error. Thus, the transmitting node will automatically retry the transmission of the message.

An Error Frame has the following structure:

- 6-12 consecutive dominant bits (Error Flag). The node that first detects the error starts transmitting the Error Flag and hopefully every node will also recognise such error at the same instant. However, there is the possibility that other nodes only recognise the bit stuffing error induced by the Error Flag. In this case, such nodes will start transmitting Error Frames and the Error Flag will be 12 bits long;
- 8 consecutive recessive bits (Error Delimiter) which signal the end of the Error Frame.

Sending Error Frames is a very interesting mechanism to ensure that every node sees the same global state of the network (state coherence). However, it is possible that a failure in a node induces the transmission of consecutive error frames, blocking all the ongoing communications.

To solve this problem, CAN controllers have two error counters (for transmitting and receiving errors, respectively) to isolate erratic nodes. For instance, if a node is consecutively signalling errors in every Data/Remote Frame (*e.g.*, due to a circuitry failure), there is a time bound after which the node cannot signal any more error with active Error Flags.

The values of these counters, which determine the operating state of the node, are increased or decreased (at different rates) as a function of the type of the detected error. These error counters acts as self-surveillance mechanisms, which disconnect faulty nodes (fault-confinement techniques).

Therefore, CAN controllers may operate in three different modes:

1) Error-active, which is the normal operating mode.
2) Error-passive, where the node is still able to transfer/receive messages, but it must wait some time before initiating a transmission (automatically decreasing the transmission priority) and the error signalling is performed with passive Error Flags (6 consecutive recessive bits). When in this operating mode, the node can no longer interfere with frames transmitted by other nodes.
3) Bus-Off, where the node is not able to transfer/receive messages.

3.4.2. Response Time Analysis of CAN Networks

In order to guarantee the real-time requirements of messages transferred by CAN networks, it is necessary to evaluate their worst-case response time. In (Tindell *et al.*, 1995) the authors address in detail the response time analysis of CAN networks. The analysis assumes fixed priorities for message streams (since access to the medium is based on the fixed identifiers, which must be assigned to message streams according to their priority) and a non-preemptive model (since lower-priority messages being transmitted cannot be pre-empted by pending higher-priority messages). The schedulability analysis of (Audsley *et al.*, 1993) presented in Section 3.3 is then adapted to the case of scheduling messages in a CAN network.

The analysis assumes a network with n message streams defined as:

$$S_m = (C_m, T_m, J_m, D_m) \tag{3.18}$$

where S_m defines a message stream m characterised by a unique identifier. A message stream is a temporal sequence of messages concerning, for instance, the remote reading of a specific process variable. C_m is the longest message duration of stream S_m and T_m is the periodicity of its requests. J_m is the jitter of the queuing of a message of stream S_m. Finally, D_m is the relative deadline of a message; that is, the maximum time interval between the instant when the message request is placed in the outgoing queue and the instant when the message must be completely transmitted.

The worst-case response time of a queued message, measured from the arrival of the message request to its complete transmission, is:

$$R_m = J_m + I_m + C_m \tag{3.19}$$

The schedulability of the message stream set is guaranteed if every message has a response time smaller than its deadline. The term I_m represents the worst-case queuing delay (longest time interval between the arrival of the message request and the start of its transmission).

The message duration of stream S_m (C_m) can be evaluated considering that for each Data Frame there is a Data Field plus 44 overhead bits. Additionally, it must be considered the overhead concerning bit stuffing and inter-frame spacing (3 bits of minimum spacing between two consecutive frames). Bit stuffing mechanisms are only applied to 34 bits of the overhead and to the 0..8 byte Data Field (it excludes the CRC delimiter, ACK and EOF). Therefore, the duration of a CAN message is given by:

$$C_m = \left(\left\lfloor \frac{34 + 8 \times n}{5} \right\rfloor + 47 + 8 \times n \right) \times t_{bit} \tag{3.20}$$

where n is the number of data bytes in the message and t_{bit} is the duration of a bit transmission.

The Deadline Monotonic (DM) priority assignment (Leung and Whitehead, 1982) can be directly implemented in a CAN network, by setting the identifier field of each message stream according to the DM rule. Therefore, the worst-case queuing delay of a message of stream S_m is (Tindell *et al.*, 1995):

$$I_m = B_m + \sum_{\forall j \in hp(m)} \left(\left\lceil \frac{I_m + J_m + t_{bit}}{T_j} \right\rceil \times C_j \right) \tag{3.21}$$

where $hp(m)$ is the set of message streams with higher-priority than S_m.

B_m is the worst-case blocking factor, which is equal to the longest duration of a lower priority message, since a message can be blocked at most once in the access to the shared resource (the network):

$$B_m = \max_{\forall k \in lp(m)} \{0, C_k\} \tag{3.22}$$

where $lp(m)$ is the set of message streams with lower-priority than message stream S_m.

As in node schedulability analysis, equation (3.21) embodies a mutual dependency, since I_m appears in both sides of the equation, meaning that it can be solved with a recurrent relationship.

The main difference between the non-preemptive and pre-emptive models is that, in equation (3.21), contrarily to the pre-emptive case (equation (3.10)), the actual transmission time of message m (C_m) is not considered during the recurrent analysis of interference. This is due to the fact that in the non-preemptive case a message being transmitted cannot be pre-empted by higher-priority messages.

Note that these schedulability analysis equations are also monotonic, thus they can be used in the holistic approach presented in Section 3.3.2, making it possible to determine the overall response times of the system.

3.4.3. Network Load

The computation of the network load is a single measurement based on the characteristics of the message streams. Such network load can be evaluated as follows:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{3.23}$$

3.4.4. Inaccessibility Analysis of CAN Networks

The use of CAN networks to support fault-tolerant real-time applications requires not only time-bounded transmission services, but also a minimum level of confidence on the continuity of service. Such continuity of service is not fully guaranteed in CAN networks, since they may be disturbed by temporary periods of network inaccessibility (periods during which nodes cannot communicate with each other, due to the existence of on-going error detection mechanisms).

Considering the existent error recovery mechanisms, the longest network inaccessibility (Rufino and Veríssimo, 1995) results from a Form Error (incorrect structure of the frame) detected at the end of the EOF delimiter. Such network inaccessibility is:

$$t_{ina} = C_{MAX} + C_{error} + C_{IFS} \tag{3.24}$$

where C_{error} and C_{IFS} are the duration of an Error Frame and the Inter-Frame Spacing (two consecutive frames must be separated by at least 3 recessive bits), respectively, and C_{MAX} is the longest duration of a CAN message.

In the presence of multiple bus errors, two different scenarios can be considered (Rufino and Veríssimo, 1995):

- A burst of successive bit errors, where only the first one corresponds to a bit corruption in a Data Frame. The others will just disturb Error Frames being transmitted in response to the first error.
- A longer network inaccessibility results from considering that bus errors are sufficiently apart to interfere with *n* Data Frames. This results in *n* failed attempts to transmit a Data Frame.

The network inaccessibility resulting from this second scenario is:

$$t_{n\_ina} = n \times \left(C_{MAX} + C_{error} + C_{IFS} \right) \tag{3.25}$$

In addition to the frame error detection mechanisms, CAN controllers have two error counters to isolate erratic transceivers, preventing them from interfering with the normal bus operation (see Section 3.4.1). The values of these counters are increased or decreased (at different rates), as a function of the detected error.

In the case of an erratic transmitter, the maximum number of transmission errors (leading to the transmission of Active Error Frames) is given by:

$$n_{tx} = \left\lceil \frac{ect}{\Delta_{tx}} \right\rceil \tag{3.26}$$

where *ect* is the error count threshold, and Δ_{tx} is the increase of the counter at each detected transmission error. As $ect=127$ and $\Delta_{tx}=8$, then 16 consecutive active Error Frames will be transmitted before a failed transmitter enters into the Error-Passive state.

For the case of a receiver, the maximum number of receiving errors (leading to the transmission of active Error Frames) is given by:

$$n_{rx} = \left\lceil \frac{ect}{\Delta_{rx1} + \Delta_{rx2}} \right\rceil \tag{3.27}$$

where Δ_{rx1} and Δ_{rx2} are used according to the detected error (ISO, 1993). As $\Delta_{rx1}=8$ and $\Delta_{rx2}=1$, then 15 Active Error Frames will be transmitted before a failed receiver enters into the Error-Passive state.

Although the time interval during which an erratic transceiver can interfere with the normal behaviour of the network is upper-bounded, an erratic transceiver will only stop transmitting Active Error Frames when its error count reaches the Error-Passive threshold. Hence, it can cause up to 16 failed transmissions in the network.

3.4.5. Inconsistencies in Messages' Transfer

In spite of the extensive error detection and recovery mechanisms in CAN networks, there are some known reliability problems (Rufino *et al.*, 1998) that can lead to an inconsistent state of the supported applications. This misbehaviour is a consequence of

different error detection mechanisms at the transmitter and receiver sides. A message is valid for the transmitter if there is no error until the end of the transmitted frame. If the message is corrupted, a retransmission is triggered according to its priority. For the receiver, a message is valid if there is no error until the last but one bit of the received frame, being the value of the last bit treated as 'do not care'. Thus, a dominant value in the last bit does not lead to an error, in spite of violating the CAN rule stating that the last 7 bits of a frame are all recessive.

In Figure 3.9, the Sender node transmits a frame to Receivers A and B. Receiver B detects a bit error in the last but one bit of the frame. Therefore, it rejects the frame and sends an Error Frame (requesting the frame retransmission) starting in the following bit (last bit of the frame). As for receivers the last bit of a frame is a 'do not care' bit, Receiver A will not detect this error and will accept the frame. However, as the transmitter re-schedules the frame, Receiver A will have *an inconsistent message duplicate*. The use of sequence numbers in messages can easily solve this problem, but it does not prevent messages from being received in different order, not guaranteeing total order of atomic multicasts.

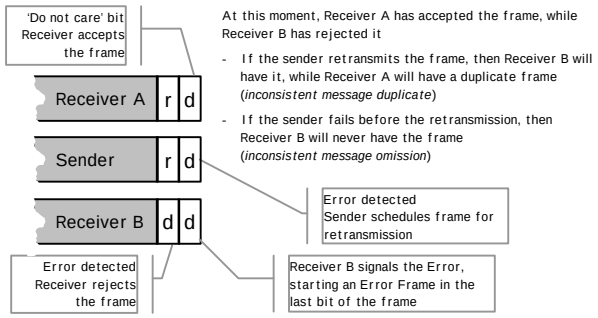

Figure 3.9. Inconsistency in CAN

On the other hand, if the Sender fails before being able to successfully retransmit the frame, then Receiver B will never receive the frame, although Receiver A has delivered it. This situation causes an *inconsistent message omission*, which is a more difficult problem to solve.

In (Rufino *et al.*, 1998), the probability of message omission and/or duplicates is evaluated, in a reference period of one hour, for a 32 node CAN network, with a network load of approximately 90%. Bit error rates ranging from 10^{-4} to 10^{-6} were used, and node failures per hour of 10^{-3} and 10^{-4} were considered. For inconsistent message duplicates the results obtained ranged from 2.87×10^{1} to 2.84×10^{3} duplicates per hour, while for inconsistent message omissions the results ranged from 3.98×10^{-9} to 2.94×10^{-6} omissions per hour.

These values show that for fault-tolerant real-time communications, CAN built-in error recovery mechanisms are not sufficient, since the use of CAN networks to support fault-tolerant real-time applications requires not only time-bounded transmission, but also the guarantee of consistency for the supported applications. Therefore, additional

mechanisms must be provided to ensure fault-tolerant real-time communication in CAN networks.

Furthermore, from the requirements imposed to the communication infrastructure in order to support active replication (Section 3.2.3), it can be perceived that CAN networks do not support atomic multicast properties. CAN error detection and recovery mechanisms ensure the Validity property, since when the sender is correct, all correct nodes will receive the message. Note that the network can be referred as a fail-consistent bus (Powell, 1992), since there is no possibility for different nodes to receive the message with different values. CAN error detection and recovery mechanisms are not, however, sufficient to guarantee the Agreement and Integrity properties (Rufino *et al.*, 1998). In fact, it is possible for a correct node to receive a message not received by some other correct node (inconsistent message omission), and it is also possible that some node receives the same message more than once (inconsistent message duplicate). Total Order is also not guaranteed, since new messages can be interleaved with retransmissions of failed messages, causing different nodes to receive the messages in different order.

The problem of inconsistent messages in CAN networks has been given some research in the last few years. In (Rufino *et al.*, 1998), a set of fault-tolerant broadcast protocols is proposed, solving the message omission and duplicate problems. In this set of protocols, atomic multicast is addressed by the TOTCAN protocol. This protocol is based on the transmission of a second data-free message (ACCEPT message), to signal that the sender is still correct, meaning that the related message can be delivered.

The transmission of the ACCEPT message is performed using a lower layer protocol (EDCAN), which is based on the retransmission of messages by every node in the system (that has correctly received the message). When a node receives a retransmission of the ACCEPT message, it will retransmit it again (even if it already has retransmitted the original ACCEPT), and multiple retransmissions will occur in normal operation even if no error occurs. Therefore, such protocols do not take full advantage of the CAN synchronous properties, producing a great run-time overhead under normal operation.

Another approach presented in the literature is to use a hardware-based solution (Kaiser and Livani, 1999) to prevent message inconsistencies. This approach is based on a hardware error detector, which automatically retransmits messages that could potentially be omitted in some nodes. This detector (SHARE) detects the bit pattern that occurs in an inconsistent message failure, and automatically retransmits the received frame, even if the transmitter handles this failure.

Although this hardware-based approach solves the inconsistent message omission problem of CAN, it does not provide solution to total order, as duplicates may occur (furthermore, inconsistent message omissions are transformed in inconsistent message duplicates). In order to achieve order, it is necessary to complement this mechanism with an off-line analysis approach (Livani and Kaiser, 1999).

In this analysis, messages must be separated in hard real-time and soft real-time. Only hard real-time messages have guaranteed worst-case response time inferior to the deadline, but it is necessary to use fixed time slots, off-line adjusting these messages to never compete for the bus, thus causing an increased burden in the analysis of the system.

3.5. The Ada 95 Language

The Ada language (ANSI, 1983; ISO, 1987) was designed for the development of embedded systems software, in order to replace multiple languages used at the United States Department of Defence. The Ada 95 language revision (ISO/IEC, 1995) brought a more open and extensible language without losing the inherent integrity of Ada 83, addressing the limitations of Ada 83 (Sha and Goodenough, 1990) for hard real-time systems programming. The need to target a broad range of applications led to the specification of a core language (self-contained) and the provision of extra annexes targeting special domains. Since in this work the interest is in the 1995 standard, from now on the term Ada will be used instead of Ada 95.

In Ada, concurrency is supported in the core of the language, through the specification of tasking constructs and of a new mechanism for task interaction: the protected object. In an Ada program there can be a number of concurrent processing units (termed *tasks*), each one with its own flow of control. These tasks are related to the POSIX notion of threads (IEEE, 1995), since all tasks in an Ada program share the same resources (memory area, files, etc).

The new task interaction model based on protected objects was one of the major changes introduced in the language revision (Intermetrics, 1995). A protected object is a passive object, which exports a set of protected operations. These operations provide mutual exclusion to tasks accessing the object data. Therefore, they can be used to provide a more efficient solution to build asynchronous communication and mutual exclusion mechanisms than the Ada 83 Rendezvous mechanism. The clearly data-oriented view brought by the protected object fits in naturally with the general spirit of the object-oriented paradigm.

The specification of the object exports the subprograms that can be used to access the object data: procedures, functions and entries. A procedure can access the private data with exclusive read/write access, while a function can only read it. A protected entry also has read/write exclusive access like a procedure, but it also specifies a barrier condition, that must be true before the task is allowed to proceed. For each entry there is a queue, where the calling tasks wait for the condition to become true.

3.5.1. Ada Support for Real-Time

Although part of the support that Ada provides to real-time systems is through the facilities provided in the core language (such as concurrency), many of the important features of Ada (concerning real-time) are provided in the Real-Time Systems and Systems Programming annexes.

The Real-Time Systems (RTS) annex provides the language with the necessary capabilities for schedulability analysis, namely the support of the Priority Ceiling Protocol in the access to protected objects, priority queuing and First-In-First-Out (FIFO) queuing within priorities. Since the goal of the language revision was to support a broad range of application domains, the language's scheduling model was removed from the core of the language and provided through the RTS annex (Intermetrics, 1995).

The RTS annex supplies the capabilities to associate priorities to Ada tasks and protected objects (ceiling priority), either at creation time or (for the case of tasks)

dynamically during execution. In spite of the provision for supporting different scheduling models, the RTS annex only specifies one scheduling model. A dispatching policy must exist, defining tasks to execute until they are blocked (FIFO_Within_Priorities). This also implies the need for the ceiling locking policy for protected object locking, that is, each protected object has an associated ceiling priority equal or greater than the priority of the highest priority task that can use the object. This locking policy is based on the Immediate Ceiling Priority Protocol (Rajkumar *et al.*, 1995), which is an implementation variant of the Priority Ceiling Protocol presented in Section 3.3, thus it provides bounded priority inversion in the access to protected objects. When a task executes an operation on a protected object that has a higher priority than the caller, the calling task will inherit this higher value as its active priority, but only during the execution of the protected operation. A policy for task queuing based on the task priority is also available, establishing priority as the criterion for task selection instead of order of arrival.

Furthermore, the RTS annex specifies a monotonic and accurate timing capability, mechanisms for synchronous and asynchronous task control and also tasking restrictions that, for instance, can impose a maximum number of tasks, no asynchronous control or no dynamic priorities.

An implementation providing the Real-Time System annex must also provide the Systems Programming annex. This annex specifies the access to machine code, interrupt handling and packages for general task identification and attributes. The existence of Ada mechanisms to specify the exact size and layout of user data types and its simple interface with other languages simplify the task of hardware interfacing.

3.5.2. Ada Support for Fault Tolerance

Although targeted to application domains where fault tolerance is required, the language does not provide direct support for fault tolerance mechanisms, except for the exception mechanism, which is a powerful tool that has been often used to provide higher-level mechanisms for software fault tolerance. Note that software fault tolerance refers to the tolerance of software faults, not to software-based fault tolerance where software is used to tolerate faults in the underlying components.

Software fault tolerance has been a topic of research on Ada in the last few years. A number of approaches have been proposed (*e.g.*, atomic actions (Wellings and Burns, 1997), recovery blocks (Rogers and Wellings, 1999) and transactions (Kienzle, 2001)), providing an extensive number of mechanisms for software fault tolerance in Ada 95.

Work is also being done in the integration of fault tolerance and distribution in Ada. Ada provides a Distributed Systems annex, which defines the model of distribution of Ada applications. A distributed application is seen as a set of program components that are distributed on the system nodes, communicating through the network. These components are called partitions (ISO/IEC, 1995). Interconnection between partitions is performed through well-defined interfaces, with specific rules defining how partitions can interact. In (Wellings and Burns, 1996) and (Burns and Wellings, 1998) the use of replication within the partition model is evaluated, and some replication mechanisms, which must be explicitly used by the application, are provided. Wolf (Wolf and Strohmeier, 1999) presents some issues regarding replica implementation within the

same partition model. Replica determinism is initially assumed and later extended to non-deterministic replicas. This approach is based on the extension of the run-time support to implement transparent replication of partitions.

ReplicAda (Heras-Quirós *et al*., 1997) presents another fault-tolerant implementation using the Ada Distributed Systems annex. It is based on a layer under the Partition Communication Subsystem that presents a transparent view to the programmer, hiding all the replication issues. This approach assumes replica determinism, mainly through the use of the Ada Restriction *pragma*. Drago (Miranda *et al.*, 1996) is another Ada extension intended for distributed fault-tolerant applications programming. The approach is based on enriching the language with new constructs, providing mechanisms for explicitly supporting the group abstraction in distributed systems.

However, the partition model of Ada does not allow flexibility in the configuration of the system, since partitions are simultaneously the unit of replication and distribution, not being possible to de-couple these roles. It is not possible to allocate a partition to one node, and one of its replicas distributed through several nodes.

3.5.3. The Ravenscar Profile

The Ada 95 programming language is widely used in the areas of critical hard real-time systems. Nevertheless, while the language provides a broad set of programming constructs, the multitasking mechanisms are rarely used, since they are considered to be too complex to be analysable, thus difficult to be certified. Therefore, a subset of the language multitasking mechanisms was defined (the Ravenscar Profile (Burns, 1997)), in order to reduce the size and overhead of Ada applications, and to allow applications to be certified concerning its real-time and fault tolerance properties.

The profile restricts the language use by removing mechanisms that are considered to be non-deterministic or which introduce high overhead. Even with these restrictions, applications may be built using the pre-emptive fixed priority computational model. Applications conforming to the Ravenscar profile consist in a set of tasks, with all interactions between these tasks performed through the use of protected objects.

Protected objects are only used to provide access to shared resources and to release sporadic tasks. The Profile does not support dynamic creation either of tasks or of protected objects, task termination, alteration of task priorities and the select statement. It also imposes that applications use the available Ceiling Locking mechanism for protected objects, and the FIFO Within Priorities dispatching mechanism. These restrictions force applications to be developed with the pre-emptive fixed priority model of computation, allowing the use of the schedulability analysis approach presented in Section 3.3.

Figures 3.10 to 3.12 present possible templates for developing Ravenscar applications (based on (Dobbing and Burns, 1998)). In Figure 3.10 a template for a periodic task is presented. The task body is a simple infinite loop containing a *delay until* statement at the beginning, in order to block the task until the release time is reached. In each iteration, a new release time for the next iteration is determined. The time type used is the one provided by the Real-Time Systems annex, since it is the only one allowed by the profile. The infinite loop never terminates, since the profile forbids task termination.

```
1:  task body Periodic is
2:     Start: Ada.Real_Time.Time := ...;
3:     Period: Ada.Real_Time.Time_Span := ...;
4:  begin
       -- Initialisation Code
5:     loop
6:        delay until Start;
          -- Task Code
7:        Start := Start + Period;
8:     end loop;
9:  end Periodic;
```

Figure 3.10. Periodic task template

```
1:  protected Release_object is
2:     entry Wait(D: out Task_Data);
3:     procedure Release(D: Task_Data);
4:  private
5:     Data: Task_Data;
6:     Released: Boolean := false;
7:  end Release_object;

8:  protected body Release_object is
9:     entry Wait(D: out Task_Data) when Released = true is
10:    begin
11:       D := Data;
12:       Released := false;
13:    end Wait;
14:    procedure Release(D: Task_Data) is
15:    begin
16:       Data := D;
17:       Released := true;
18:    end Release;
19: end Release_object;

20: task body Sporadic is
21:    data: Task_Data;
22: begin
       -- Initialisation Code
23:    loop
24:       Release_object.Wait(data);
          -- Task Code
25:    end loop;
26: end Periodic;
```

Figure 3.11. Sporadic task template

A sporadic task (Figure 3.11) also has an infinite loop, but with the blocking statement being provided by a call to an entry of a protected object. This protected object must only be used to release a single task, but it allows the transfer of data between the releasing and the released task. It is also possible to use the *Suspension_Object* mechanism defined in the Real-Time Systems annex for the release of sporadic tasks, when no data is to be transferred.

In the profile model, tasks share data asynchronously through the use of a protected object (Figure 3.12), which only exports procedures and functions in its interface (that is, no entries). The reason is that protected entries are only to be used for the release of sporadic tasks.

```
1:  protected Shared_data_object is
2:     procedure Write(D: Obj_Data);
3:     function Read return Obj_Data;
4:  private
5:     Data: Obj_Data;
6:  end Shared_data_object;

7:  protected Shared_data_object is
8:     procedure Write(D: Obj_Data) is
9:     begin
10:       Data := D;
11:    end Write;
12:    function Read return Obj_Data is
13:    begin
14:       return Data;
15:    end Write;
16: end Shared_data_object;
```

Figure 3.12. Shared data template

The Ravenscar profile allows the use of the pre-emptive fixed priority computational model in critical applications, as it is demonstrated by available studies (Lundqvist and Asplund, 1999; Audsley *et al.*, 2000) and implementations (Puente *et al.*, 2000; Aonix, 1998). Moreover, a commercial implementation of the Profile (Raven (Aonix, 1998)) is certifiable under the Avionics Standard DO178B (RTCA, 1992), being already in use in some applications (Wellings, 2000).

Nevertheless, it is considered that further studies are necessary for the use of the profile in replicated and distributed systems (Wellings, 2000). The interaction between multitasking pre-emptive software and replication introduces new problems, which must be considered, particularly for the case of a transparent and generic approach. The restrictions of the Ravenscar profile make difficult the implementation of an efficient support for replicated or distributed programming, which may result on an increased application complexity (Audsley and Wellings, 2000). Therefore, any environment for transparent replication using the Ravenscar profile must be simple to implement and use, but at the same time must provide the capabilities required by the fault-tolerant real-time applications.

3.6. Summary

This chapter presented a survey of relevant work related to the development of fault-tolerant real-time systems. This survey is strictly necessary as the background for the remaining chapters of this thesis.

Initially, a survey of fault-tolerant real-time systems is given, focusing on the issue of software-based fault tolerance mechanisms. Afterwards, the response time analysis

approach for the schedulability analysis of real-time applications is presented, demonstrating how real-time guarantees can be provided to applications.

A survey of the Controller Area Network is also presented, since it is the network used as the communication infrastructure in the DEAR-COTS architecture for the replication and distribution of fault-tolerant real-time applications. A brief survey of the main characteristics of the network is provided, focusing on its real-time behaviour and on its impairments for fault-tolerant communication. The problems of real-time behaviour in the presence of network and transceiver errors and inconsistencies in message transfers are identified, and some discussion on existent solutions is provided.

Finally, a brief description of the Ada 95 language is presented, with a special emphasis on its support for fault-tolerant real-time systems. A small introduction to the concurrency model of the language is presented, together with some discussion on its support for real-time and fault-tolerant systems. A description of the Ravenscar profile is also provided, describing how it can be used for the development of hard real-time applications.

Chapter 4

Replication Management Framework

4.1. Introduction

The DEAR-COTS Hard Real-Time Subsystem (HRTS) provides a framework to support distributed fault-tolerant hard real-time applications. It supports the active replication of software with dissimilar replicated task sets in each node. The HRTS provides a transparent programming model that allows applications to be developed focusing on the requirements of the controlled system, thus abstracting from the distribution/replication details. This model provides a generic solution, allowing applications with different structures and configurations to be developed.

This chapter presents this framework, intended for the support of replicated software components. The framework is based on the structuring of applications in software components, which can then be replicated, and on a repository of generic task interaction objects that hide from the application the details of replication and distribution.

This chapter is largely drawn from (Pinho *et al.*, 2001a), and is structured as follows. Section 4.2 explains how applications can be replicated, and how these applications can be configured according to their real-time and fault tolerance requirements. The proposed approach relies on the definition of a suitable replication unit (the component), which is an abstraction that is used for system configuration, without imposing restrictions on how applications are structured. Therefore, applications are provided with a generic and flexible mechanism for replication configuration, that is independent of their distribution requirements.

Afterwards, Section 4.3 presents the replication framework. This framework is based on a repository of generic task interaction objects that are used in the development of fault-tolerant real-time applications. The repository is used during the design and the configuration phases, and provides a set of generic objects with different capabilities. These objects are supported by a middleware, responsible for the replication and distribution management mechanisms, that also shields the repository from changes in the lower-level communication infrastructure, and minimises the effort needed for the creation of new objects.

Section 4.4 describes the available task interaction objects. These objects provide an object model to application tasks interaction, and hide from the application tasks the low-level mechanisms for replication and distribution. During application development, simple resources (objects) are available for sharing data between tasks and for releasing tasks, not implementing any distribution or replication management. The appropriate

distributed/replicated resources replace these simple resources in the configuration phase. This allows applications to be implemented abstracting from the distribution/replication details, and afterwards configured and allocated throughout the system nodes.

Section 4.5 describes the underlying software layer responsible for the processing of the replication and distribution mechanisms and for the interconnection with the communication infrastructure.

4.2. Replication Model

Although the goal is to transparently manage distribution and replication, it is considered that a completely transparent use of the replication/distribution mechanisms may introduce unnecessary overheads. Therefore, during application development the use of replication or distribution is not considered (transparent approach). Later, in a configuration phase, the system developer configures the application replication and allocates the tasks in the distributed system.

The hindrance of this approach is that, as the application is not aware of the possible distribution and replication, complex applications could be built relying heavily on task interaction. This would cause a more inefficient implementation. However, the model for tasks, where task interaction is minimised, precludes such complex applications. Tasks are designed as small processing units, which, in each invocation, read inputs, carry out the processing, and output the results. The goal is to minimise task interaction, in order to improve the system's efficiency.

In order to allow the use of the response time analysis (Joseph Pandya, 1986; Audsley *et al.*, 1993), each task is released only by one invocation event, but can be released an unbounded number of times. A periodic task is released by the runtime (temporal invocation), while a sporadic task can be released either by another task or by the environment. After being released, a task cannot suspend itself or be blocked while accessing remote data (remote blocking).

Tasks are allowed to communicate with each other using interaction objects, either *Shared Data* objects or *Release Event* objects (which can also carry data). *Shared Data* objects are used for asynchronous data communication between tasks, while *Release Event* objects are used for the release of sporadic tasks. This task interaction model, although simple, maps the usual model found in hard real-time systems.

Note that, in the HRTS, remote blocking is avoided by preventing tasks from reading remote data. Hence, when sharing data between tasks configured to reside in different nodes, the *Shared Data* object must be replicated in these nodes. It is important to guarantee that tasks in different nodes must have the same consistent view of the data. This is accomplished by multicasting all data changes to all replicas. This multicasting must guarantee that all replicas receive the same set of data change requests in the same order, thus atomic multicasts must be used.

4.2.1. Replication Unit

As there is the goal of fault tolerance through replication, it is important to define the replication unit. If the application were defined as the replication unit, in order to replicate part of the application all of its tasks would have to be replicated, unnecessarily increasing the processing load. If the task were the replication unit, each task output would have to be consolidated, unnecessarily increasing the inter-task communication load. The Ada language includes the partition concept (ISO/IEC, 1995), providing a more flexible approach, since the distribution and replication unit is a part of the application. However, it forces the replication unit to be also the distribution unit, precluding replicas to be configured as distributed components.

Thus, using any of these solutions as the replication unit would be very restrictive. It is necessary to devise a new replication unit, de-coupling the roles of distribution and replication units. Therefore, the notion of *component* is introduced. Applications are divided in components, each one being a set of tasks and resources that interact to perform a common job. The component can include tasks and resources from several nodes, or it can be located in just one node. In each node, several components may coexist. This component is just a configuration abstraction, which is used to structure replication units, and to allow system configuration. The degree of replication is defined as *n-replicated component*.

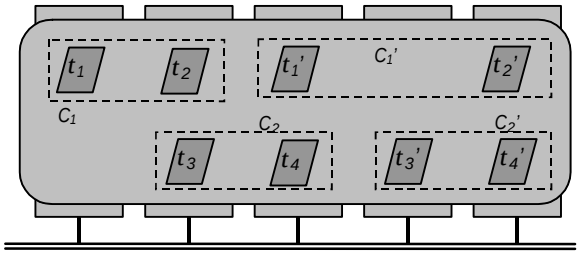

Figure 4.1. Replicated hard real-time application

As an example, Figure 4.1 shows a real-time application with 4 tasks (t_1, t_2, t_3 and t_4). The application is divided in two different components (C_1 and C_2), which are replicated (C_1' and C_2'). Component C_1 encompasses tasks t_1 (in node *1*) and t_2 (in node *2*). Its replica encompasses tasks t_1' (in node *3*) and t_2' (in node *5*). Component C_2 encompasses tasks t_3 (in node *2*) and t_4 (in node *3*), while its replica encompasses tasks t_3' (in node *4*) and t_4' (in node *5*). Note that, although the example presents only *2-replicated* components, in order to tolerate fail-uncontrolled behaviour of the replicas, it is necessary to use 2**f*+1 replicas to tolerate *f* faults (thus *3-replicated* components to tolerate a single fault).

A similar concept is the capsule defined in the Delta-4 architecture (Powell, 1991). As with the DEAR-COTS component, a Delta-4 capsule is the unit of replication, embodying a set of tasks (referred to as threads) and objects. However, a capsule has its own thread scheduling and separate memory space, and is also the unit of distribution. Thus, the Delta-4 concept of capsule is more related to Unix processes, while the presented component is a more lightweight concept, as HRTS components do not have

an implementation counterpart. It is just an abstraction that is used to allow system configuration.

The component concept does not impose restrictions on how applications are structured. Although it is considered that tasks joined in the same component are somehow related, it is the system developer role to decide the component structure of the system. This decision can be taken on the basis of patterns of task interactions or tasks' related roles.

4.2.2. Component Failure Assumptions

The component is the unit of replication, therefore a component is a unit of fault-containment. Faults in one task may cause the failure of one component. However, if a component fails, by producing an incorrect value (or not producing any value), the group of replicated components will not fail since the output consolidation will mask the failed component. This means that, in the model of replication, the internal outputs of tasks (task interaction within a component) do not need to be agreed. The output consolidation is only needed when the result is made available to other components or to the controlled system.

On the other hand, a more severe fault in a component can spread to the other parts of the application in the same node, since there is no separate memory spaces inside the application. In such case, other application components in the node may also fail, but component replication will mask such failure.

Separate memory spaces for applications is not forcibly required, depending on the support provided by the operating system. It is however advocated that, in order to provide the required fault tolerance level, an operating system that allows the use of separated memory spaces should be used. Note that, in order to increase the effectiveness of the fault tolerance mechanisms, the DEAR-COTS Support Software should also reside in a separate memory space.

The replication/distribution mechanisms are essential to the correct behaviour of the system. It is therefore important to prevent errors in the framework's software. If software faults occur in just one node, they can be masked due to the node replication, or the node can be made silent by using the TCB (Veríssimo *et al.*, 2000a). If a greater reliability in the development of this software is considered necessary, other approaches like diversity or validation should be used.

As active replication is used, there is the need to guarantee replica determinism, *i.e.*, that replicated tasks execute with the same data and timing-related decisions are the same in each replica. This determinism can be achieved restricting the application from using non-deterministic timing mechanisms. In this case, the use of multitasking would not be possible, since task synchronisation and communication mechanisms inherently lead to timing non-determinism. Guaranteeing that replicas take the same scheduling decisions, by performing an agreement in every scheduling decision, allows for the use of non-deterministic mechanisms. However, it imposes the modification of the underlying scheduling mechanisms and leads to a huge overhead on the system, since agreement must be made at every dispatching point.

The use of timed messages (Poledna *et al.*, 2000) allows a restricted model of multitasking to be used and eliminates the need for agreement between the internal tasks

of each component. With timed messages, agreement is only needed to guarantee that all replicated components work with the same input values and that they all vote on the final output. The use of timed messages implies the use of appropriate clock synchronisation protocols, since clock deviations must be upper-bounded.

4.2.3. Component Flexibility

By creating components, it is possible to define the replication degree of specific parts of the real-time application, according to the reliability of its components. However, by replicating components, efficiency decreases due to the increase of the number of tasks and exchanged messages. Hence, it is possible to trade failure assumption coverage for efficiency and vice-versa. Although efficiency should not be regarded as *the* goal for a fault-tolerant hard real-time system, it can be increased by decreasing the redundancy degree.

As can be seen in Figure 4.2, several possibilities exist for the configuration of an application. The top part of the figure shows the configuration presented in Figure 4.1, while the bottom part presents a different configuration, where the application is divided in three components and only component C_2 is replicated. The lighter colour is used for the replication-related additions to the system. The double arrows indicate communication between different components, thus communication with consolidated data.

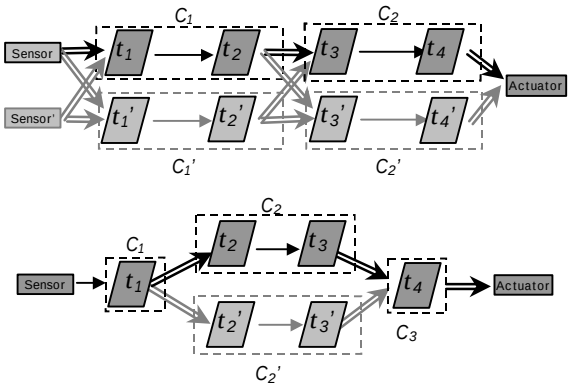

Figure 4.2. Examples of application configuration

Note that the second solution is more efficient, as there are only two more tasks than strictly needed by the application and the number of message exchanges is much smaller. However, the fault coverage of both the sensor and components C_1 and C_3 (and the nodes where they execute) must be higher than in the previous solution, as they are not replicated.

Although not shown in the previous figures, a component can have more than one input task (task that receives data or events from outside the component) and more than one output task (task that output results to other components or to the environment). Component tasks that are not input nor output tasks are named internal tasks.

4.3. Framework Structure

The support to this replication model is provided in a twofold way. First, an underlying software layer (the Communication Manager (Pinho and Vasques, 2001c)) provides the appropriate communication protocols. This Communication Manager provides a group communication interface, including the needed communication mechanisms for replication and distribution. It provides protocols for atomic multicast and for replicated data consolidation. Nevertheless, group communication technology by itself is too low-level to allow complex fault-tolerant applications, and a lot of extra functionality has to be added to applications to guarantee their fault tolerance and performance requirements (Johnson *et al.*, 2000). Thus, there is the need for an extra abstraction between the application and the group communication mechanisms.

The DEAR-COTS architecture provides such extra abstraction by means of a repository of task interaction objects (Figure 4.3). These objects provide a transparent interface, by which application tasks are not aware of replication and distribution issues. Their run-time execution is supported by the Replica Manager, which is a software layer underlying the application. This layer is also responsible for the interface to the communication mechanisms provided by the Communication Manager. Together, these two software layers constitute the HRTS Support Software, a middleware layer between the application and the COTS components.

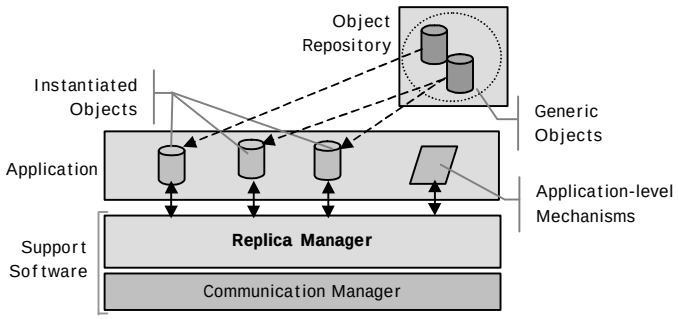

Figure 4.3. Framework structure

The Object Repository is used during the design and configuration phases, and provides a set of generic objects with different capabilities. These generic objects are instantiated with the appropriate data types and incorporated into the application. They are responsible for hiding from the application the details of the lower-level calls to the support software. This allows applications to focus on the requirements of the controlled system rather than on the distribution/replication mechanisms.

The Object Repository does not use an object-oriented approach, as there is no provision for run-time polymorphism and no code inheritance between objects. However, as an object-based approach, it possesses several advantages:

- it allows objects with the same interface to be replaced in the pre-run-time configuration phase;
- it allows the transparent wrapping of replication and distribution mechanisms;
- it eases the modularity and decomposition of the system.

This approach also allows the addition and reuse of new objects. If other generic task interaction objects are later realised to be important, they can be incorporated in the Object Repository (and thus made available to new applications) as long as they follow the same approach as those currently available.

In order to simplify the upgrade of the Object Repository, the Replica Manager layer is responsible for performing the main processing of the provided mechanisms, and the generic objects in the repository just provide the transparency and object model to the applications.

The Replica Manager is also responsible for the consistency of the replicated components. It provides the required mechanisms for supporting both the task interaction objects and the applications (recording periodic tasks' release times and allowing applications to change the state of applications components). The Replica Manager also shields the Object Repository generic objects from changes in the Communication Manager structure, either due to network changes, or due to the use of a different communication layer.

4.3.1. Guaranteeing Replica Determinism

In the model of replication, the internal outputs of tasks (task interaction within a component) do not need to be consolidated. The output consolidation is only needed when the result is made available to other components or to the controlled system. However, it is necessary to provide deterministic execution of replicated components (since active replication is used).

The use of the timed messages concept (Poledna *et al.*, 2000) allows the use of the pre-emptive multitasking model and eliminates the need for agreement between the internal tasks of each component. The use of the timed message concept is transparently managed by the task interaction objects, when it is required.

Release Event objects do not need to use timed messages, since there is a synchronous interaction between the releasing and released tasks. It is known that all the replicas of the released task will also be released by the equivalent event in the replicated component. The same happens if a *Shared Data* object is used only by tasks that have a precedence relation (Poledna *et al.*, 2000), *e.g.* through offsets.

When an application is configured in such a way that a *Shared Data* object is used in a replicated component, it is necessary to keep several versions of the written value, each one associated with its validity time. When a task reads the value, it must read the most recent value that has a validity time older than the release time of the task. Therefore it is necessary to maintain a record of tasks' release times. This is the responsibility of the Replica Manager layer. For periodic tasks, the layer simply stores the task release time, each time it is released. The release time of sporadic tasks released by other tasks, is determined by adding the release time of the releasing task with its best-case response time (tasks released by external events are considered in Sections 4.4.4 and 4.4.5).

When distributed computations are involved, it is not possible for the Replica Manager in the destination node to determine sporadic tasks' release times, or for objects to determine messages' validity times, since these are based on the release time of the source task. Therefore, these are determined at the source and transmitted together with the actual release event or data message.

4.4. Object Repository

In the HRTS, tasks communicate with each other by using *Shared Data* and *Release Event* objects. The Object Repository provides a set of generic objects (Figure 4.4), which are instantiated by the application with the appropriate application-related data types.

Although multiple objects are available, with different capabilities and goals, during application development only three different object types are available in the repository: *Shared Data*, *Release Event* and *Release Event with Data* objects, without any distribution or replication capabilities, since at this stage the system is not yet configured.

```
Shared Data Object

1:   when write(data):
2:      Obj_Data := data

3:   when read:
4:      return Obj_Data

Release Event Object

5:   when wait:
6:      Task_Suspend

7:   when release:
8:      Suspended_Task_Resume

Release Event with Data Object

9:   when wait:
10:     Task_Suspend
11:     return Obj_Data

12:  when release(data):
13:     Obj_Data := data
14:     Suspended_Task_Resume
```

Figure 4.4. Specification of development available objects

These objects have a well-defined interface. Tasks may *write* and *read* a *Shared Data* object and *wait* in or *release* a *Release Event* object. Note that *Release Event* objects are to be used in a one-way communication, thus a task can only have one of two different roles (*wait* or *release*). *Release Event* and *Release Event with Data* objects have a similar interface; the only difference is that with the latter it is also possible to transfer data.

At system configuration time, the application is distributed over the nodes of the system and some of its components are also replicated. Thus, some (or all) of the used objects must be replaced by similar ones with extra capabilities. That is, with distribution and/or replication capabilities.

4.4.1. Simple Program Example

In order to exemplify how the task interaction mechanisms can be used, a simple example is presented. Later on, in Section 4.4.6, the same example is used to demonstrate how applications can be replicated and distributed and how they are affected by the configuration phase.

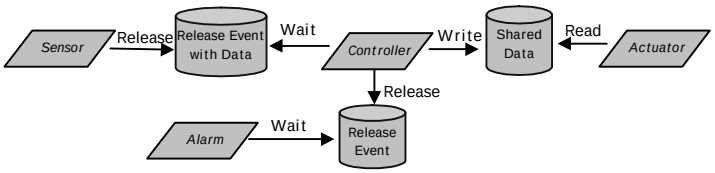

Figure 4.5. Application example

Figure 4.5 presents the structure of the application example. The application is constituted by four tasks, which provide a simple control loop between a sensor and an actuator. The *Sensor* task is a periodic task, responsible for reading the value of the sensor and passing it to the *Controller* task, which performs the control algorithm. A periodic *Actuator* task is then responsible for the actual writing of the output. The *Alarm* task is responsible for some type of notification if the *Controller* task signals an abnormal condition. Note that the purpose is to exemplify how the interaction objects can be used and not to indicate how applications should be structured. Therefore, the structuring of tasks as periodic or sporadic is provided for demonstration purposes only.

The *Sensor* task interacts with the *Controller* task through a *Release Event with Data* object, in order to simultaneously release the *Controller* task and forward it the device data. The *Controller* task performs the control algorithm, and then uses a *Shared Data* object to make the control data available to the *Actuator* task. If an abnormal situation is detected, the *Controller* task releases the *Alarm* task, using a *Release Event* object.

Figures 4.6 and 4.7 present a possible implementation of the application using the Ada 95 language. Figure 4.6 presents the declaration of the needed repository objects. Note that, at this phase, there is no consideration of replication and distribution issues. However, due to the necessity of storing the release time of periodic tasks, these tasks must use an available Replica Manager interface (Figure 4.7, lines 7 and 32) to request their periodic execution (this situation is further detailed in Section 4.5.3).

```
1:  package Device_Event is new
          Object_Repository.Release_Event_With_Data(Device_Data);
2:  Device_Event_Obj: Device_Event.Release_Event_With_Data_Obj;
3:  package Control_Shared_Data is new
          Object_Repository.Shared_Data(Control_Data);
4:  Control_Data_Obj: Control_Shared_Data.Shared_Data_Obj;
5:  package Alarm_Event is new Object_Repository.Release_Event;
6:  Alarm_Obj: Alarm_Event.Release_Event_Obj;
```

Figure 4.6. Application program: object specifications

```
1:   task body Sensor is
2:      Start: Ada.Real_Time.Time := ...;
3:      Period: Ada.Real_Time.Time_Span := ...;
4:      Dev_Data: Device_Data;
5:   begin
6:      loop
7:         Replica_Manager.Request_Periodic(Start);

8:         Read_Some_Device(Dev_Data);
9:         Device_Event_Obj.Release(Dev_Data);

10:        Start := Start + Period;
11:     end loop;
12:  end Sensor;

13:  task body Controller is
14:     Dev_Data: Device_Data;
15:     Ctrl_Data: Control_Data;
16:  begin
17:     loop
18:        Device_Event_Obj.Wait(Dev_Data);

19:        Ctrl_Data := Do_Some_Processing(Dev_Data);
20:        Control_Data_Obj.Write(Ctrl_Data);

21:        if Some_Test(Ctrl_Data) then
22:           Alarm_Obj.Release;
23:        end if;

24:     end loop;
25:  end Controller;

26:  task body Actuator is
27:     Start: Ada.Real_Time.Time := ...;
28:     Period: Ada.Real_Time.Time_Span := ...;
29:     Ctrl_Data: Control_Data;
30:  begin
31:     loop
32:        Replica_Manager.Request_Periodic(Start);

33:        Ctrl_Data := Control_Data_Obj.Read;
34:        Actuate(Ctrl_Data);

35:        Start := Start + Period;
36:     end loop;
37:  end Actuator;

38:  task body Alarm is
39:  begin
40:     loop
41:        Alarm_Obj.Wait;

42:        Some_Notification;
43:     end loop;
44:  end Alarm;
```

Figure 4.7. Application program: task specifications

For the interaction between the *Sensor* and *Controller* tasks, a *Release Event with Data* object is created (Figure 4.6, lines 1 and 2) by instantiating a generic package with the appropriate data type and by declaring an instance of the object. The *Sensor* task uses this object (Figure 4.7, line 9) to *release* the *Controller* task (Figure 4.7, line 18).

For the interaction between *Controller* and *Actuator* tasks a similar approach is used, but with a *Shared Data* object (created in Figure 4.6, lines 3 and 4). The Controller task *writes* to the object (Figure 4.7, line 20), while the *Actuator* task performs the related *read* (Figure 4.7, line 33).

Finally, a *Release Event* object is declared in Figure 4.6 (lines 5 and 6), for the interaction between *Controller* (*release* in Figure 4.7, line 22) and *Alarm* (*wait* in Figure 4.7, line 41) tasks.

Application tasks use these objects, through their well-defined interfaces. The goal is to avoid modification of tasks after the configuration phase, only different objects will be used.

4.4.2. Interaction Internal to a Component

The interaction between tasks belonging to the same component (Figure 4.8) does not require consolidation between replicas of the component. However, it may require the use of distributed mechanisms (if the component is spread through several nodes) or the use of timed messages (if the component is replicated).

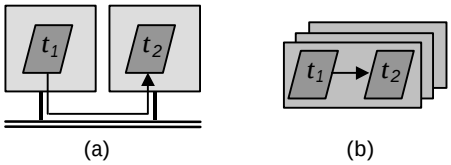

Figure 4.8. Internal interaction: distributed (a) or deterministic (b)

In the case of *Release Events* objects, as it is a synchronous one-way interaction, there is no need to support the timed messages mechanism. However, it is necessary to change the specification for the *Release Event* objects (Figure 4.4), even for the case of non-distributed interaction, as the Replica Manager must store the release time of the sporadic task if the component is replicated. Therefore, in the *Deterministic Release Event* object (Figure 4.9), the *release* interface requests the release to the Replica Manager, and a *private_release* interface is used by the Replica Manager to release the task.

Note that for the case of a non-replicated component it is not necessary to use this object, since storing tasks' release times is only required when the component interacts with a group of replicated components (see Section 4.4.3).

If releasing and released tasks are allocated to different nodes in the system, there is the need for a mechanism to release tasks in other nodes. This is performed by replacing the simple *Release Event* object by two objects (Figure 4.10) that work together to perform the same action (proxy model).

```
Release Event Object

1:   when wait:
2:      Task_Suspend

3:   when release:
4:      Replica_Manager.Request_Release_Event

5:   when private_release:
6:      Suspended_Task_Resume

Release Event with Data Object

6:   when wait:
7:      Task_Suspend
8:      return Obj_Data

9:   when release(data):
10:     Obj_Data := data
11:     Replica_Manager.Request_Release_Event

12:  when private_release:
13:     Suspended_Task_Resume
```

Figure 4.9. *Deterministic Release Event* objects

```
Release Event Proxy Object

1:   when release:
2:      Replica_Manager.Forward_Release_Event

Release Event Receive Object

3:   when wait:
4:      Task_Suspend

5:   when private_release:
6:      Suspended_Task_Resume

Release Event with Data Proxy Object

7:   when release(data):
8:      Replica_Manager.Forward_Release_Event(data)

Release Event with Data Receive Object

9:   when wait:
10:     Task_Suspend
11:     return Obj_Data

12:  when private_release(data):
13:     Obj_Data := data
14:     Suspended_Task_Resume
```

Figure 4.10. *Distributed Release Event* objects

Therefore, in the releasing task side, the *Release Event Proxy* object is responsible for forwarding the event to the corresponding *Release Event Receive* object in the other node. The equivalent pair of objects is also available for the case of the *Release Event with Data* object. The *Proxy* object has only the *release* interface, which requests the Replica Manager to forward the request to the other node. The *Receive* object has the corresponding *wait* interface, and it also provides a *private_release* to be used by the Replica Manager when a request arrives. The release time of the task being released is determined at the source by the Replica Manager (using the available information about the releasing task) and is forwarded to the destination node.

```
1:   when write (data):
2:      t_val := Replica_Manager.Request_Validity_Time(Writing_Task)
3:      DataBuffer := DataBuffer ∪ (data,t_val)

4:   when read:
5:      newest_data := null
6:      t_val := 0
7:      t_rel := Replica_Manager.Request_Release_Time(Reading_Task)
8:      for all i in Data_Buffer loop
9:         if t_val(i) < t_rel and t_val(i)> t_val then
10:           newest_data := Data(i)
11:           t_val := t_val(i)
12:        end if
13:     end loop
14:     return newest_data
```

Figure 4.11. *Deterministic Shared Data* object

For the case of the *Shared Data* object, three situations are identified, concerning the need to support replica determinism, to support distribution (when the object is used by tasks in different nodes) or both. For the first case, the Repository provides a generic object, the *Deterministic Shared Data* object (Figure 4.11) with the same interface of the *Shared Data* object, but with extra functionalities related to the support of timed messages. This object no longer holds a single data element, but a buffer of elements. Associated with each element, the object also records the related validity time. The *write* interface, as well as adding the element to the buffer, requests the related validity time from the Replica Manager layer. The *read* interface requests from the Replica Manager the release time of the reading task, and chooses from the buffer the newest value that is older than this release time.

```
1:   when write (data):
2:      Replica_Manager.Request_Dissemination(Data)

3:   when read:
4:      return Obj_Data

5:   when private_write (data):
6:      Obj_Data := data
```

Figure 4.12. *Distributed Shared Data* object

When a *Shared Data* object is used by tasks allocated to different nodes in the system (distribution requirements), it must be locally replicated in all those nodes (in order to avoid remote reading). Therefore, every *write* to the object must be atomically multicast to all objects. Note that this is a second level replication, which is independent of component replication.

As a consequence, the *Distributed Shared Data* object (Figure 4.12) provides a single data element, and the *read* interface is the same as in the simplest form of the object. However, the *write* interface is different, as it no longer updates the value in the object. It simply requests the Replica Manager to disseminate that value (using the communication mechanisms provided by the Communication Manager[3]). Additionally, there is a third interface (*private_write*), which is used by the Replica Manager to update the value when it is delivered.

```
1:  when write (data):
2:     t_val := Replica_Manager.Request_Validity_Time(Writing_Task)
3:     Replica_Manager.Request_Dissemination(Data,t_val)

4:  when read:
5:     newest_data := null
6:     t_val := 0
7:     t_rel := Replica_Manager.Request_Release_Time(Reading_Task)
8:     for all i in Data_Buffer loop
9:        if t_val(i) < t_rel and t_val(i)> t_val then
10:          newest_data := Data(i)
11:          t_val := t_val(i)
12:       end if
13:    end loop
14:    return newest_data

15: when private_write (data,t_val):
16:    DataBuffer := DataBuffer ∪ (data,t_val)
```

Figure 4.13. *Deterministic Distributed Shared Data* object

The third situation is when an object is simultaneously used by tasks in different nodes (distribution requirements) and it requires deterministic execution (replication requirements). The *Deterministic Distributed Shared Data* object (Figure 4.13) has the buffer and validity times of the *Deterministic* object, and its *write* interface must also request the validity time of the value.

However, as the *Distributed* object, it does not add the value to the buffer, but request its dissemination by the Replica Manager. A further interface (*private_write*) is also available, enabling the Replica Manager to update the value when it is delivered by the atomic multicast mechanism. The *read* interface is the same as the one in the *Deterministic* object.

[3] The Communication Manager provides different atomic multicast mechanisms, for different assumptions and behaviours in case of faults. The actual mechanism used is configuration dependent.

4.4.3. Interaction Between Groups

When tasks belonging to different components interact (Figure 4.14), there is the need to consolidate values or events between the component replicas. As a set of replicated components is defined as a group, this type of interaction is defined as inter-group, instead of inter-component.

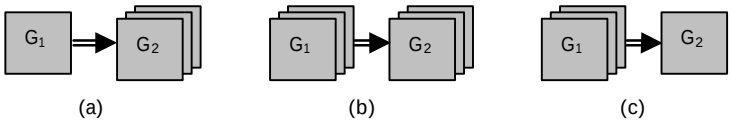

Figure 4.14. Inter-Group interaction: 1-to-many (a), many-to-many (b) or many-to-1 (c)

In the case where a group is releasing a task in another group (Figure 4.15), the Replica Manager must consolidate the release proposals from the replicas. If a release with data is requested, this data must also be consolidated. A similar approach to the *Distributed Release Event* is used.

```
Inter-Group Release Event Proxy Object

1:   when release:
2:      Replica_Manager.Propose_Release_Event

Inter-Group Release Event Receive Object

3:   when wait:
4:      Task_Suspend

5:   when private_release:
6:      Suspended_Task_Resume

Inter-Group Release Event with Data Proxy Object

7:   when release(data):
8:      Replica_Manager.Propose_Release_Event(data)

Inter-Group Release Event with Data Receive Object

9:   when wait:
10:     Task_Suspend
11:     return Obj_Data

12:  when private_release(data):
13:     Obj_Data := data
14:     Suspended_Task_Resume
```

Figure 4.15. *Inter-Group Release Event* objects

When a task requests a release in another group, the *Inter-Group Release Event Proxy* object forwards this request to the Replica Manager. In the other side, tasks wait in the corresponding *Receive* object, which is released by the Replica Manager.

The Replica Manager is also responsible for determining the release time of the sporadic task being released. However, this is simplified due to the properties of the Communication Manager, which guarantees a common delivery time of consolidated values in every node (Pinho and Vasques, 2001c). This common time can be taken as the release time of the sporadic task.

If the releasing task is in a non-replicated component, it is not necessary to propose the release, but only to request it. The Replica Manager is responsible for detecting such cases and for bypassing the regular behaviour, in order to optimise the system.

For the case of the *Shared Data* objects, the *Inter-Group Shared Data* object (Figure 4.16) is responsible for transparently managing the consolidation of the value being written, and at the same time for maintaining the deterministic behaviour by determining its validity time. This approach is similar to that used for the *Deterministic Distributed* object, except that the Replica Manager is requested to propose a value and not to disseminate it. Also, it is not necessary to request the validity time of the message at the source, since, as in the case of the *Inter-Group Release Event*, the common delivery time of the Communication Manager can be used for the validity time of the data being written.

```
1:  when write (data):
2:     Replica_Manager.Propose_Value(Data)

3:  when read:
4:     newest_data := null
5:     t_val := 0
6:     t_rel := Replica_Manager.Request_Release_Time(Reading_Task)
7:     for all i in Data_Buffer loop
8:        if t_val(i) < t_rel and t_val(i)> t_val then
9:           newest_data := Data(i)
10:          t_val := t_val(i)
11:       end if
12:    end loop
13:    return newest_data

14: when private_write (data,t_val):
15:    DataBuffer := DataBuffer ∪ (data,t_val)
```

Figure 4.16. *Inter-Group Shared Data* object

Two special cases must be considered for the *Inter-Group Shared Data* object. The first case is when the writer task is in a non-replicated component and thus it is not necessary to propose a value, but only to disseminate it. The second case is when the reader task is in a non-replicated component and thus it is not necessary to determine a data validity time, as there is no reader replication. Once more, the Replica Manager is responsible for detecting such cases and for bypassing the regular behaviour.

4.4.4. Interaction with the Soft Real-Time Subsystem

The interconnection between the HRTS and the SRTS must be supported by appropriate mechanisms for the transfer of information between both subsystems. Data being

transferred from the HRTS to the SRTS does not present a major problem, since it is assumed that this information has a higher reliability level, as it is considered that hard real-time applications have been designed to achieve higher reliability levels. Nevertheless, if this data comes from replicated components, the appropriate consolidation must be performed.

Conversely, the reliability of the data from the SRTS may not be high enough to be directly used in the system. If possibly erroneous values are expected, the received data must be filtered. As the definition of what it is erroneous is application-specific (since it depends on the semantics and not just on the syntax), a generic mechanism cannot be used. Additionally, if the data is to be provided to replicated components, it must be disseminated using the appropriate mechanisms (*e.g.*, atomic multicasts).

Additionally, mechanisms for the communication between both subsystems are expected to depend on the actual platform of a particular instance of the architecture (mainly on its operating system). Therefore it is not possible to implement a generic mechanism for such purpose, and a model for such interconnection is provided. Figure 4.17 presents this model for the case of data received from the SRTS (data sent to the SRTS is similar, just flows in the opposite direction). The application has to supply a specific interconnection task, which reads the value from the HRTS/SRTS interface, performs the necessary filtering, and interacts with the rest of the system through one of the available objects (it can write to a shared data object, or release a sporadic task).

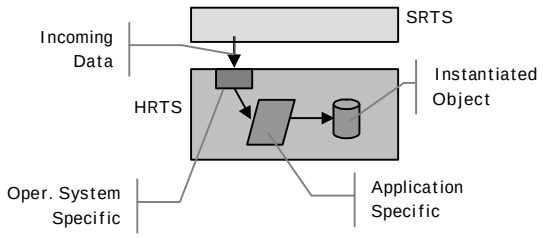

Figure 4.17. Model of interconnection with the SRTS

Note that, if fault tolerance is to be achieved, the interconnection with the Soft Real-Time Subsystem should also be replicated. In order to achieve deterministic execution, every external interaction with the system must have a common time reference. As a consequence, it is necessary that the interface task is a component by itself, in order to interact with the remaining system with consolidated values and times (data validity time and/or sporadic task release time).

4.4.5. Interaction with the Controlled System

Interconnection with the controlled system is performed through the use of sensors and actuators, connected to specific nodes. As the interconnection with these sensors and actuators is application and platform specific, the model for this interconnection is the same as in the interconnection with the SRTS, where application tasks are responsible for interconnecting with the devices, and inserting their value (or event) in the system.

Note that output actuator agreement may be made either in the computational system or by mechanical or electronic voting on the result. In the first case, it means that there is a single task responsible for interconnecting with a single actuator. Thus, the system relies on the reliability of both the task (and the node where it is allocated) and the actuator. Although the architecture itself does not forbid such configuration, it is considered that it implies assumptions not provided by COTS components. Voting outside the computational system provides much better coverage of the COTS components failure assumptions (fail-uncontrolled). The way that such agreement is made outside the computational system is, however, outside of the scope of the generic architecture.

4.4.6. Configured Application Example

Section 4.4.1 presented a simple application to exemplify how the Object Repository could be used during the development phase. The same example is re-visited in this section, in order to exemplify how distributed/replicated applications are modified by the configuration phase.

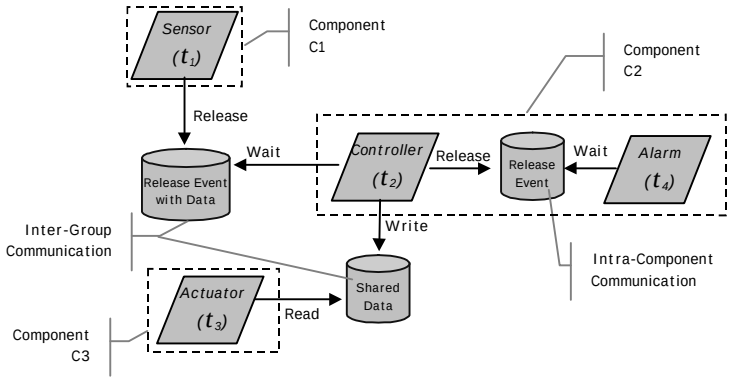

Figure 4.18. Application configuration

Figure 4.18 presents the configuration of the application. As noted in Section 4.4.5, since tasks *Sensor* and *Actuator* interact with the controlled system, they must be configured as components. Therefore, component C_1 is constituted by task *Sensor* (t_1) and component C_3 by task *Actuator* (t_3). Component C_2 encompasses tasks *Controller* (t_2) and *Alarm* (t_4). For the purpose of this example, a simplified replication is considered. Obviously, in order to tolerate fail-uncontrolled behaviour of the replicas, it would be necessary to use *2*f+1* replicas to tolerate *f* faults.

Figure 4.19 presents the allocation of the application tasks over the HRTS nodes. Node *1* is configured with components C_1 and C_2, while node *2* is configured with a replica of component C_1, with component C_3 and with a task of a replica of component C_2. Finally, node *3* is configured with the other task of the replica of component C_2, and with a replica of component C_3.

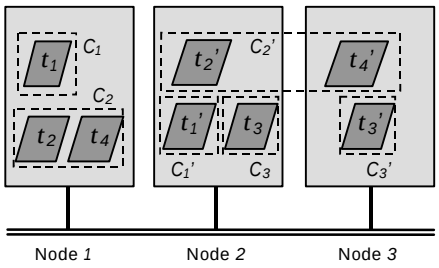

Figure 4.19. Node configuration

Figure 4.20 presents the program configured to execute in node *1*. The *Release Event with Data* object of Figure 4.7 (Section 4.4.1) is replaced by its *Inter-Group* equivalent (lines 1 and 2), since tasks *Sensor* and *Controller* are in different components. The same occurs with the *Shared Data* object between *Controller* and *Actuator* tasks. The *Release Event* object used for the interaction between *Controller* and *Alarm* tasks is replaced by a *Deterministic Release Event* object, since it is related to an intra-component interaction and, as it is replicated, the release time of the *Alarm* task must also be recorded. Note that application tasks are not changed since the new objects have the same interface as the ones in Section 4.4.1. The program in this node does not provide the *Actuator* task, since no replica of component C_3 is allocated to the node.

```
1:  package Device_Event is new
          Object_Repository.Inter_Group.Release_Event_With_Data(
                                                  Device_Data);
2:  Device_Event_Obj: Device_Event.Release_Event_With_Data_Obj;

3:  package Control_Shared_Data is new
       Object_Repository.Inter_Group.Shared_Data(Control_Data);
4:  Control_Data_Obj: Control_Shared_Data.Shared_Data_Obj;

5:  package Alarm_Event is new
       Object_Repository.Intra_Comp.Deterministic_Release_Event;
6:  Alarm_Obj: Alarm_Event.Release_Event_Obj;

7:  task Sensor;      -- no changes

20: task Controller;  -- no changes

    -- no Task Actuator

47: task Alarm;       -- no changes
```

Figure 4.20. Node *1* program

Figure 4.21 presents the program for node *2*. In this node there is no *Alarm* task, and as the replica of component C_2 is spread between nodes *2* and *3*, a *Distributed Release Event Proxy* object is used. In node *3* (Figure 4.22) the counterpart *Receive* object is used. As in this latter node there is only the *Actuator* and *Alarm* tasks, it is not necessary to create any object responsible for the interaction between *Sensor* and *Controller* tasks.

```
1:   package Device_Event is new
       Object_Repository.Inter_Group.Release_Event_With_Data(
                                                    Device_Data);
2:   Device_Event_Obj: Device_Event.Release_Event_With_Data_Obj;

3:   package Control_Shared_Data is new
        Object_Repository.Inter_Group.Shared_Data(Control_Data);
4:   Control_Data_Obj: Control_Shared_Data.Shared_Data_Obj;

5:   package Alarm_Event is new
        Object_Repository.Intra_Comp.Distributed_Release_Event;
6:   Alarm_Obj: Alarm_Event.Proxy_Release_Event_Obj;

7:   task Sensor;         -- no changes

20:  task Controller;     -- no changes

36:  task Actuator;       -- no changes

     -- no Task Alarm
```

Figure 4.21. Node *2* program

```
1:   package Control_Shared_Data is new
        Object_Repository.Inter_Group.Shared_Data(
                                    Control_Data);
2:   Control_Data_Obj: Control_Shared_Data.Shared_Data_Obj;

5:   package Alarm_Event is new
        Object_Repository.Intra_Comp.Distributed_Release_Event;
6:   Alarm_Obj: Alarm_Event.Receive_Release_Event_Obj;

     -- no Task Sensor

     -- no Task Controller

36:  task Actuator;    -- no changes

47:  task Alarm;       -- no changes
```

Figure 4.22. Node *3* program

4.5. HRTS Replica Manager

The Replica Manager layer (Figure 4.23) is intended to support the proposed task interaction objects implementing the main processing of the replication and distribution mechanisms.

The Property Recorder module is the database of the Replica Manager. It records both the structure and information of tasks and components. Repository objects use this module to query and/or change the release times of tasks, when a deterministic

behaviour is required. This module is also used by the Replication Support and Application Support modules, to query and record tasks' release times and application configuration information.

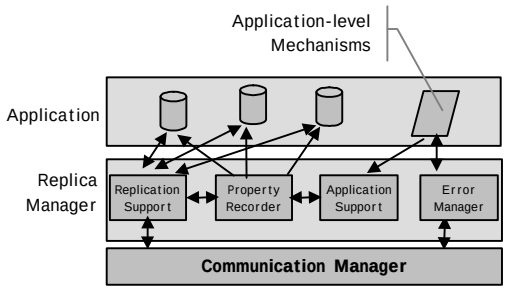

Figure 4.23. Replica Manager structure

The Replication Support module provides the required mechanisms for supporting replicated task interaction. It also interfaces the repository objects with the communication subsystem, thus it also receives/sends data values and events both from/to the objects and from/to the Communication Manager.

The Application Support module provides applications executing in the HRTS with the required support for the recording of periodic tasks' release times and for allowing components to be shutdown, silenced or activated.

The Error Manager module keeps a record of detected errors in the node (and in the system, if configured to disseminate error detection). This module also provides mechanisms for notification of errors, which can be used by applications error recovery procedures.

4.5.1. Property Recorder Module

The Property Recorder module is responsible for storing all the information (Table 4.1) needed to guarantee the correct behaviour of the replication/distribution framework. In this module, two different categories of information are stored. The application configuration information, which is off-line knowledge, and is constituted by the component structure, task information and network information. This type of information is similar in every node, since it is related to the global knowledge of the supported applications, which is off-line knowledge.

The application execution information cannot be defined off-line, since it is constituted by the tasks' release times and the state of the application components (active, silenced, shutdown). The Property Recorder at each node only stores the information related to the components and tasks that execute in the node. The Replication Support module is responsible for, when necessary, forwarding such local information to the other nodes.

Table 4.1. Application information

Application Configuration	Component Structure:	Task and object identifiers of each component
	Task Information:	Worst- and best-case execution time (may include internal computations)
	Network Information:	Message streams worst-case delivery time
Application Execution	Component Structure:	Component state
	Task Information:	Release time of tasks

Table 4.2 presents the interface provided by the Property Recorder. This interface is used by the repository objects (Section 4.4) and also by the Replication Support and Application Support modules of the Replica Manager. Note that there are two different interfaces to record the release time of a task. The first one is to be used when both released and releasing tasks are in the same node, thus the Property Recorder has the necessary information to determine the release time (release time and best-case execution time of the releasing task).

Table 4.2. Property Recorder interface

Application Configuration	Query_Released_Task(Obj_Id)
	Query_Component_Replication_Degree(Comp_Id)
	Query_Component_Id (Task_Id)
	Query_Component_Id (Obj_Id)
	Query_Group(Obj_Id)
	Query_Source_Group(Obj_Id)
	Query_Dest_Groups(Obj_Id)
Application Execution	Query_Message_Validity_Time(Task,Msg)
	Record_Task_Release_Time(Task_Id)
	Record_Task_Release_Time(Task_Id, Time)
	Query_Task_Release_Time(Task_Id)
	Query_Component_State(Comp_Id)
	Record_Component_State(Comp_Id, State)

When distributed release events are used, this information is only available at the source node, in spite of the release time being required at the destination node. Therefore, *Query_Task_Release_Time* is used at the source node to determine the release time of the task. Such release time is then sent through the network, in order to be stored at the destination node, using the second *Record_Task_Release_Time* interface.

Moreover, for the case of activating a component, the Application Support module also needs to record the release time of the periodic tasks, also requiring the second interface.

4.5.2. Replication Support Module

The Replication Support module is the core of the Replica Manager layer. This module is responsible for implementing the replication and distribution mechanisms, and also for the interconnection with the communication subsystem through the interface provided by the Communication Manager.

This interface allows the atomic multicast of messages to destination groups, the notification of message reception, and also the atomic multicasting of messages within the same group. Using a group communication interface to the communication subsystem allows an easier management of component replication, and also simplifies the necessary adaptations, if the communication subsystem or the Communication Manager are changed.

This group communication interface is specified in terms of *Groups* and *Group Objects*. Several different calls can be made to the same group, referring to different interaction objects. Therefore, this interface specifies which group is being addressed and, inside the group, which object is being addressed. Table 4.3 presents the syntax of the calls that can be made to the interface, and also the syntax of the necessary handler to receive messages.

Table 4.3. Communication Manager interface

To the Communication Manager	Multicast(Message, Sender_Group_Id, Receiver_Group_Array, Receiver_Obj_Id)
	Consolidated_Multicast(Message,Sender_Group,Receiver_Group_Array, Receiver_Obj_Id)
From the Communication Manager	Receive(Message, From_Group, To_Group, To_Obj, Deliver_Time)

The *Multicast* call allows the atomic multicast of a message to a set of groups, without requiring any type of consolidation from the communication system. The *Consolidated_Multicast* call is to be used when consolidation is required between the replicas of the sending component.

Messages between distributed elements within the same replica can use the *Multicast* call, using the same group for *Sender* and *Receiver* groups. Note that using a multicast call the same message can be sent or proposed to a set of groups. This is necessary, since the same interaction object can be used by a set of components, thus, when replication is considered, by a set of groups. It is expected that most of the time the writing group will also be one of the receiving groups, if the same object is used for reading and writing in a component. If it is to be sent or proposed to a single group, then a *Receiver_Group_Array* with a cardinality of one can be used.

Concerning the support to the interaction objects, when a release event is requested (Figure 4.24) it is necessary to record the release time of the released task. When a release event is to be forwarded (Figure 4.24), it is necessary to obtain from the Property Recorder the release time of the sporadic task being released, as such value must be sent together with the event message. When data is to be disseminated inside a component (Figure 4.25), its validity time must also be disseminated if the source component is replicated (lines 5 to 8).

```
1:  when Request_Release_Event
2:     Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
3:     Property_Recorder.Record_Task_Release_Time(Task_Id)
4:     Object(Obj_Id).private_release

5:  when Forward_Release_Event
6:     Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
7:     Message := Msg_Type(Event) ∪
            Property_Recorder.Query_Task_Release_Time(Task_Id)
8:     This_Group := Property_Recorder.Query_Group(Obj_Id)
9:     Communication_Manager.Multicast( Message, This_Group,
                                        This_Group, Obj_Id)

10: when Forward_Release_Event(data)
11:    Task_Id := Property_Recorder.Query_Released_Task(Obj_Id)
12:    Message := Msg_Type(Event) ∪ data ∪
            Property_Recorder.Query_Task_Release_Time(Task_Id)
13:    This_Group := Property_Recorder.Query_Group(Obj_Id)
14:    Communication_Manager.Multicast( Message, This_Group,
                                        This_Group, Obj_Id)
```

Figure 4.24. Support for *Intra-Component Release Events*

```
1:  when Request_Dissemination(Data)
2:     Message := Msg_Type(Data) ∪ Data
3:     This_Group := Property_Recorder.Query_Group(Obj_Id)
4:     Communication_Manager.Multicast( Message, This_Group,
                                        This_Group, Obj_Id)

5:  when Request_Dissemination(Data, t_val)
6:     Message := Msg_Type(Data_and_Validity) ∪ Data ∪ t_val
7:     This_Group := Property_Recorder.Query_Group(Obj_Id)
8:     Communication_Manager.Multicast( Message, This_Group,
                                        This_Group, Obj_Id)
```

Figure 4.25. Support for *Intra-Component* data dissemination

To propose release events or data values (Figures 4.26 and 4.27), it is necessary to query the Property Recorder to obtain source and destination group identifiers. Moreover, if the source component is replicated, it is necessary to use the *Consolidated_Multicast* interface to consolidate the outputs from the different replicas.

```
1:   when Propose_Release_Event
2:      comp := Property_Recorder.Query_Component_Id(Obj_Id)

3:      if Property_Recorder.Query_Component_State(comp)
              not Silenced then
4:         Message := Msg_Type(Event)
5:         Source_Group :=
              Property_Recorder.Query_Source_Group(Obj_Id)
6:         Dest_Groups :=
              Property_Recorder.Query_Dest_Groups(Obj_Id)
7:         degree :=
              Property_Recorder.Query_Comp_Replication_Degree(comp)

8:         if degree = 1 then
9:            Communication_Manager.Multicast( Message,
                                               Source_Group,
                                               Dest_Groups,
                                               Obj_Id)
10:        else
11:           Communication_Manager.Consolidated_Multicast(
                                               Message,
                                               Source_Group,
                                               Dest_Groups,
                                               Obj_Id)
12:        end if
13:     end if

14:  when Propose_Release_Event(data)
15:     comp := Property_Recorder.Query_Component_Id(Obj_Id)

16:     if Property_Recorder.Query_Component_State(comp)
              not Silenced then
17:        Message := Msg_Type(Event_Data) ∪ data
18:        Source_Group :=
              Property_Recorder.Query_Source_Group(Obj_Id)
19:        Dest_Groups :=
              Property_Recorder.Query_Dest_Groups(Obj_Id)
20:        degree :=
              Property_Recorder.Query_Comp_Replication_Degree(comp)

21:        if degree = 1 then
22:           Communication_Manager.Multicast( Message,
                                               Source_Group,
                                               Dest_Groups,
                                               Obj_Id)
23:        else
24:           Communication_Manager.Consolidated_Multicast(
                                               Message,
                                               Source_Group,
                                               Dest_Groups,
                                               Obj_Id)
25:        end if
26:     end if
```

Figure 4.26. Support for *Inter-Group Release Events*

```
1:   when Propose_Value(data)
2:      comp := Property_Recorder.Query_Component_Id(Obj_Id)

3:      if Property_Recorder.Query_Component_State(comp)
              not Silenced then
4:         Message := Msg_Type(Data) ∪ data
5:         Source_Group :=
              Property_Recorder.Query_Source_Group(Obj_Id)
6:         Dest_Groups :=
              Property_Recorder.Query_Dest_Groups(Obj_Id)
7:         degree :=
              Property_Recorder.Query_Comp_Replication_Degree(comp)

8:         if degree = 1 then
9:            Communication_Manager.Multicast( Message,
                                               Source_Group,
                                               Dest_Groups,
                                               Obj_Id)
10:        else
11:           Communication_Manager.Consolidated_Multicast(
                                               Message,
                                               Source_Group,
                                               Dest_Groups,
                                               Obj_Id)
12:        end if
13:     end if
```

Figure 4.27. Support for *Inter-Group* data dissemination

Note that both the support for inter-group release events and data dissemination query the Property Recorder module, in order to avoid communication if the component is silenced.

The Replica Manager, in order to be notified of a message reception and its associated delivery (or consolidation) time, must use the *Receive* handler (Figure 4.28). The Communication Manager protocols guarantee that the timestamp is the same in all the receiving nodes (Pinho and Vasques, 2001c). Small differences may occur, due to the different interleaving of the low-level communication handlers and also due to different clock values, but such differences are upper-bounded.

Note that, although the message may have been sent (or proposed) to a set of groups, the *Receive* handler receives messages to a single group. Messages sent to a set of groups have already been consolidated (if needed) and are individually delivered to each group by the Communication Manager.

This *Receive* handler is also used to prevent sporadic tasks from being released, if the related component has been shutdown (lines 23 and 30). Note that it is not necessary to prevent the release of sporadic tasks internal to a component, since, by preventing the release of both sporadic tasks released by other components and periodic tasks within the component, internal sporadic tasks will not be released.

```
1: when Receive(Message, Obj_Id, t_deliver)
2:    if Intra_Component (Obj_Id) then -- internal to a group
3:       case Msg_Type(Message) is
4:          Event =>
5:             Task_Id := Property_Recorder.Query_Released(Obj_Id)
6:             Property_Recorder.Record_Task_Release_Time(Task_Id,
                            Released_Time(Message))
7:             Object(Obj_Id).private_release
8:          Event_Data =>
9:             Task_Id := Property_Recorder.Query_Released(Obj_Id)
10:            Property_Recorder.Record_Task_Release_Time(  Task_Id,
                               Released_Time(Message))
11:            Object(Obj_Id).private_release(Data(Message))
12:         Data =>
13:            Object(Obj_Id).private_write(Data(Message))
14:         Data_and_Validity =>
15:            Object(Obj_Id).private_write(Data(Message),
                               Validity_Time(Message))
16:      end case
17:   else
18:      case Msg_Type(Message) is
19:         Event =>
20:            Task_Id := Property_Recorder.Query_Released(Obj_Id)
21:            Property_Recorder.Record_Task_Release_Time(Task_Id,
                                                      t_deliver)
22:            comp :=
                  Property_Recorder.Query_Comp_Id(Released_Task)
23:            if Property_Recorder.Query_Component_State(comp)
                     not Shutdown then
24:               Object(Obj_Id).private_release
25:            end if
26:         Event_Data =>
27:            Task_Id := Property_Recorder.Query_Released(Obj_Id)
28:            Property_Recorder.Record_Task_Release_Time(Task_Id,
                                                      t_deliver)
29:            comp :=
                  Property_Recorder.Query_Comp_Id(Released_Task)
30:            if Property_Recorder.Query_Component_State(comp)
                     not Shutdown then
31:               Object(Obj_Id).private_release(Data(Message))
32:            end if
33:         Data =>
34:            comp :=  Property_Recorder.Query_Component_Id(Obj_Id)
35:            degree :=
                  Property_Recorder.Query_Comp_Replicat_Degree(comp)
36:            if degree = 1 then
37:               Object(Obj_Id).private_write(Data(Message))
38:            else
39:               Object(Obj_Id).private_write(Data(Message),
                            Validity_Time(Message))
40:            end if
41:      end case
42:   end if
```

Figure 4.28. *Receive* handler specification

4.5.3. Application Support Module

The Application Support module provides some support to applications executing in the HRTS. As periodic tasks are not released by the support software, but by the operating system, there is the need for an appropriate interface (Figure 4.29). Also, the support software needs to record the release time of periodic tasks and to prevent them from being released if the related component has been shutdown. A solution to this problem is to restrict the application to use a Support Software interface to the operating system, in order to request its next release.

```
1: when Request_Periodic(next_release_time)
2:    Property_Recorder.Record_Task_Release_Time(This_Task,
                                                 next_release_time)
3:    comp := Property_Recorder.Query_Component_Id(This_Task)
4:    if Property_Recorder.Query_Component_State(comp)
            not Shutdown then
5:        OS_Specific_Release(This_Task,next_release_time)
6:    else
7:        Task_Suspend
8:        next_release_time :=
             Property_Recorder.Query_Task_Release_Time(This_Task)
9:        OS_Specific_Release(This_Task,next_release_time)
10:   end if
```

Figure 4.29. Periodic task support

This interface records the release time of the task and only schedules it for execution if the component has not been shutdown. If the component has been shutdown, the task is suspended and will not be released again. Only after restarting the component, the task will be re-activated, the Property Recorder will be queried about its new release time, and the task is re-scheduled. Note that the application must be aware that *next_release_time* may have changed since the last execution of the task.

```
1:   when Shutdown_Component(Comp_Id)
2:      Property_Recorder.Record_Component_State(Comp_Id, Shutdown)

3:   when Silence_Component(Comp_Id)
4:      Property_Recorder.Record_Component_State(Comp_Id, Silenced)

5:   when Start_Component(Comp_Id,
                        Tasks_Release_Times[Task_Id, Release_time])
6:      for all Task_Id in Tasks_Release_Times loop
7:         Property_Recorder.Record_Task_Release_Time(Task_Id,
                                                      Release_Time)
8:         Wakeup_Task (Task_Id)
9:      end loop
10:     Property_Recorder.Record_Component_State(Comp_Id, Active)
```

Figure 4.30. Reconfiguration support

Finally, there is the possibility to reconfigure the system (Figure 4.30), either due to mode changes, or due to error recovery situations. The reconfiguration support allows components to be shutdown, silenced and/or activated. In this last case, it is necessary to provide new release times for the component tasks, which are stored in the Property Recorder before restoring the suspended tasks. The Replication Support module uses this configuration information to prevent the release of the components' tasks or their interaction with other components.

4.5.4. Error Manager Module

The goal of the Error Manager module is to record (and possibly notify) errors occurring in the communication and consolidation of values. It also provides mechanisms by which application or system error recovery methods can be notified of error detection in order to take actions to shutdown or silence components. Table 4.4 presents the interfaces provided by such module.

Table 4.4. Error Manager interface

From the Communication Manager	Record_Error(Error_Information)
To Application Error Recovery procedures	Query_Error_Information
	Error_Notification

As the Communication Manager is responsible for the atomic multicasts and for the consolidation of replicated components' outputs, it is up to this layer to detect possible errors in the system. It notifies the Error Manager module, which, in addition to storing the error information, may execute two (non-exclusive) actions: it can notify an application-level error recovery module; or it can disseminate this error detection throughout the system. This error information is related to the detectable errors in the communication system and on the consolidation of replicated outputs.

In the consolidation phase, component errors may be detected if a component has not proposed a value (omission fault) or the proposed value was rejected by the *decide* [4] function of the Communication Manager *Consolidate* protocol (Pinho and Vasques, 2001c). Consolidation-related errors may occur when the *Consolidate* protocol is unable to consolidate values, due to the violation of the failure assumptions. Although this should never happen, it is important to define the actions to be taken in the case of incorrect failure assumptions.

Communication-related errors occur when an error is detected in the network. As the Communication Manager is targeted to the Controller Area Network (CAN) (ISO, 1993), the possible errors in the network are inconsistent message duplicates and omissions (Rufino *et al.*, 1998). Although these are tolerated by the provided *Atomic Multicast* protocols (Pinho and Vasques, 2001c), they indicate network problems, which

[4] The *decide* function is application-specific, having the possibility of rejecting proposed values.

may bring down the system. Moreover, the occurrence of inconsistent message omissions indicates that there was, at least, one node crash.

In addition to recording errors, the module can perform two different actions. One action is to disseminate the error detection throughout the system. Therefore, if the Support Software is configured for dissemination, the module can request from the Communication Manager a specific message to be sent to the other Error Managers in the system. The module may also interact with an application or system error recovery module to notify the error detection. This can be performed either by periodically querying the Error Manager module, or by requesting the notification of a sporadic task. Error recovery mechanisms can use it to reconfigure the system.

4.6. Summary

This chapter presented the framework provided by the DEAR-COTS Hard Real-Time Subsystem for the support of replicated software components. This support relies on the structuring of applications in software components, which can then be replicated, and on a repository of generic task interaction objects that are used by fault-tolerant real-time applications.

The Object Repository is used during the development and configuration phases, and provides a set of generic objects with different capabilities. These objects provide the generic and transparent approach, thus are responsible for hiding from the application the low-level mechanisms for replication and distribution support. This allows applications to focus on the requirements of the controlled system rather than on the distribution/replication details.

The provided objects are supported by a middleware layer (the Replica Manager), which both shields the repository from changes in the lower-level communication mechanisms, and reduces the effort necessary for the creation of new task interaction generic objects.

Chapter 5

Fault-Tolerant Real-Time Communication

5.1. Introduction

In the DEAR-COTS Hard Real-Time Subsystem (HRTS), the communication infrastructure is responsible for guaranteeing timely and consistent multicast of information and for the consolidation of replicated components' outputs. The HRTS framework provides a software layer (the Communication Manager), which is the responsible for such mechanisms. This chapter presents this software layer, where, in order to address the problem of inconsistency in CAN message deliveries, a set of atomic multicast and consolidation protocols is proposed. Such set of protocols explores the CAN synchronous properties to minimise its run-time overhead. The model provided for the evaluation of the message streams' response time demonstrates that the real-time capabilities of CAN are preserved, since predictability of message transfers is guaranteed.

The chapter is largely drawn from (Pinho and Vasques, 2001a) and is structured as follows. Section 5.2 presents the requirements and failure assumptions imposed by the HRTS, for fault-tolerant real-time communication. The types of message exchanges required by the replication model of the framework are identified, and the failure assumptions are presented and justified.

The Communication Manager layer is then described in Section 5.3, with a special focus on the proposed set of atomic multicast and consolidation protocols. This layer presents a group communication interface to the higher levels of the framework, providing the required communication support. A set of atomic multicast protocols, with different failure assumptions, is provided, allowing the guarantee of consistency in CAN message transfers with a minimum overhead. Consolidation of replicated messages is achieved through the use of a protocol that does not require extra overhead. Message fragmentation and concatenation is also provided by means of a protocol that uses a range of consecutive CAN identifiers for each stream requiring fragmentation.

In Section 5.4 a set of pre-run-time schedulability conditions is presented, enabling the timing analysis of the supported communication protocols. These conditions are use to determine the delays required for the proper behaviour of the proposed protocols, and also the worst- and best-case response times of each message stream. Therefore, the proposed protocols guarantee consistency of CAN communication, whilst maintaining the real-time properties of CAN message transfers.

For a better understanding of these protocols, a numerical example is presented in Section 5.5. This example presents a strategy for the identification of which protocols to use according to the application configuration, and demonstrates that the protocols provide a suitable solution for the use of CAN as the communication infrastructure of the DEAR-COTS HRTS.

Finally, Section 5.6 presents a comparison with other approaches for atomic multicast in CAN that also rely on the software management of CAN inconsistencies. It is shown that the proposed approach minimises the overhead of protocol-related messages. This overhead, although still large, is the strictly necessary to cope with inconsistent message omissions using a software-based approach.

5.2. Communication Requirements

As the HRTS replication model considers the existence of active replication, there is the need to guarantee that replicas execute deterministically, that is, replicated tasks execute with the same data and timing-related decisions are the same in each replica (Powell, 1991). In order to provide deterministic execution of the replicated components, it must be guaranteed that all messages sent by correct components are delivered to all their recipients. Moreover, there must also be an all-or-none guarantee in the case of a message sent by an incorrect component: either all correct components deliver that message, or none of them deliver it. Furthermore, there is the need to agree in the order by which messages are delivered, and to consolidate messages from replicated components' outputs.

From the replication model presented in the previous chapter, it is clear that four types of message exchanges must be supported: *1-to-1*, *1-to-many*, *many-to-1* and *many-to-many*.

For *1-to-1 communication* (communication from a non-replicated component to another non-replicated component, or communication internal to a component) there is only the need for a reliable multicast, since order issues are not relevant, as there are no replicated receivers. However, when a result is to be disseminated to a group of replicated components (*1-to-many communication*), atomic multicast protocols (Hadzilacos and Toueg, 1993) must be used to guarantee that replicated receivers get the same information, in the same order.

When a group of replicated components receives a message from another group of replicated components (*many-to-many communication)*, an agreement must be performed. If an underlying atomic multicast mechanism is used to disseminate each value, then it is guaranteed that every receiver will have the same input values, and in the same order. The agreement decision can then be performed by a simple *Consolidate* protocol, which decides on one of the received values (or on some value computed from them).

The case of communication from a group of replicated components to a single component (*many-to-1 communication*) is a simplified version of the previous one. The receiving component has only to decide from the set of received inputs. The same *Consolidate* protocol can be used, but using only a reliable multicast from the replicated transmitters to the single receiver (since there are no order requirements).

5.2.1. Failure Assumptions

A synchronous distributed system is considered, where a fixed number of components exchange information through a synchronous communication channel. The use of a real-time network (CAN), together with the use of appropriate schedulability analysis techniques (Tindell *et al.*, 1995), allows the system to be considered synchronous.

In the assumed network model, temporary failures are a consequence of either bus errors or network interface (transceiver) errors. Such network failures have the following semantics:

- Bus error bursts never affect more than n_{bus} transmissions during an interval of analysis T_{bus}. This means that, even for the case of multiple sources of errors, the time interval during which the network is inaccessible is upper-bounded.
- Transceivers either behave correctly or crash after a given number of failures (n_{transc}), during an interval of analysis T_{transc}. This behaviour is guaranteed by the CAN protocol, since in the case of multiple errors, the node goes first into the *Error-Passive* state and then into the *Bus-Off* state.
- Multicasts fail either by inconsistent message omissions, or inconsistent message duplicates. All other errors are detected (and failed messages retransmitted) by the CAN built-in error detection and recovery mechanisms, with a sufficiently high probability (ISO, 1993).
- A single message can be disturbed by at most k_{dup} duplicates. As the probability of an inconsistent message duplicate is approximately 10^{-4} (the transmission of 2.87×10^7 messages per hour results in, at most, 2.84×10^3 duplicate messages (Rufino *et al.*, 1998)), it is not foreseen the necessity of k_{dup} being considered greater than 2.
- During a time interval T, greater than the worst-case delivery time of any message, at most one single inconsistent message omission occurs in the network. Considering the existence of 3.98×10^{-9} to 2.94×10^{-6} inconsistent message omissions per hour (Rufino *et al.*, 1998), the occurrence of a second omission error in a period T of, at most, several seconds has an extremely low probability.
- There are no permanent medium faults, such as the partitioning of the network. This type of faults must be masked by appropriate network redundancy schemes.

As the goal of the provided protocols is only to tolerate network-related faults, nodes are assumed to be fail-silent in what concerns communications, that is, it is assumed that all communication requests performed by any node are correct. It is also considered that protocol software does not fail by producing incorrect messages (either value or timing faults). These assumptions may be quite restrictive when considering other fault models, such as network partitioning, network interface controllers permanent faults or incorrect message requests, which may cause the network to become unreachable or messages to be incorrectly delivered.

A study performed in order to evaluate the behaviour of CAN networks in the presence of either bus or network interface errors (presented in the Annex) has demonstrated that CAN networks are not resilient to network interface permanent (or intermittent) faults. A faulty network interface can cause a sequence of (at the most) 16 erroneous messages, causing the network to become unreachable for all nodes for a large

period. Therefore, it is considered that more stringent failure assumptions can only be covered through the use of network redundancy approaches, with the provision of special-purpose hardware and/or software with memory protection schemes, in order to provide a fail-silent behaviour of the node.

Although it is not a goal of this work to discuss how network redundancy could be achieved in DEAR-COTS, two different approaches are considered to be of significance. The first one is to provide only the replication of the physical medium, through the use of schemes based on (Rufino *et al.*, 1999), and using the TCB to achieve fail-silent behaviour of the node, by precluding faulty nodes from sending incorrect communication requests. This approach can tolerate the partitioning of the network, but it can not preclude faulty nodes from contaminating the network.

Another more complex approach is to use a scheme similar to the one presented in (Hilmer *et al.*, 1998), where redundant communication channels are provided, with redundant network interfaces in each node. In this approach, also the HRTS Support Software could be replicated (using memory separation schemes). Detection of communication failures, and redundancy management can be performed by specialised hardware, however this would defeat the COTS-based approach of DEAR-COTS. It is considered that the software management of network redundancy is the best approach for DEAR-COTS, such as in the soft fail-silent node of Delta-4 (Powell, 1991).

5.3. Communication Manager

The structure of the proposed Communication Manager intended to support fault-tolerant real-time communication in CAN networks is presented in Figure 5.1.

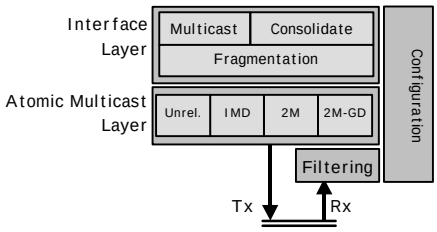

Figure 5.1. Communication Manager structure

The Filtering module allows that only nodes registered to receive a particular message stream will process messages related to that stream, decreasing the number of messages in error situations. The Atomic Multicast layer provides a set of multicast protocols, with different failure assumptions and different behaviours in the case of errors. These protocols range from an unreliable one, to a protocol that guarantees delivery even in the presence of inconsistent message omissions.

The Interface layer provides a group abstraction to the upper layers, in order to abstract them from the implementation of communication protocols and from membership and location issues. Consolidation of replicated components is provided in the Consolidate module. By using atomic multicasts, it is guaranteed that every

replicated component receives the same set of messages in the same order. Therefore, it is only required the existence of a consolidation protocol at the receiver side.

Fragmentation and concatenation of messages is also provided in this layer, after atomic multicast delivery of messages. The used approach is by allocating a range of consecutive CAN identifiers for each message stream requiring fragmentation. These fragments are concatenated at the receiver side.

The Configuration module is responsible for storing all the information required for the correct functioning of the protocols. This information concerns the particular identifiers a node is registered to receive, which particular atomic multicast protocol to use when a multicast request is received, the fragment information and the information required to consolidate replicated messages.

As the goal of this work is to support hard real-time applications, the Communication Manager considers the existence of a fixed set of processing units, and thus, the full set of message streams and their characteristics (periodicity, size, and replication) are previously known. This off-line knowledge is required in order to allow the use of state-of-the-art response time analysis to provide real-time guarantees to the supported applications.

Although a group communication interface is provided to the upper layers, a group communication approach is not considered at the communication layer level. This allows not to burden the CAN identifier field with source and destination information, particularly since CAN multicast capabilities already provide location transparency. Nevertheless, a group communication support is provided to the upper levels, so that these upper layers are only loosely dependent of the communication infrastructure used. This latter interface allows the atomic multicast of messages to destination groups, the notification of message reception, and also the atomic multicasting of messages within the same group. Using a group communication interface to the communication subsystem, allows an easier management of component replication, and also simplifies the necessary adaptations, if the communication subsystem or the Communication Manager are changed.

5.3.1. Communication Manager Interface

The Interface layer provides the group abstraction to the upper layer (Replica Manager). This group communication interface is specified in terms of *Groups* and *Group Objects*. Several different calls can be made to the same group, referring to different interaction objects. Therefore, this interface specifies which group is being addressed and, inside the group, which object is being addressed (the triple <*Sender_Group, Receiver_Groups, Obj_Id*> is referred as the *Message Address*). Table 5.1 presents the interface provided by the layer.

The *Multicast* call allows the Replica Manager to request the atomic multicasting of a message to a set of groups, without requiring any type of consolidation from the communication system. The *Consolidated_Multicast* call is to be used when consolidation is required between the replicas of the sending component. Since the protocol for consolidation of replicated messages is only required at the receiver side (see Section 5.3.4), at the transmitter side there is no need for a special mechanism.

Table 5.1. Communication Manager interface

From the Replica Manager	Multicast(Message, Message Address)
	Consolidated_Multicast(Message, Message Address)
To the Replica Manager	Receive(Message, From_Group, To_Group, To_Obj, Deliver_Time)
	Record_Error(Error_Information)
From the Atomic Multicast Layer	Deliver(Message_Id, Message, Deliver_Time)
	Error_Notification(Message_Id, Error_Type)
To the Atomic Multicast Layer	Protocol.Atomic_Multicast(Id, Data)

Figure 5.2 presents the specification of the *Multicast* call. It queries the Configuration module to map the *Message Address* to the corresponding CAN identifier range (since transmitting a message may require transmitting several low-level CAN frames), and also queries the multicast protocol to use. Afterwards, the appropriate atomic multicast is requested.

```
1:  when Multicast(Message, Sender_Group, Receiver_Group _Array,
                   Receiver_Obj_Id):

2:     Msg_Id_Range :=
          Configuration.Query_Message_Identifier_Range(
                                Sender_Group,
                                Receiver_Group _Array,
                                Receiver_Obj_Id)

3:     for Id in Msg_Id_Range loop
4:        Data := fragment_message(Message, fragment_number)
5:        prot := Configuration.Query_Message_Protocol(Id)
6:        case prot is
7:             Unrel => Unreliable.atomic_multicast(Id, Data)
9:             IMD   => IMD.atomic_multicast(Id, Data)
11:            2M    => 2M.atomic_multicast(Id, Data)
13:            2M_GD => 2M_GD.atomic_multicast(Id, Data)
15:       end case
16:    end loop
```

Figure 5.2. *Multicast* specification

Figure 5.3 presents the specification of the *Consolidated_Multicast* call. This interface simply forwards the request to the *Multicast* interface, since, at the sender side there is no difference between a consolidated multicast and a simple multicast. Consolidation of the results from replicated components is necessary only on the receiver side. Since component structure is off-line knowledge, the receiver side can query the Configuration module to know which messages are to be consolidated.

```
1:   when Consolidated_Multicast(Message, Sender_Group,
                                 Receiver_Group _Array,
                                 Receiver_Obj_Id):

2:      Multicast( Message, Sender_Group, Receiver_Group _Array,
                   Receiver_Obj_Id):
```

Figure 5.3. *Consolidated Multicast* specification

The Replica Manager, in order to be notified of a message reception and its associated delivery (or consolidation) time, uses the *Receive* handler that the Interface layer provides. When a message is delivered by the atomic multicasts or the consolidate protocols, the interface layer delivers such message individually to each group. Furthermore, the messages contain a timestamp (their delivery instant) which is then used by the Replica Manager for inter-group messages' management. Small differences may exist in this timestamp between different nodes, due to the different interleaving of the low-level communication handlers and also due to different clock values, but such differences are upper bounded.

The *Record_Error* interface is used by the Communication Manager to notify the Replica Manager of detected errors by the atomic multicast and consolidate protocols. When the consolidate protocol detects a lost message, or a rejected value, it notifies the upper Replica Manager layer. When the atomic multicast protocols detect inconsistent message omissions or duplicates, the interface layer is notified using the *Error_Notification* call, which is then forward to the Replica Manager.

The *Protocol.Atomic_Multicast* interface is used to request the sending of a multicast. Since the protocol to use is recorded in the Configuration module, the Interface layer requests the multicast directly to the related protocol (Figure 5.2).

5.3.2. Configuration Module

The Configuration module provides a database of off-line known information (Table 5.2) about the message streams of the system. It stores information about the mapping between the *Message Address* and CAN message identifiers (and vice-versa), and information about which protocols the messages use, together with the necessary information concerning the protocol-related delays (Sections 5.3.3 to 5.3.5).

Table 5.2. Message streams information

Message Information	Message Address to CAN Identifier range (and vice-versa)
	Message Address to Protocol
	Protocol-related delays
	Identifiers to receive
Consolidation Information	Replicated Components Messages

The Interface layer uses this information to decide which particular multicast to use when a transmission is requested. In order to allow fragmentation and consolidation of messages, this module also stores the information on which streams are subject to fragmentation, and the range of identifiers necessary. In order to allow the consolidation of replicated components' messages the module also stores the information about which messages are replicated outputs.

The Configuration module also stores the information of the particular identifiers that are to be received by the node. The Filtering module uses this information to query if the node is registered to process a particular message identifier, thus filtering the received messages.

5.3.3. Atomic Multicasts

The Atomic Multicast module provides several protocols, with different failure assumptions and different behaviours in the case of errors. The *IMD* (Inconsistent Message Duplicate) protocol provides an atomic multicast that just addresses the inconsistent message duplicate problem. The *2M* (Two Messages) protocol provides an atomic multicast addressing both inconsistent message duplicates and omissions, where messages are not delivered in an error situation. Finally, the *2M-GD* (Guaranteed Delivery) protocol is an improvement of the *2M* protocol, which guarantees message delivery, if at least one node has correctly received it. The *Unreliable* protocol is a simple multicast protocol that does not provide any guarantees.

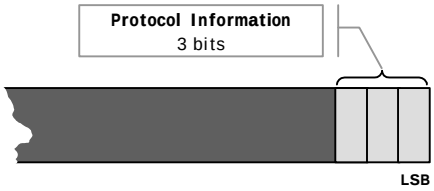

Figure 5.4. Identifier field

The proposed atomic multicast protocols use the less significant bits of the frame identifier (Figure 5.4) to carry protocol information, identifying the type of each particular message (Table 5.3) without interfering with the message criticality (defined by the most significant bits of the frame identifier).

These atomic multicast protocols provide the system developer with the possibility of trading efficiency for reliability, since they can be simultaneously used in the same system. The *IMD* protocol uses less bandwidth, but it does not cover the inconsistent omission failure assumption. On the other side, the use of protocols with higher assumption coverage (*e.g.* the *2M* protocol) introduces extra overheads in the system. Hence, streams with higher criticality may use protocols with higher assumption coverage, while streams with lower criticality may use protocols with lower overhead.

Knowing that CAN frames are simultaneously received in every node, the atomic multicast properties are guaranteed by delaying the delivery of a received frame for a specific (bounded) time. The approach is similar to the Δ-protocols (Cristian *et al.*,

1995), where, in order to obtain order, message delivery is delayed during a specific time (Δ). The difference is that, in the proposed approach delivery delays are evaluated on a stream by stream basis, increasing the system throughput, as messages are delayed according to their worst-case response times. It is assumed that clocks are approximately synchronised by the use of an appropriate protocol (Rodrigues *et al.*, 1998), guaranteeing both the deterministic execution of replicated components (Poledna *et al.*, 2000) and the correct evaluation of the delivery delays.

Table 5.3. Protocol information

Protocol Bits	Message Type	
0 0 0	*2M-GD* Protocol	Data Msg.
0 0 1		Confirmation Msg.
0 1 0		Retrans. Msg.
0 1 1	*2M* Protocol	Data Msg.
1 0 0		Confirmation Msg.
1 0 1		Abort Msg.
1 1 0	*IMD* Protocol	
1 1 1	*Unreliable* Protocol	

5.3.3.1. *IMD* Protocol

The *IMD* protocol (Figures 5.5 and 5.6) provides an atomic multicast that just addresses the inconsistent message duplicate problem. In order to guarantee that duplicates are correctly managed, every node when receiving a message marks it as unstable, tagging it with a $t_{deliver}$ stamp (current time plus a $d_{deliver}$ delay). In Figure 5.7, all receivers of the message, delay the respective delivery until $t_{deliver}$.

```
1:   when atomic_multicast (id, data):
2:      send (id, data)

3:   when sent_confirmed (id, data):   -- if the node also receives
                                       -- the message
4:      receivedMsgSet := receivedMsgSet ∪ msg(id,data)
5:      t_deliver(id) := clock + δ_deliver(id)

6:   deliver:
7:      for all id in receivedMsgSet loop
8:         if t_deliver(id) < clock then
9:              deliver( receivedMsgSet(id) )
10:        end if
11:     end loop
```

Figure 5.5. *IMD* protocol specification: transmitter

```
1:   when receive (id, data):
2:      if id ∉ receivedMsgSet then
3:         receivedMsgSet := receivedMsgSet ∪ msg(id,data)
4:         state(id) := unstable
5:      end if
6:      t_deliver(id) := clock + δ_deliver(id)

7:   deliver:
8:      for all id in receivedMsgSet loop
9:         if state(id) = unstable and t_deliver(id) < clock then
10:             deliver( receivedMsgSet(id) )
11:        end if
12:     end loop
```

Figure 5.6. *IMD* protocol specification: receiver

However, if a duplicate is received before $t_{deliver}$ (Figure 5.8), the duplicate is discarded and $t_{deliver}$ is updated (since in a node not receiving the original message $t_{deliver}$ refers to the duplicate). Therefore, it is guaranteed that every receiver will deliver the message with the same timestamp. For the transmitter (if it also delivers the message), as the CAN controller will only acknowledge the transmission when every node has received it correctly (no more retransmissions), there will be no duplicates. Thus, the transmitter can deliver the message after its $d_{deliver}$.

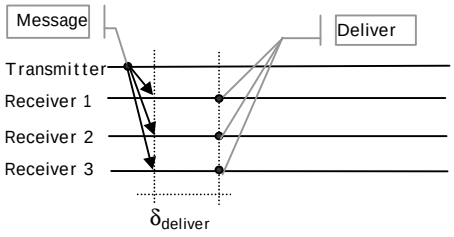

Figure 5.7. *IMD* protocol in an error-free situation

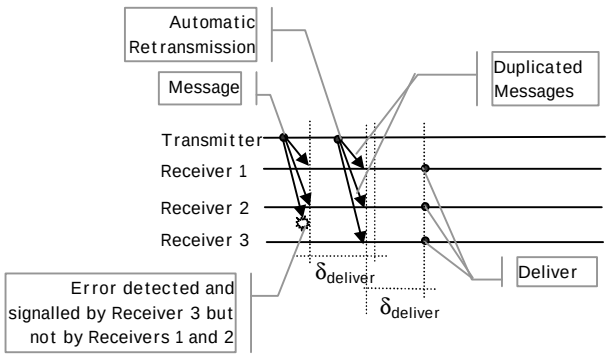

Figure 5.8. Inconsistent message duplicate

5.3.3.2. *2M* Protocol

The *2M* protocol (Figures 5.9 and 5.10) addresses both inconsistent message duplicates and inconsistent message omissions, guaranteeing that either all or none of the receivers will deliver the message. For the latter, not delivering a message is equivalent to a transmitting node crash before sending the message.

```
1:  when atomic_multicast (id, data):
2:     send (id, message, data)
3:     send (id, confirmation)

4:  when sent_confirmed (id, message, data): -- if also receives
5:     receivedMsgSet := receivedMsgSet ∪ msg(id,data)
6:     state(id) := confirmed
7:     t_deliver(id) := clock + δ_deliver(id)

8:  deliver:
9:     for all id in receivedMsgSet loop
10:       if state(id) = confirmed
                  and t_deliver(id) < clock then
11:           deliver( receivedMsgSet(id) )
12:       end if
13:    end loop
```

Figure 5.9. *2M* protocol specification: transmitter

```
1:  when receive (id, type, data):
2:     if type = message then
3:        if id ∉ receivedMsgSet then
4:           receivedMsgSet := receivedMsgSet ∪ msg(id,data)
5:           state(id) := unstable
6:        end if
7:        t_deliver(id) := clock + δ_deliver(id) -- duplicate update
8:        t_confirm(id) := clock + δ_confirm(id)
9:     elsif type = confirmation then
10:       state(id) := confirmed
11:    elsif type = abort then
12:       if id ∈ receivedMsgSet then
13:          receivedMsgSet := receivedMsgSet – msg(id)
14:       end if
15:    end if

16: deliver:
17:    for all id in receivedMsgSet loop
18:       if state(id) = confirmed and t_deliver(id) < clock then
19:          deliver( receivedMsgSet(id) )
20:       elsif state(id) = unstable and t_confirm(id) < clock then
21:          send (id, abort)
22:          receivedMsgSet := receivedMsgSet – msg(id)
23:       end if
24:    end loop
```

Figure 5.10. *2M* protocol specification: receiver

In the *2M* protocol, a node wanting to send an atomic multicast (Figure 5.9) transmits the data message, followed by a confirmation message, which carries no data. A receiving node before delivering the message, must receive both the message and its confirmation (Figure 5.11).

When a message is received (Figure 5.10), the node marks it as unstable, tagging it with $t_{confirm}$ and $t_{deliver}$ stamps. A node receiving a duplicate message discards it, but updates both $t_{confirm}$ and $t_{deliver}$. As the data message has higher priority than the related confirmation (Table 5.3), then all duplicates will be received before the confirmation. Duplicate confirmation messages will always be sent before any abort (confirmation messages have higher priority than related abort messages), thus they will confirm an already confirmed message.

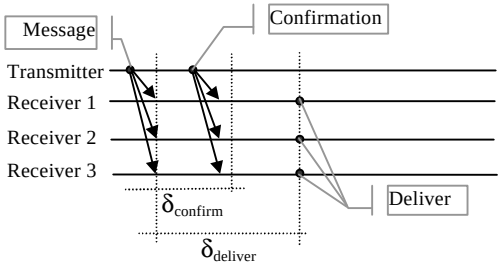

Figure 5.11. *2M* protocol in an error-free situation

If a node does not receive the confirmation before $t_{confirm}$ it multicasts the related abort frame (Figures 5.12 and 5.13 present this situation for the case of inconsistent message and inconsistent confirmation situations, respectively). This implies that several aborts can be simultaneously sent (at most one from each consumer node). A message is only delivered if the node does not receive any related abort frame, until after $t_{deliver}$ (a node receiving the message, but not the confirmation, does not know if the transmitter has failed while sending the message, or while sending the confirmation).

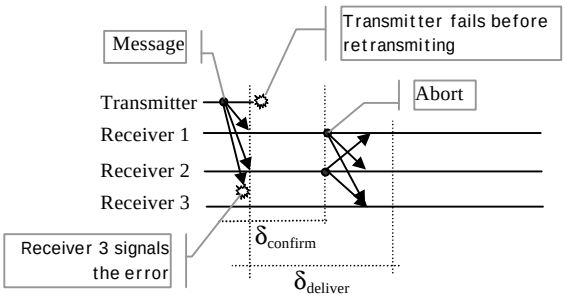

Figure 5.12. Inconsistent message omission while sending the message

The advantage of the *2M* protocol is that in a fault-free execution behaviour there is only one extra frame (without data) per multicast. More protocol related messages in the bus will only be transferred in the case of an inconsistency error (low probability). Note

that the transmission of an abort only occurs in the case of a previous failure of the transmitter. Therefore, from the failure assumptions presented in Section 5.2.1 (there is no second inconsistent message omission in the same period *T*), this abort will be free of inconsistent message omissions.

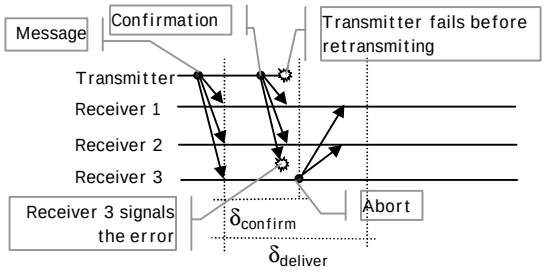

Figure 5.13. Inconsistent message omission while sending the confirmation

The transmitter can automatically confirm the message (Figure 5.9), since if it does not fail, every node will correctly deliver the message and the confirmation. The situation is the same as for the *IMD* protocol, since if the transmitter remains correct and delivers the message, then it will retransmit any failed message.

5.3.3.3. *2M-GD* Protocol

The *2M* protocol can be modified to guarantee the delivery of a transmitted message to all nodes, if it is correctly received by at least one node. In the *2M-GD* protocol (Figures 5.14 and 5.15), nodes receiving the message but not the confirmation retransmit the message (instead of an abort). This protocol is however less efficient than the *2M* protocol (in error situations), since messages are retransmitted with the data field.

```
1:  when atomic_multicast (id, data):
2:     send (id, message, data)
3:     send (id, confirmation)

4:  when sent_confirmed (id, message, data):
5:     receivedMsgSet := receivedMsgSet ∪ msg(id,data)
6:     state(id) := confirmed
7:     t_deliver(id) := clock + δ_deliver(id)

8:  deliver:
9:     for all id in receivedMsgSet loop
10:       if state(id) = confirmed and t_deliver(id) < clock then
11:            deliver( receivedMsgSet(id) )
12:       end if
13:    end loop
```

Figure 5.14. *2M-GD* protocol specification: transmitter

```
1:  when receive (id, type, data):
2:     if type = message then
3:           if id ∉ receivedMsgSet then
4:                receivedMsgSet := receivedMsgSet ∪ msg(id,data)
5:                state(id) := unstable
6:        end if
7:        t_deliver(id) := clock + δ_deliver(id)
8:        t_confirm(id) := clock + δ_confirm(id)
9:     elsif type = confirmation then
10:       state(id) := confirmed
11:    elsif type = retransmission then
12:       if id ∉ receivedMsgSet then
13:            receivedMsgSet := receivedMsgSet ∪ msg(id,data)
14:       end if
15:       state(id) := confirmed
16:       t_deliver(id) := clock + δ_deliver_after_error(id)
17:    end if

18: deliver:
19:    for all id in receivedMsgSet loop
20:       if state(id) = confirmed and t_deliver(id) < clock then
21:            deliver( receivedMsgSet(id) )
22:       elsif state(id) = unstable and t_confirm(id) < clock then
23:            send (id, retransmission, data)
24:       end if
25:    end loop

26: when sent_confirmed (id, retrans, data): -- if retransmitted
27:    state(id) := confirmed
28:    t_deliver(id) := clock + δ_deliver_after_error(id)
```

Figure 5.15. *2M-GD* protocol specification: receiver

To guarantee order of delivery, it is necessary to use a $t_{deliver\_after\_error}$ stamp to solve inconsistent retransmission duplicates. When a protocol retransmission is received (Figures 5.16 and 5.17), the node tags it with $t_{deliver\_after\_error}$ to delay the delivery of the message, until it can guarantee that no duplicates of the retransmission will be received.

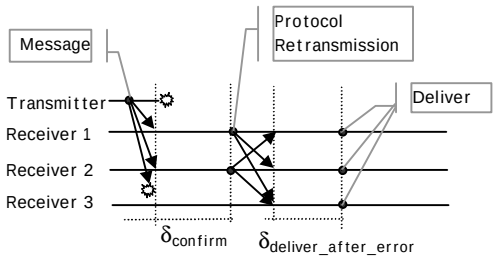

Figure 5.16. Inconsistent message omission while sending the message

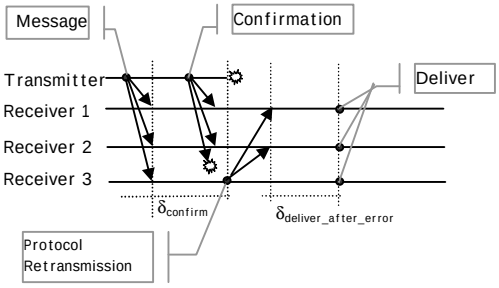

Figure 5.17. Inconsistent message omission while sending the confirmation

5.3.4. Message Fragmentation

The Fragmentation module is necessary to deal with message streams with more than 8 byte messages (CAN's limit). It provides a mechanism by which messages can be fragmented at the transmitter side, and afterwards concatenated at the receiver side.

Taking advantage of the fact that for hard real-time applications there is a fixed set of message streams with fixed data size, the proposed approach allocates a range of CAN identifiers for each message stream requiring fragmentation. For instance, if a specific message stream requires 7 fragments to transfer a data unit, these fragments will have the identifier range from *Base+0* to *Base+6*. The resulting fragments are treated as independent messages by the atomic multicast protocols. In order to ease the task of determining the worst-case response time of fragmented messages, fragments are sent from the lowest to the highest identifier, thus from the highest to the lowest priority.

```
1:  when multicast_delivered (id, data):
2:     if FragSet(id) = ∅ then
3:        t_abort(id) := clock + δ_abort(id)
4:        state(id) := concatenating
5:     end if
6:     FragSet(id) := FragSet(id) ∪ data
7:     count(id) := count(id) + 1
8:     if count(id) = Frags then     -- Frags is the
                                     -- number of fragments
9:        deliver( FragSet(id) )
11:    end if

12: abort:
13:    if state(id) = concatenating and t_abort(id) < clock then
14:       abort(id)
15:    end if
```

Figure 5.18. *Concatenate* protocol specification: receiver

At the receiver side it is necessary to concatenate the fragments to obtain the data unit. The Fragmentation module provides a *Concatenate* protocol (Figure 5.18), built on

top of the atomic multicast protocols, which is responsible for concatenating the received fragments.

Since the identifier range provides fragment information, it is easy for the protocol to simply concatenate the fragments, after being delivered by the atomic multicast layer. The data unit is delivered to the upper layers after receiving all the fragments (Figure 5.19).

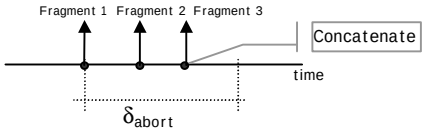

Figure 5.19. *Concatenate* in error free situation

However, if the transmitter crashes in the middle of the transmission, only a subset of the fragments will be received. Therefore, it is necessary to abort the delivery, if all the fragments are not received before a specific time (d_{abort}) (Figure 5.20).

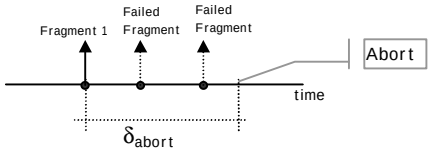

Figure 5.20. *Concatenate* in error situation

Note that, even if the message is to be transmitted using the *2M* or *2M-GD* protocol, it is sufficient to send all but the last fragment using the *IMD* protocol. The reason is that, in the case of an inconsistent message omission no more fragments will be transmitted, and thus the message will be automatically aborted. Therefore, streams that use either the *2M* or *2M-GD* atomic multicast protocols, and that require fragmentation, may have smaller overheads.

5.3.5. Replica Consolidation

The Consolidate module is built on top of the atomic multicast protocols. By using atomic multicasts, it is guaranteed that every replica of a replicated component receives the same set of messages in the same order. The *Consolidate* protocol (Figure 5.21) delays the *decide* phase, until it knows that it has received the full set of messages (Figure 5.22), or until a specific time (d_{decide}) has elapsed (Figure 5.23).

In an error-free situation, the protocol will receive the full set of messages from the replicated components, and thus will perform the consolidation at the instant of arrival of the last message. Note that by using the underlying atomic multicast protocols, each receiver knows that all receivers have correctly and orderly received the set of messages. Thus it can correctly assume that all receivers will take the instant of arrival of the last message as the delivery instant of the consolidated multicast.

```
1:   when multicast_delivered (id, data):
2:      if inputSet(id) = ∅ then
3:         t_decide(id) := clock + δ_decide(id)
4:         state(id) := unstable
5:      end if
6:      inputSet(id) := inputSet(id) ∪ data
7:      count(id) := count(id) + 1
8:      if count(id) = N then       -- N is the number of replicas
9:         decide( inputSet(id) )
10:     end if

11:  decide:
12:     if state(id) = unstable and t_decide(id) < clock then
13:        decide( inputSet(id) )
14:     end if
```

Figure 5.21. *Consolidate* protocol specification: receiver

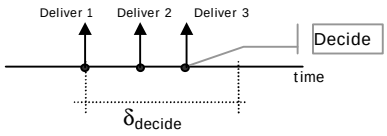

Figure 5.22. *Consolidate* in error free situation

If some of the messages are lost (Figure 5.23), the protocol will wait until t_{decide} to perform the decision phase, since it can assume that all of the receivers have lost the same message (due to the atomic multicast protocols). Note that this decision delay is dependent on the worst-case response time of the sender tasks (see Section 5.4.6).

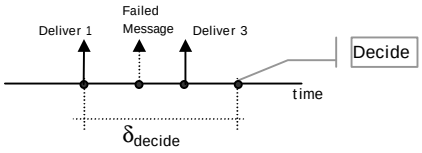

Figure 5.23. *Consolidate* in error situation

Note that this protocol can be used to implement the *many-to-1* and *many-to-many* communication. In the particular case of *many-to-1* communication, there is no need to solve the inconsistent message omission problem, since just one node will deliver the message. However, it is still necessary to address the inconsistent message duplicate problem, as the receiving node may have duplicate messages. Thus, in this case it is sufficient to use the *IMD* protocol as the underlying atomic multicast protocol.

5.3.6. Guaranteeing Communication Properties

Considering that a correct node is a node that does not fail while a multicast is in progress, atomic multicast properties (Section 3.2.3) are guaranteed by the proposed protocols, since:

- Validity: As the CAN built-in mechanisms guarantee that any message will be automatically retransmitted, in the case of either a network or a receiving node error, then the Validity property is guaranteed.
- Agreement: For the *IMD* protocol, if the transmitter does not fail, the CAN built-in mechanisms guarantee that every correct node will receive the message. Thus, they will all deliver it. For the *2M* protocol, a correct node only delivers a data message after receiving the related confirmation message and knowing that it will not receive any abort from other correct nodes. Therefore it knows that all correct nodes will also deliver the message. In the *2M-GD* protocol the behaviour is the same, except in error situations, where if a correct node receives the message it will retransmit it, thus every correct node will receive and deliver it.
- Integrity: As the delivery of the message is delayed, duplicates are discarded and the Integrity property is guaranteed. On the other hand, the CAN built-in mechanisms guarantee that a message is from the actual sender, since a bit error in the identifier field is detected with a sufficiently high probability (ISO, 1993).
- Total Order: The CAN network guarantees that correct messages are received in the same order by all the receiving nodes. However, the existence of duplicates and omissions may preclude messages from being orderly delivered. The use of the $t_{deliver}$ (and $t_{deliver\_after\_error}$) parameter guarantees the total order of message delivery.

The *Consolidate* protocol must guarantee the Agreement property of consolidation. However, this property is only necessary for the case of *many-to-many* communication. In this case, by using an atomic multicast protocol to disseminate the proposed values and by using the t_{decide} parameter, it is guaranteed that all replicated receiving components have the same set of proposed values, and in the same order. Therefore they will all decide on the same value. As for the Validity property, it only depends on the *decide* function used, not on the *Consolidate* protocol itself.

5.4. Response Time Analysis

In order to guarantee the real-time requirements of applications it is necessary to previously analyse the response time of the proposed protocols. Since the presented atomic multicast protocols are based on delaying the delivery and consolidation phases, the response time analysis is constrained by the evaluation of these delays. Moreover, the response time analysis of CAN networks (Sections 3.4.2 and 3.4.3) must be integrated with the temporary periods of network inaccessibility (Section 3.4.4).

Some (or all) of the message streams in the system may involve the exchange of extra messages in the network, either from errors (duplicate messages) or from protocol-related messages (confirmation, abort and retransmission messages), which

must also be integrated in the response time analysis. Extra messages related to a message stream S_m are referred respectively has S_m^{dup}, S_m^{conf}, S_m^{ab} and $S_m^{retrans}$.

This analysis does not consider execution delays caused by protocol execution in each node. However, these can easily be integrated, since they can be bounded through the use of the same response time analysis (Audsley *et al.*, 1993) as for the application software.

5.4.1. Integrating Network Inaccessibility in the Response Time Analysis

In order to integrate the inaccessibility analysis in the response time analysis of CAN networks, the maximum inaccessibility time $Ina(I_m)$ interfering with the transmission of a message of stream S_m must be added to equation (3.21):

$$I_m = B_m + \sum_{\forall j \in hp(m)} \left(\left\lceil \frac{I_m + J_m + t_{bit}}{T_j} \right\rceil \times C_j \right) + Ina(I_m) \tag{5.1}$$

where $Ina(m)$ is a consequence of both bus errors (Ina_{bus}) and transceiver errors (Ina_{transc}):

$$Ina(I_m) = Ina_{bus} + Ina_{transc} \tag{5.2}$$

The maximum number of errors (n_{errors}) that can interfere with the transmission of message m (considering the existence of n errors in a period T) is given by:

$$n_{errors} = n \times \left\lceil \frac{I_m + C_m}{T} \right\rceil \tag{5.3}$$

Hence, according to the considered failure assumptions (a maximum of n_{bus} errors during a time interval T_{bus}), the inaccessibility due to bus errors is (t_{ina} is given by equation (3.24)):

$$Ina_{bus} = n_{bus} \times \left\lceil \frac{I_m + C_m}{T_{bus}} \right\rceil \times t_{ina} \tag{5.4}$$

The maximum inaccessibility due to an erratic transceiver is a consequence of 16 consecutive errors (as a node with an erratic transceiver will go into the Error-Passive state after 16 consecutive errors). Therefore, the maximum inaccessibility due to transceiver errors is:

$$Ina_{transc} = 16 \times t_{ina} \tag{5.5}$$

As a duplicate message is a consequence of a retransmission (due to an inconsistently failed message) the duration of its transmission is also included in the $Ina(I_m)$ term.

The network load considering periods of temporary network inaccessibility is given by:

$$U_{ina} = \frac{n_{bus} \times t_{ina}}{T_{bus}} + \frac{16 \times t_{ina}}{T_{transc}} \tag{5.6}$$

Consequently, embodying equation (5.6) in equation (3.23) the overall network load is:

$$U = \left(\sum_{\forall m} \frac{C_m}{T_m} \right) + U_{ina} \tag{5.7}$$

5.4.2. Response Time Analysis of the *IMD* Protocol

The *IMD* protocol delay ($d_{deliver}$) is used to guarantee that a message is only delivered when it is known that there will be no more duplicates. As the receiving node must evaluate such delay based on local information, it must take the arrival instant as its time reference. It must delay the delivery until the time it takes to completely retransmit a failed message. In the presence of a duplicate message (Figure 5.24), $d_{deliver}$ is reset.

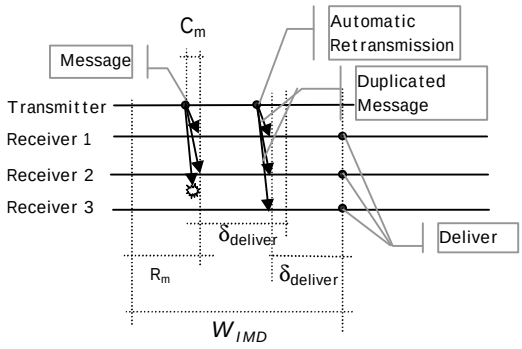

Figure 5.24. *IMD* protocol with one duplicate message

Thus, $d_{deliver}$ must be greater or equal to the worst-case response time of the duplicate message. This response time is equivalent to the worst-case response time of the original message, as it has the same priority. However, as the transmitter immediately tries to retransmit the failed frame, this retransmission will not suffer any blocking:

$$d_{deliver} = R_m^{dup} \quad \wedge \quad B_m^{dup} = 0 \tag{5.8}$$

The worst-case delivery time for message stream S_m must consider the delay introduced by duplicates (considering the possible existence of k_{dup} duplicates):

$$W_m^{IMD} = R_m + (k_{dup} + 1) * d_{deliver} \tag{5.9}$$

The best-case delivery time considers that the message is transmitted with its best-case response time, that is, there is no interference or blocking and no duplicates are transmitted:

$$B_m^{IMD} = C_m + d_{deliver} \tag{5.10}$$

5.4.3. Response Time Analysis of the *2M* Protocol

For the *2M* protocol, it is considered that both the message and the confirmation are put in the transmission queue atomically. By using the CAN identifier field to transfer protocol information (Table 5.3), it is possible to guarantee that there is no order inversion. In the *2M* (and *2M-GD*) protocol assigning to the confirmation message a lower priority than its related data message guarantees that the confirmation will only be transmitted after the data message (the same occurs with the relation between abort/retransmissions and confirmations). Therefore, the request for transmission of both the data and confirmation messages can be atomically performed, reducing the worst-case response time of the related message stream.

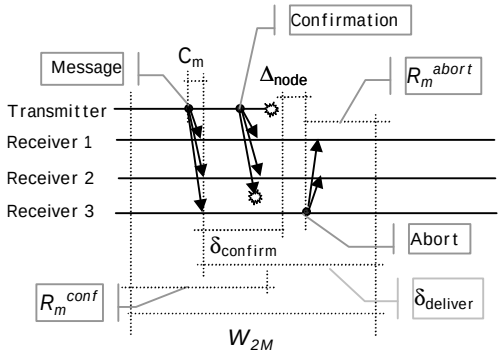

Figure 5.25. *2M* protocol with confirm omission

Therefore, as the message has a higher priority than the confirmation it will be scheduled ahead. Thus, the confirmation will not suffer any blocking from lower-priority messages and, as the arrival instant of the message is taken as the time reference, the confirmation will not suffer interference from the related message (Figure 5.25).

Thus, $d_{confirm}$ must be set to, at least:

$$d_{confirm} = R_m^{conf} - C_m \quad \wedge \quad B_m^{conf} = 0 \tag{5.11}$$

Although disturbances may lead to the duplication of confirmation messages, the $Ina(I_m)$ term of equation (5.1) already integrates these duplicates in the evaluation of the response time of the confirmation message.

The $d_{deliver}$ interval must be determined considering that every receiver must wait until it is known that it will not receive any abort message. These abort messages will be sent by the nodes that do not receive the confirmation message before $d_{confirm}$ (Figure 5.25). This means that the response time of the node itself (D_{node}) to generate an abort message request must also be considered:

$$d_{deliver} = d_{confirm} + \Delta_{node} + R_m^{abort} \tag{5.12}$$

Note that several abort messages may be transmitted in the network, related to the same omission error. However, to determine the $d_{deliver}$ bound it is only necessary to

consider the first one to be transmitted, thus to consider the smaller D_{node} of all receiving nodes. The possible existence of several aborts in the network in case of error must be properly considered for the interference caused in the response-time of lower-priority messages.

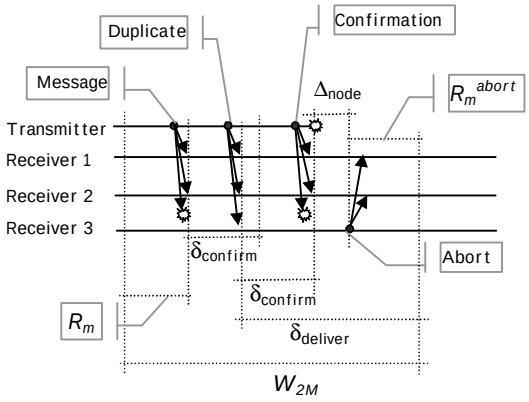

Figure 5.26. *2M* protocol with message duplicate followed by confirm omission

The worst-case delivery time of message stream S_m is when a message is transmitted with its worst-case response time with possible duplicates (Figure 5.26), thus resetting both $d_{confirm}$ and $d_{deliver}$.

Hence, the worst-case delivery time must consider an extra $d_{confirm}$ for each assumed duplicate message:

$$W_m^{2M} = R_m + k_{dup} * d_{confirm} + d_{deliver} \tag{5.13}$$

The best-case delivery time considers the best-case response time of the message and that there are no duplicates or omissions:

$$B_m^{2M} = C_m + d_{deliver} \tag{5.14}$$

5.4.4. Response Time Analysis of the *2M-GD* Protocol

The *2M-GD* protocol has a similar behaviour to the *2M* protocol. For $d_{confirm}$ and $d_{deliver}$ it is only necessary to replace the worst-case response time of the abort message (R_m^{abort}) in equation (5.12) with the worst-case response time of the retransmitted message (which is equal to the worst-case response time of the original message). Multiple retransmissions need also to be considered for the response time evaluation of lower-priority messages.

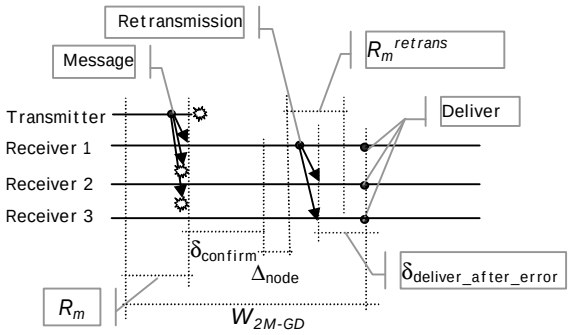

Figure 5.27. *2M-GD* protocol with message omission followed by transmitter failure

However, an extra delay $d_{deliver\_after\_error}$ must also be determined (Figure 5.27), since a receiving node cannot guarantee that other nodes have correctly received the retransmitted message (due to inconsistent retransmission duplicates). Thus, a similar approach to the *IMD* protocol is followed, delaying the delivery until all duplicates are correctly received. Hence, $d_{deliver\_after\_error}$ is equal to the worst-case response time of a duplicated retransmission message:

$$d_{deliver\_after\_error} = R_m^{retrans} \quad \wedge \quad B_m^{retrans} = 0 \tag{5.15}$$

The worst-case delivery time of the *2M-GD* protocol is then evaluated considering that an inconsistent message omission occurs, and retransmissions are needed. Duplicate retransmission messages must be considered, since it must be guaranteed that every node delivers the message at the same time (once again, this response time is determined without blocking, since duplicates are immediately re-scheduled).

However, the existence of multiple retransmissions must also be considered (Figure 5.28). A node receiving a second retransmission will consider it as a duplicate retransmission, and will reset $d_{deliver\_after\_error}$.

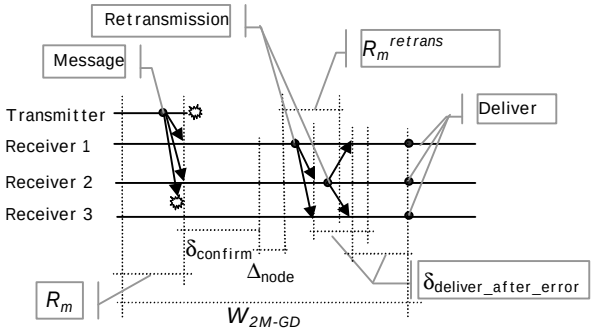

Figure 5.28. *2M-GD* Protocol with retransmissions

Therefore, the worst-case response time must consider the maximum number of retransmissions (n_m^{rec} is the number of receivers of message stream S_m):

$$W_m^{2M-GD} = R_m + k_{dup} * d_{confirm} + d_{deliver} + (n_m^{rec} + k_{dup}) * d_{deliver\_after\_error} \quad (5.16)$$

The best-case delivery time is similar as for the *2M* protocol:

$$B_m^{2M-GD} = C_m + d_{deliver} \quad (5.17)$$

5.4.5. Response time Analysis of the *Concatenate* protocol

The *Concatenate* protocol has to delay the *abort* phase (d_{abort}), until it knows that it will not receive any further fragments. This delay is dependent on the worst-case delivery time of the fragments (considering the delivery by the atomic multicast layer). Therefore, d_{abort} may be determined assuming the best-case delivery time for the highest-priority fragment, and the worst-case delivery time for the lowest-priority fragment (Figure 5.29):

$$d_{abort} = W_{lpf(m)} - B_{hpf(m)} \quad (5.18)$$

where *lpf(m)* and *hpf(m)* are, respectively, the lowest-priority and the highest-priority fragment of a message stream S_m.

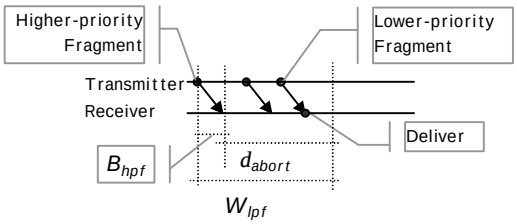

Figure 5.29. *Concatenate* in error free situation.

The worst-case delivery time of the *Concatenate* protocol is then evaluated by considering that the worst-case response time of the lowest-priority fragment already considers the interference of the higher-priority fragments. Note that, a lost fragment implies that the sending node has crashed (if not, it would have retransmitted the fragment). Thus, in this case, the message will be aborted (Figure 5.30).

Therefore, the worst-case delivery time of the *Concatenate* protocol is equal to the worst-case delivery time of the lowest-priority fragment:

$$W_m^{concat} = W_{lpf(m)} \quad (5.19)$$

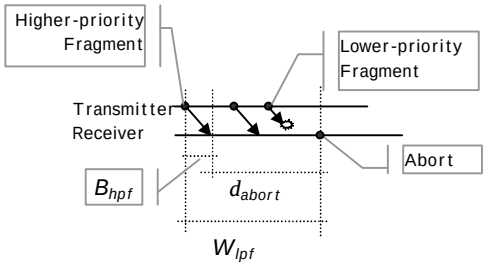

Figure 5.30. *Concatenate* in error situation

For the best-case delivery time of the *Concatenate* protocol, it is necessary to consider the best-case delivery time of the lowest-priority fragment (last fragment to be transmitted, thus also the last to be delivered), related to the beginning of the transmission of the set of fragments (thus considering the actual transmission time (C_m) of the other fragments):

$$B_m^{concat} = B_{lpf(m)} + \sum_{\forall i \in frag'(m)} \{C_i\} \tag{5.20}$$

where *frag'(m)* is the set of fragments of a message belonging to a message stream S_m, excluding the fragment with the lowest-priority.

5.4.6. Response Time Analysis of the *Consolidate* Protocol

The *Consolidate* protocol follows an approach similar to the previous *Concatenate* protocol, delaying the *decide* phase (d_{decide}) until it knows that it will not receive any further messages. This delay is dependent on the worst-case delivery time of the replicated messages (considering the delivery by the atomic multicast protocols), referred to an initial time reference common to all sending nodes. This common time reference must be the release time of the sending tasks. Therefore, the worst-case delivery time of the messages must also consider the worst-case response time of the related sending tasks.

If the replicated tasks are periodic, then their release time is common in all the nodes (with a small jitter, as clocks are just approximately synchronised). If these tasks are sporadically released by external events, then their release time must be also agreed between the different replicas (to guarantee deterministic execution). Thus, this agreed time instant can be considered as the reference for the worst-case response time of the replicated messages. If replicated tasks are sporadically released by other tasks, then their release time may vary between replicated components. In this case, it is necessary to consider the release time of the initial task with a common release time, and determine the worst-case response time of the sending task related to that initial time reference.

Knowing the worst-case delivery time for each replicated message, an upper bound for d_{decide} may be determined assuming the best-case delivery time for the first message to arrive, and the worst-case delivery time for the last message to arrive (Figure 5.31):

$$d_{decide} = \max_{\forall i \in rep(m)} \left\{ W_i^{com} \right\} - \min_{\forall i \in rep(m)} \left\{ B_i^{com} \right\} + e \tag{5.21}$$

where W_i^{com} is the worst-case delivery time of message *i* and B_i^{com} is the best-case delivery time of message *i*, both considering a common time reference, and *rep(m)* is the set of replicated messages of a particular message stream S_m. *e* is the maximum clock deviation.

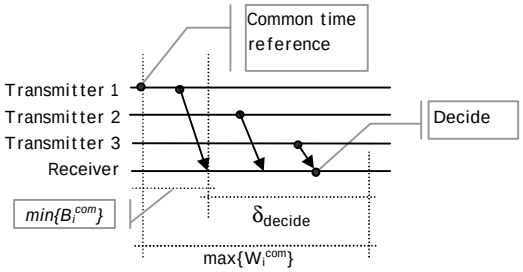

Figure 5.31. *Consolidate* in error free situation.

The best-case delivery time of the consolidated message can be determined, considering that all messages are received with their best-case delivery time. Therefore, the best-case delivery time of the consolidated message is:

$$B_m^{decide} = \max_{\forall i \in rep(m)} \left\{ B_i^{com} \right\} + e \tag{5.22}$$

The worst-case delivery time depends on the assumed failure assumptions. In an error free situation (Figure 5.31) the worst-case delivery time would be:

$$W_m^{decide} = \max_{\forall i \in rep(m)} \left\{ W_i^{com} \right\} + e \tag{5.23}$$

However, if some of the replicated messages are not delivered (due to inconsistent message omissions or failed sending tasks), the protocol will wait d_{decide} from the arrival of the first message. Figure 5.32 presents such situation, where the message from Transmitter *1* is lost. Since the receiver uses the arrival instant of the first message as the time reference, d_{decide} will be added to the arrival instant of the second message delivered by the atomic multicast protocol.

Therefore, considering that *f* replicated messages are not delivered, the worst-case delivery time of the consolidated message is:

$$W_m^{decide} = \min_{\forall i \in rep'(m)} \left\{ W_i^{com} \right\} + d_{decide} \tag{5.24}$$

where *rep'(m)* is the set of replicated messages, excluding the *f* messages with the smaller worst-case delivery time.

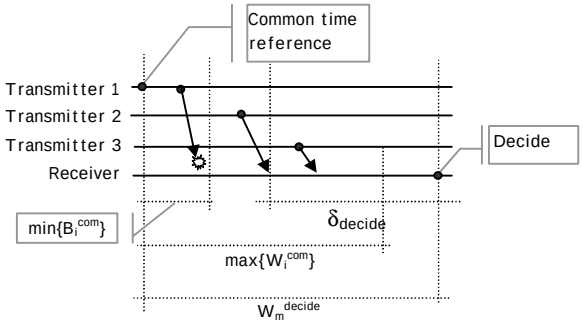

Figure 5.32. *Consolidate* with one lost message

5.4.7. Integrating Communication Overheads in the Response Time Analysis

The response time analysis of CAN networks must be updated to integrate the overheads concerning confirmation messages and possible aborts or retransmissions of message streams that use the *2M* or *2M-GD* protocols.

Therefore, the worst-case queuing delay (equation (5.1)) must be updated to consider periods of interference from higher-priority message streams using atomic multicast protocols:

$$I_m = B_m + \sum_{\forall j \in hp(m)} \left(\left\lceil \frac{I_m + t_{bit}}{T_j} \right\rceil \times (C_j + C_j^{extra}) \right) + Ina(I_m) + \max_{\forall j \in hp(m)} \left\{ extra\_msg_j \right\} \qquad (5.25)$$

C_j^{extra} is the interference caused by the confirmation message, which is:

$$C_j^{extra} = \begin{cases} C_j^{conf} & 2M \text{ or } 2M-GD \text{ protocol} \\ 0 & otherwise \end{cases} \qquad (5.26)$$

Additionally, *max{extra_msg$_j$}* accounts for the aborts or retransmissions in the network, due to inconsistent message omissions. As it is assumed the existence of a single inconsistent message omission during a period T (greater than the largest worst-case delivery time), each receiver of message stream S_j will transmit, at most, one abort/retransmission due to inconsistent message omissions, that is:

$$extra\_msg_j = \begin{cases} n_j^{rec} * C_j^{abort} & 2M \text{ protocol} \\ n_j^{rec} * C_j^{retrans} & 2M-GD \text{ protocol} \\ 0 & otherwise \end{cases} \qquad (5.27)$$

where n_j^{rec} is the number of receivers of message stream S_j.

The $Ina(I_m)$ term (equation (5.25)) integrates the periods of network inaccessibility caused by errors in frame transmission, therefore it already includes the retransmissions of inconsistently failed messages (that is, duplicates).

Considering the network utilisation, equation (5.7) must be also updated. For each message stream transmitted with the *2M* or *2M-GD* protocol, an extra confirmation message must be considered (C_m^{extra}, equation (5.26)). Also, it must be considered the maximum number of extra messages related to inconsistent message omissions, per period of analysis T:

$$U = \left(\sum_{\forall m} \frac{C_m + C_m^{extra}}{T_m} \right) + U_{ina} + \frac{\max\{extra\_msg_m\}}{T_m} \tag{5.28}$$

Note that these equations maintain the properties of CAN schedulability analysis. Therefore, they can be used in the holistic approach (Tindell and Clark, 1994) presented in Chapter 3, allowing determining the overall response times of the system.

5.5. Numerical Example

In order to clarify the use of the presented model, a simple example is used. In this example (Figure 5.33) a system where a distributed hard real-time application executes is considered. The system is constituted by four nodes, connected by a CAN network at a rate of 1 Mbit/sec. The application is constituted by four tasks ($t_1..t_4$), which are spread over the nodes. As component replication is also used, then some of these tasks are also replicated. In this simple application, each task outputs its results to the following task.

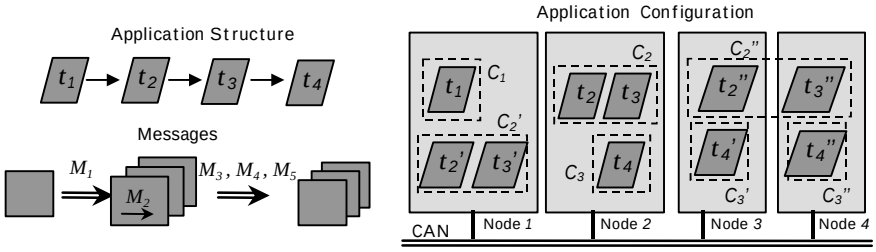

Figure 5.33. Application example

The application is divided in three components: component C_1 encompasses task t_1, component C_2 encompasses t_2 and t_3, and finally component C_3 is just t_4. Components C_2 and C_3 are replicated in three replicas, while component C_1 is not replicated.

Table 5.4 presents each task's characteristics, while Table 5.5 presents the characteristics of the necessary message streams (all values are in milliseconds). Note that messages from t_2 to t_3 and t_2' to t_3' are internal to the node, since they are intra-component, and both tasks are in the same node. Since message stream S_1 is a *1-to-many* communication, the *2M-GD* protocol is used in order to guarantee that every replica of task τ_2 delivers the message. Therefore, there will be an extra confirmation

message with the same period of message stream S_1, but without data. Since it is considered that an inconsistent message omission may occur, then it is also necessary to account for 3 possible retransmissions (one from each receiving node).

Table 5.4. Tasks' characteristics

Task	Type	WCET	Period	Comp.	Nodes
t_1	Periodic	2	5	*C1*	*1*
t_2	Periodic	2	10	*C2*	*1,2,3*
t_3	Sporadic	3	10	*C2*	*1,2,4*
t_4	Periodic	4	15	*C3*	*2,3,4*

Table 5.5. Messages streams' characteristics

Stream	Bytes	Period	From	To	Prot.
S_1	4	5	t_1	t_2,t_2',t_2''	*2M-GD*
S_2	8	10	t_2''	t_3''	*IMD*
S_3	6	10	t_3	t_4,t_4',t_4''	*2M*
S_4	6	10	t_3'	t_4,t_4',t_4''	*2M*
S_5	6	10	t_3''	t_4,t_4',t_4''	*2M*

Message stream S_2 is internal to a component (although the component is spread between nodes *3* and *4*), and it is a *1-to-1* communication. In this case, it is sufficient to use the *IMD* protocol, since only duplicates are to concern. Message streams S_3 to S_5 are messages from replicated τ_3 to replicated τ_4, needing consolidation in every replica of τ_4. As this consolidation will mask node failures of the senders, then it is sufficient to use to *2M* protocol for the transmission of messages. Therefore there will be an extra confirmation message for each message sent (and possible abort messages).

The following assumptions are considered:

- a maximum of 2 message faults in each 10 ms time interval, resulting from a bit error rate of approximately 10^{-4}, which is an expectable range for bit error rates in aggressive environments;
- one inconsistent message omission during the period of analysis;
- one duplicate in the transmission of a message ($k_{dup} = 1$);
- a Δ_{node} equal to 100 μS and a maximum deviation between clocks (e) of 100 μS.

The goal of this example is to analyse the responsiveness of the proposed protocols, for both the response time and the delivery time of messages. *Response time* is considered as the time interval between requesting a message transfer until the message is fully received at the receiver side. *Delivery time* is considered as the time interval between requesting a message transfer until the message is delivered to the upper layers of the receiver. If multicast protocols are not used, these times are equivalent, as it can be assumed that messages are delivered when they are correctly received.

Table 5.6 presents the response time for each message stream and the network load when multicast protocols are not used (the *Unreliable* protocol is used instead of *IMD/2M/2M-GD* protocols). R_m^{NP} represents the worst-case response time (NP: no protocols), P is the periodicity and C_m is the actual time taken to transmit a message. U is the network utilisation.

Table 5.6. Message streams' response time without protocols

Stream	P	C_m	R_m^{NP}
S_1	5	0.089	0.519
S_2	10	0.127	0.630
S_3	10	0.108	0.741
S_4	10	0.108	0.852
S_5	10	0.108	0.852
U		6.590 %	

As it can be seen, the worst-case response time of messages is considerably greater than their actual transmission time. Although interference from higher-priority messages is one of the factors leading to such difference, the main factor is the network bit error rate. For instance, a message of stream S_1 in an error-free environment would have a worst-case response time of 0.219 ms. The possible existence of errors in the network more than duplicates its worst-case response time, even when multicast protocols are not used.

Table 5.7. Protocol-related delays

Stream	Prot.	$d_{confirm}$	$d_{deliver}$	$d_{del\_aft\_er}$
S_1	*2M-GD*	0.350	0.969	0.389
S_2	*IMD*	-	0.848	-
S_3	*2M*	0.901	2.013	-
S_4	*2M*	1.065	2.341	-
S_5	*2M*	1.229	2.558	-

Table 5.8. Message streams' delivery time considering protocols

Stream	Prot.	R_m^{MP}	W_m	B_m	W_m/R_m^{MP}
S_1	*2M-GD*	0.519	3.394	1.058	6.54
S_2	*IMD*	0.959	2.655	0.975	2.77
S_3	*2M*	1.070	3.984	2.121	3.72
S_4	*2M*	1.234	4.640	2.449	3.76
S_5	*2M*	1.287	5.074	2.666	3.94
U			9.09 %		

Tables 5.7 and 5.8 present the messages' delays and delivery times considering the use of the proposed multicast protocols. R_m^{MP} represents the worst-case response time of a message stream when multicast protocols (MP) are considered. W_m and B_m are, respectively, the worst- and best-case delivery time for message stream S_m.

As it can be seen in Table 5.8, the worst-case delivery time is greater than the related worst-case response time, because apart from the multicast-related introduced delays, it is assumed that each message may be disturbed by one duplicate. For instance, the worst-case delivery time for message stream S_5 is not only given by the message stream response time plus its $d_{deliver}$, but also by summing an extra $d_{confirm}$ due to a message duplicate.

The last column of Table 5.8, presents the ratio worst-case delivery time/worst-case response time, when considering the use of multicast protocols. It is obvious that the *IMD* protocol is the one that introduces smaller delays (message stream S_2), while the *2M-GD* protocol is the one with the higher delays (message stream S_1). Therefore, the system developer can use this reasoning to balance reliability vs. efficiency in the system. Moreover, the multicast protocols increase network utilisation less than 50%, since multicast-related retransmissions only occur in inconsistent message omission situations. Although this network load increase is still large, it is much smaller than in other approaches, and it is the strictly necessary to cope with inconsistent message omissions using a software-based approach.

Since messages from replicated tasks t_3 to replicated tasks t_4 need to be consolidated, it is necessary to determine the d_{decide} parameter of the *Consolidate* protocol. As stated, it is necessary to find the worst-case and best-case delivery time for each one of the message streams (S_3 to S_5). However, these delivery times must refer to a common time base. Thus, it is necessary to determine the best-case and worst-case response time of replicated tasks t_3. This task is a sporadic task released by t_2. Hence, its response time is dependent of the response time of t_2.

These response times can be easily determined using the analysis presented in (Audsley *et al.*, 1993). However, replicated component C_2'' is spread over nodes *3* and *4*. Thus, in order to determine the worst-case and best-case response times of t_3'' it is necessary to consider the delivery time of message stream S_2.

Table 5.9. Consolidation

Task	*WCRT*	*BCRT*	Stream	W^{com}	B^{com}
τ_3	5	5	S_3	8.984	7.121
τ_3'	9	7	S_4	13.64	9.449
τ_3''	7.655	5.975	S_5	12.729	8.641

Table 5.9 presents the best-case and worst-case response time of replicated tasks t_3, and the associated worst-case and best-case delivery time of messages streams S_3 to S_5 (all referring to the common release time of task t_2). Therefore (from equation (5.21)):

$$d_{decide} = 13.64 - 7.121 + 0.1 = 6.619 \text{ ms} \tag{5.29}$$

The worst-case response time of the consolidation (from the time that the first message is scheduled for transmission) is determined assuming that all messages but one are delivered at their worst-case delivery time, and the other is not delivered. In this case (assuming that the message not delivered is the one with the lower worst-case response time (S_3)), the message from stream S_4 will arrive with its worst-case response time of 4.640, which summed to the d_{decide} gives a worst-case consolidation time of:

$$W^{decide} = R_{M4} + d_{decide} = 11.259 \text{ ms} \tag{5.30}$$

Although the delay introduced in the system by this type of consolidation, the advantage is that no extra overhead is introduced in the network, preserving at the same time the predictability of the system. As one of the main goals of the proposed multicast

protocols is to provide fault tolerant CAN communication, whilst preserving CAN's real-time characteristics (thus allowing the off-line analysis of messages' response times), such goal is achieved as the predictability of message transfers is guaranteed.

5.6. Comparison with Similar Approaches

In (Rufino *et al.*, 1998), a similar approach for providing fault-tolerant broadcast protocols is proposed, solving the message omission and duplicate problems. The RELCAN protocol is similar to the *2M-GD* protocol, being based on the transmission of a second data-free message (CONFIRM message), to signal that the sender is still correct. If this confirm message does not arrive before a specific timeout, the message is retransmitted. However, this retransmission is performed using a lower layer protocol (EDCAN), which is based on the retransmission of messages by every node in the system (that has correctly received the message). When a node receives a retransmission, it will retransmit it again (even if it already has retransmitted the original message). This behaviour leads to a huge number of messages in the network. Although the authors refer the possibility of several identical messages being clustered in the bus (all transmitted at the same time), this situation can not always be assumed. It is possible that some of these messages are not simultaneously transmitted, since sender nodes have distinct processing delays. Therefore, the worst-case response time grows exponentially with the number of nodes in the network, which is not the case for the *2M-GD* protocol.

In the RELCAN protocol, the transmission request of the CONFIRM message is only made after receiving information from the CAN controller that the data message has already been sent. This two-phase approach is necessary to guarantee that there is no order inversion, that is, the CONFIRM message is only sent after the related data message. In the *2M* (and *2M-GD*) protocol this non-inversion guarantee is provided by giving to the confirmation message a lower priority than its related data message. Therefore, the request for transmission of both the data and confirmation messages can be atomically performed, reducing the worst-case response time of the related message stream.

One of the disadvantages of the RELCAN protocol is that it does not provide total order (thus it cannot be used to achieve atomic multicasts). When a data message is received, it is immediately delivered, meaning that in the presence of inconsistent message errors the order is not preserved. In (Rufino *et al.*, 1998), total order is addressed by the TOTCAN protocol. This protocol is also based on a two-phase approach, but the transmission of an ACCEPT message (similar to the CONFIRM message) is performed using the EDCAN protocol. Since multiple retransmissions will occur in normal operation, even if no error occurs, the TOTCAN protocol incurs a higher overhead, increasing significantly the network utilisation. For instance, when transmitting a fault-free message in a network with four nodes, in addition to the message there will be the ACCEPT message plus three retransmissions. Therefore, in the best-case (data message with 8 bytes), the overhead is approximately 150%, compared with the 40% of the *2M* protocol. In case of sender failure it does not deliver the message (it guarantees that the message is delivered by all or none of the recipients as the *2M* protocol).

5.7. Summary

This chapter presented the proposed set of atomic multicast and consolidation protocols, upon which the CAN fault-tolerant real-time communication is guaranteed. In the proposed approach, atomic multicasts are guaranteed through the transmission of just an extra message (without data) for each message that must tolerate inconsistent message omissions. Only in case of an inconsistent message omission (low probability) there will be more protocol-related retransmissions. Inconsistent message duplicates are solved with a protocol that does not require extra transmissions, guaranteeing total order. Moreover, atomic multicast properties are achieved without more overheads than the strictly needed for a reliable multicast. Consolidation of replicated inputs is also provided through the use of a consolidate protocol, built on top of the multicast protocols.

These protocols explore the CAN synchronous properties to minimise their run-time overhead, and thus to provide a consistent and timely service to the supported applications. The model and assumptions of these protocols for the evaluation of the message streams' response time are also presented, demonstrating that the real-time capabilities of CAN are preserved, since predictability of message transfers is guaranteed.

Chapter 6

Lessons Learnt from the Framework Implementation

6.1. Introduction

This thesis provides a generic and transparent framework for the development of fault-tolerant real-time applications conforming to the Ravenscar profile. However, it is also necessary to assess if the Ravenscar profile is expressive enough for the implementation of the complex middleware intended for the support to replicated/distributed applications. Hence, this chapter presents the main guidelines used for the implementation of a prototype of the framework, describing the framework structure and its main mechanisms. Also, and more important, this chapter discusses some of the most relevant lessons learnt from the framework implementation.

As the goal of implementing the prototype was to assess the difficulties presented by the Ravenscar profile to implement replication and distribution, the full specification of the prototype is not presented. Instead, the focus is given to the prototype structure and to the most relevant mechanisms used in its implementation. Since the implementation of the communication and replication mechanisms follows the protocols presented in Chapters 4 and 5, their implementation will not be presented.

This chapter is structured as follows. Section 6.2 describes the structure of the prototype, focusing on the implementation of the framework main modules (*Object Repository*, *Replica Manager* and *Communication Manager*), also presenting the prototype's main limitations. Afterwards, Section 6.3 presents the support provided to the final configuration phase of the application. Finally, Section 6.4 discusses the problems related to the Ravenscar restrictions, and draws some conclusions on the expressiveness of the profile.

6.2. Prototype Implementation

The specification of the prototype will be presented using a decomposition similar to the HRT-HOOD design methodology (Burns and Wellings, 1995a). Packages will be represented by round boxes (hierarchies are presented by placing the child package box inside the parent's box), an arrow represents a *use* relationship between packages, and a rectangular box represents used packages outside of the current hierarchy. In order to be

able to represent the internal package resources (subprograms, tasks, and protected objects), when necessary a rectangular box will be used (Figure 6.1).

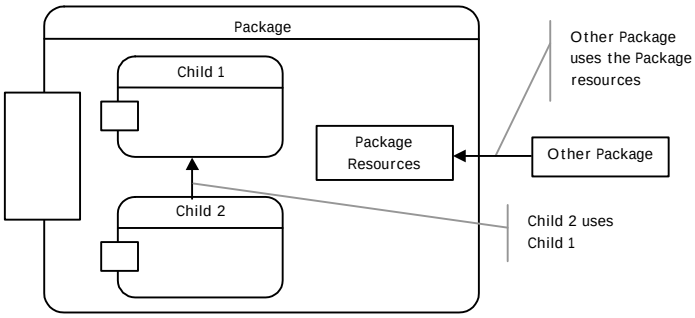

Figure 6.1. Model example

The structure of the prototype implementation is presented in Figure 6.2. The *Object_Repository*, *Replica_Manager* and *Communication_Manager* hierarchies map the framework structure presented in Chapter 4. The *Framework_Types* package hierarchy contains the specification of all types that are global to the prototype, and although not shown is implicitly used in the remaining figures of this chapter. The *Application* hierarchy contains the constants and data structures needed to configure the framework for application-specific information. This hierarchy is used by *Framework_Types* (for type declaration) and by the *Property_Recorder* and *Configuration* modules of the *Replica_Manager* and *Communication_Manager*, respectively. Finally, the *Can_Board_Driver* package is the interface to the specific CAN board used in the prototype implementation. This package is only used by the *Communication_Manager*.

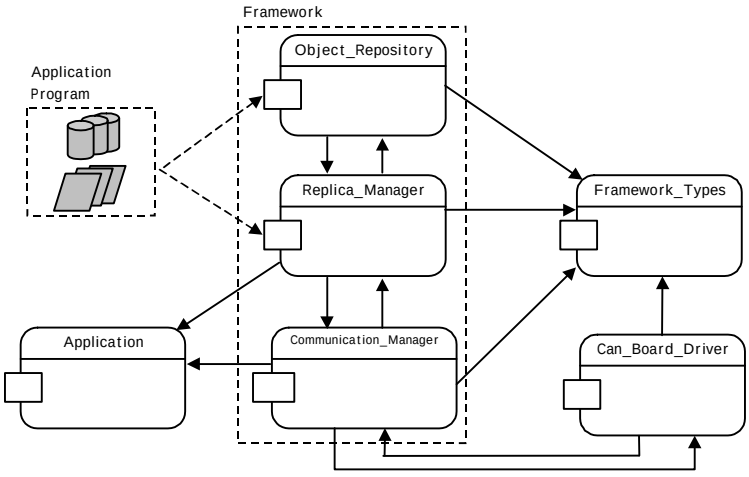

Figure 6.2. Prototype implementation structure

As presented in the prototype implementation structure, there is no direct interaction between the *Object_Repository* and the *Communication_Manager* hierarchies. The *Replica_Manager* performs this interface, in order to shield the *Repository* from changes in the communication infrastructure.

The control flow within the framework maps the downstream and upstream flow of communication streams.

6.2.1. Object Repository

The structure of the *Object_Repository* hierarchy is presented in Figure 6.3. The child packages *Shared_Data*, *Release_Event* and *Release_Event_With_Data* provide the simple objects intended to be used when no replication and distribution is considered, without any replication/distribution management mechanisms.

The *Inter_Group* sub-hierarchy (Figure 6.4) provides child packages for the interaction objects between groups of replicated components. The *Intra_Component* sub-hierarchy (Figure 6.5) provides the interaction objects related to interactions inside a component. Packages *Inter_Group* and *Intra_Component* are empty packages, and are only used for hierarchy reasons.

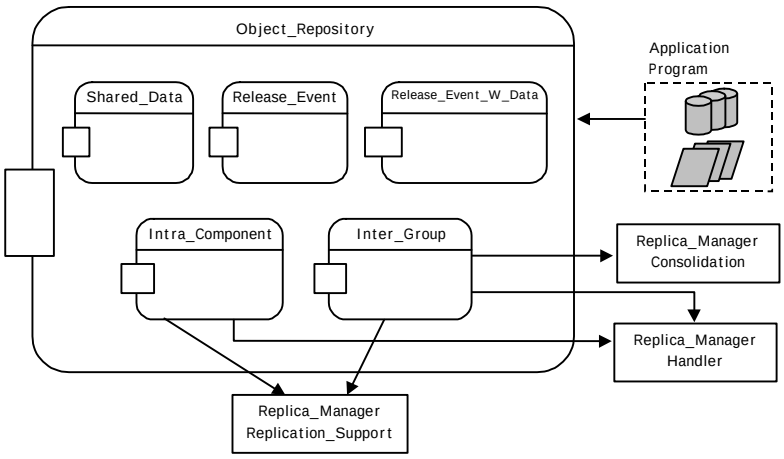

Figure 6.3. *Object_Repository* structure

The *Replication_Support* package is used for the support of distribution and replication. The *Consolidation* hierarchy of the *Replica_Manager* provides some of the required mechanisms for replica consolidation. All objects that act as receivers of communication streams require the *Handler* package, since the absence of dynamic priorities in the Ravenscar profile forces objects to create their own reception handler.

The *Intra_Component* hierarchy (Figure 6.5) does not require the *Consolidation* mechanism. Nevertheless, all its objects require the *Replication_Support* package, and the objects with distribution capabilities must also use the *Handler* package.

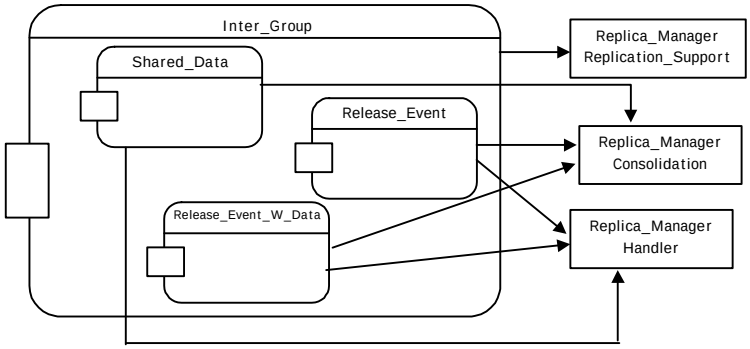

Figure 6.4. *Object_Repository Inter_Group* structure

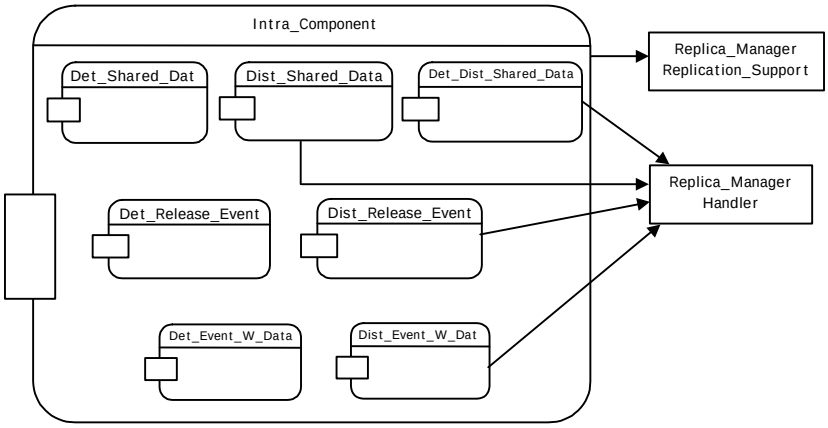

Figure 6.5. *Object_Repository Intra_Component* structure

In the *Object_Repository*, all terminal objects are implemented as generic packages. The use of generic packages allows reusing the same implementation mechanisms for objects with different data types. Even objects that do not require data (as the simple *Release_Event*) are implemented as generics, since they require objects to be instantiated with extra information (for instance object identifier and ceiling priority). The use of generic packages allows parameterisation at compile time, providing significant reuse capabilities.

Since the goal of transparency is to allow the simple objects to be replaced by objects with distribution and replication capabilities, the public interface for similar interaction objects is the same. For instance, Figures 6.6 and 6.7 present the public interface for the *Release_Event* and *Inter_Group Release_Event* objects, respectively.

Although the differences between the generic parameters (the *Inter_Group* object requires extra parameters: the number of proposing replicas and the procedure to be used for the decision on the release instant), the interface to the application code is the same

(lines 5 to 8 of Figure 6.6 and lines 7 to 10 of Figure 6.7). This approach is used in all similar objects, allowing the configuration phase to modify just the objects declaration, not changing the application tasks' code (application configuration issues will be further detailed in Section 6.3).

```
1:  generic
2:     Id    : Framework_Types.Obj_Id_Type;
3:     Prio  : System.Priority;

4:  package Object_Repository.Release_Event is

5:     type Release_Obj is private;

6:     function Request_Release_Obj return Release_Obj;

7:     procedure Wait    (Obj: Release_Obj);  -- potentially
                                              -- blocking
8:     procedure Release (Obj: Release_Obj);

9: private
       -- private interface
10: end Object_Repository.Release_Event;
```

Figure 6.6. *Release_Event* object public interface

```
1:  generic
2:     Id             : Framework_Types.Obj_Id_Type;
3:     Prio           : System.Priority;
4:     N_Replicas     : Framework_Types.Rep_Id_Type;

5:     with procedure Decide (
          Instant_Values  : Framework_Types.Instant_Array_Type;
          Valid_Instants  : Framework_Types.Boolean_Array_Type;
          Rejected_Inst   : out Framework_Types.Boolean_Array_Type;
          Release_Instant : out Ada.Real_Time.Time;
          Release_Ok      : out Boolean);

6:  package Object_Repository.Inter_Group.Release_Event is

7:     type Release_Obj is private;

8:     function Request_Release_Obj return Release_Obj;

9:     procedure Wait    (Obj: Release_Obj);  -- potentially
                                              -- blocking
10:    procedure Release (Obj: Release_Obj);

11: private
       -- private interface
12: end Object_Repository.Inter_Group.Release_Event;
```

Figure 6.7. *Inter_Group Release_Event* object public interface

Internally, the *Repository* objects are implemented as Ada protected objects, in order to provide mutual exclusion for the access to the object state and, for the case of *Release* objects, to allow application tasks to be blocked by a protected entry. As an example, Figure 6.8 presents the specification of the protected type used for the *Inter_Group Release_Event* object.

```
1:  generic
       -- ...
2:  package Object_Repository.Inter_Group.Release_Event is

3:     type Release_Obj is private;
       -- ...
4:  private
5:     protected type Release_Receive_Type (
                         Prio: System.Priority;
                         Id: FT.Obj_Id_Type) is
6:        pragma Priority(Prio);
7:        entry Wait;
8:        procedure Release;
9:        function Get_Id return FT.Obj_Id_Type;
10:    private
11:       Obj_Id: FT.Obj_Id_Type := Id;
12:       Released: Boolean := False;
13:    end Release_Receive_Type;

14:    type Release_Obj is access all Release_Receive_Type;

15: end Object_Repository.Inter_Group.Release_Event;
```

Figure 6.8. Example of object private implementation

In order to support the consolidation mechanism, the *Inter_Group* objects are required to instantiate a new consolidation object of the *Replica_Manager* (Figure 6.9, line 3). This object will be made available to the *Communication_Manager* for the decision phase, since this phase requires a data-specific procedure (detailed in Section 6.3).

```
1:  package body Object_Repository.Inter_Group.Release_Event is
       -- ...
2:     package RM_RS renames Replica_Manager.Replication_Support;
3:     package Consolidate is new
            RM_RS.Consolidation.Release_Event ( Id, N_Replicas,
                                                Decide );
4:     function Request_Release_Obj return Release_Obj is
5:        Obj : Release_Obj;
6:     begin
          -- ...
7:        Consolidate.Register_Object (Id);
8:        return Obj;
9:     end Request_Release_Obj;
10: end Object_Repository.Inter_Group.Release_Event;
```

Figure 6.9. *Inter_Group Consolidate* object creation example

In a similar approach, objects with distribution requirements are required to instantiate a new *Handler* package of the *Replica_Manager* (Figure 6.10 presents the example for the *Intra_Component Distributed_Release_Event* object). This *Handler* package (Figure 6.11) provides the task required by the *Replica_Manager* to interact with the object (thus it is created with the same priority as the ceiling priority of the object).

```
1:  package body Object_Repository.Intra_Comp.Dist_Release_Event is

2:     package RM_RS renames Replica_Manager.Replication_Support;

3:     package Handler is new RM_RS.Receive.Handler (Id,
                          Ada.Real_Time.Time'Size, Prio );

       -- ...
4:     function Request_Release_Receive_Obj
            return Release_Receive_Obj is
5:        Obj: Release_Receive_Obj;
6:     begin
          -- ...
7:        Handler.Init;
8:        return Obj;
9:     end Request_Release_Receive_Obj;

10: end Object_Repository.Intra_Comp.Dist_Release_Event;
```

Figure 6.10. *Handler* task creation example

```
1:  generic
2:     Id: FT.Obj_Id_Type;
3:     Size: Integer;
4:     Prio: System.Any_Priority;
5:  package Replica_Manager.Replication_Support.Receive.Handler is

6:     package FT renames Framework_Types;
7:     package A_RT renames Ada.Real_Time;

8:     task type Handler ( Obj: FT.Obj_Id_Type;
                           Size: Integer;
                           Prio: System.Any_Priority) is

9:        pragma Priority (Prio);
10:    end Handler;

11:    procedure Init;

12:    procedure Release ( Msg_Type: FT.Message_Type;
                           Str: FT.Stream;
                           Tdeliver: A_RT.Time);

13: end Replica_Manager.Replication_Support.Receive.Handler;
```

Figure 6.11. *Handler* task specification

6.2.2. Replica Manager

Figure 6.12 presents the structure of the *Replica_Manager* hierarchy. This structure maps the actual structure of the Replica Manager presented in Chapter 4. The *Error_Manager* package, in addition from being used by the framework, may also be used to notify any error occurrence to the application. The *Application_Support* package provides application tasks with the support specified in Chapter 4 (Section 4.5.3), and also with the necessary interface to initialise the framework and to wait for the initialisation to complete. The application can use this interface to only start tasks (by starting the application components) after the complete initialisation of the system.

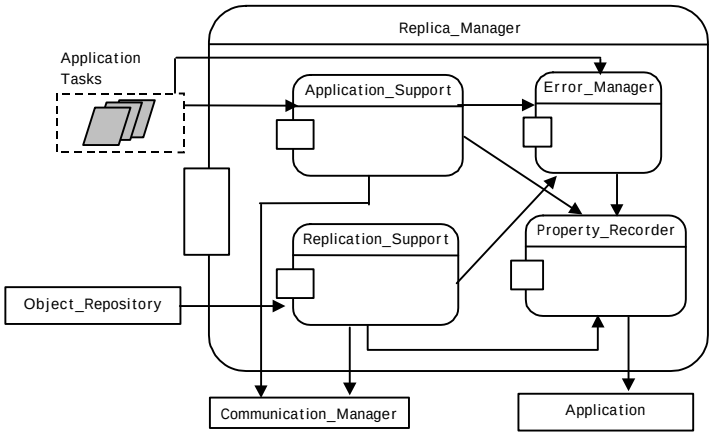

Figure 6.12. *Replica_Manager* structure

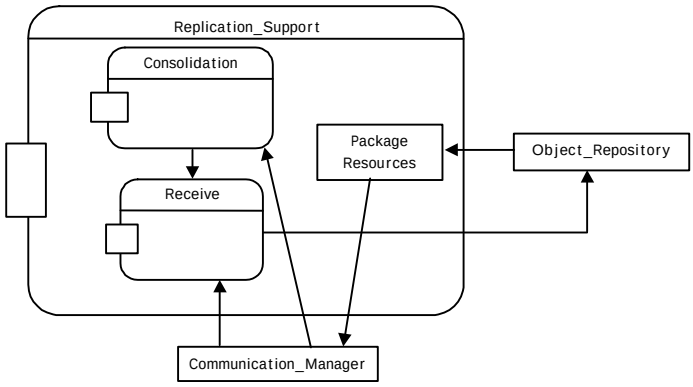

Figure 6.13. *Replication_Support* structure

The *Replication_Support* package (Figure 6.13) provides the support to the instantiated objects in the application. In order to avoid elaboration problems, a child

package (*Receive*) is provided for the reception of messages from the *Communication_Manager*, or from the *Consolidation* support of the *Replica_Manager*.

The *Consolidation* support of the *Replica_Manager* (Figure 6.14) is used to achieve the needed shielding between the *Object_Repository* and the *Communication_Manager* (as described in the previous subsection). In order to use this support, objects requiring consolidation (see Figure 6.9) instantiate a new child package of the *Replica_Manager Consolidation* (according to the object type).

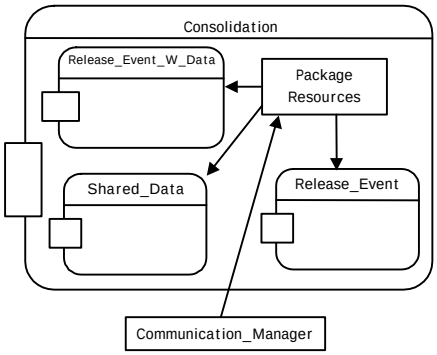

Figure 6.14. *Consolidation* structure

These packages provide the interface for the *Consolidate* protocol (in the *Communication_Manager*) to record the replicas' proposals and to request the execution of the *decide* procedure (Figure 6.15 presents the interface to support *Inter_Group Release Events* objects).

```
1:  generic
2:     Id: FT.Obj_Id_Type;
3:     N_R: FT.Rep_Id_Type;
4:     with procedure Decide (
         Instant_Values    : Framework_Types.Instant_Array_Type;
         Valid_Instants    : Framework_Types.Boolean_Array_Type;
         Rejected_Instants : out Framework_Types.Boolean_Array_Type;
         Release_Instant   : out Ada.Real_Time.Time;
         Release_Ok        : out Boolean);

5:  package
     Replica_Manager.Replication_Support.Consolidation.Release_Event is

6:     procedure Register_Object (Obj_Id : FT.Obj_Id_Type);

7:     procedure Add_Proposed_Release (Obj_Id : FT.Obj_Id_Type;
                                       Rep_Id : FT.Rep_Id_Type;
                                       Instant: A_RT.Time);

8:     procedure Consolidate (Obj_Id : FT.Obj_Id_Type);
       -- ...
```

Figure 6.15. Example of *Replica_Manager Consolidation* support

When the consolidation is performed, the *Receive* package of the *Replication_Support* is used to deliver the consolidated message. Note that, although these objects perform the consolidation processing, the control (the *Consolidate* protocol) is implemented in the *Communication_Manager* layer.

The *Receive* package (Figure 6.16) receives messages either from the *Communication_Manager*, or from the *Consolidate* module. However, as it is necessary to interact with the *Repository* objects through the handler tasks, this package, according to the target object, releases the previously created handler task (see Figure 6.10)

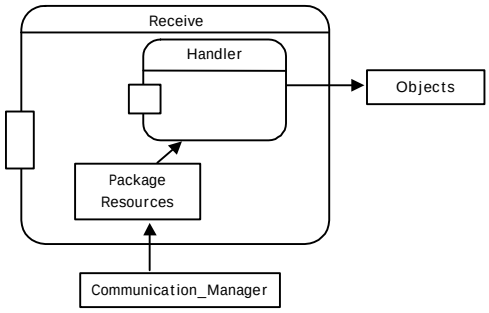

Figure 6.16. *Receive* structure

This handler task (Figure 6.17) is a simple sporadic task, released by a protected object (declared in line 2, used in line 9), which processes the received message (line 10) using a procedure (*Receive_Proc*) that performs the *Replica_Manager Receive_Handler* specified in Chapter 4 (Figure 4.28). Both the *Receive_Proc* procedure and the protected object responsible for the release of the handler task are specified in the parent package (*Receive*, Figure 6.18).

```
1: package body Replica_Manager.Replication_Support.Receive.Handler
     is

       -- ...
2:     Object: Receive_Handler_Release (Id, Size);

3:     task body Handler is
4:        Msg_Type: FT.Message_Type;
5:        Str: FT.Stream (1 .. Size);
6:        Tdeliver: A_RT.Time;
7:     begin
8:        loop
9:           Object.Wait (Msg_Type, Str, Tdeliver);
10:          Receive_Proc (Msg_Type, Str, Obj, Tdeliver);
11:       end loop;
12:    end Handler;

       -- ...
13: end Replica_Manager.Replication_Support.Receive.Handler;
```

Figure 6.17. Implementation of the handler task

```
1:  package Replica_Manager.Replication_Support.Receive is
       -- ...
2:  private
3:     procedure Receive_Proc (Msg_Type: FT.Message_Type;
                               Str: FT.Stream;
                               Obj_Id: FT.Obj_Id_Type;
                               Tdeliver: A_RT.Time);
4:     protected type Receive_Handler_Release (Id: FT.Obj_Id_Type;
                                               Size: Integer) is
5:        procedure Release ( Msg_Type: FT.Message_Type;
                              Str: FT.Stream;
                              Tdeliver: A_RT.Time);
6:        entry Wait (Msg_Type: out FT.Message_Type;
                      Str: out FT.Stream;
                      Tdeliver: out A_RT.Time);
7:     private
8:        Obj_Id: FT.Obj_Id_Type:= Id;
9:        MType: FT.Message_Type;
10:       S: FT.Stream (1 .. Size);
11:       Time: A_RT.Time;
12:       Released : Boolean := False;
13:       pragma Priority (System.Any_Priority'Last - 1);
14:    end Receive_Handler_Release;
       -- ...
15: end Replica_Manager.Replication_Support.Receive;
```

Figure 6.18. *Receive* package

6.2.3. Communication Manager

The *Communication_Manager* hierarchy (Figure 6.19) provides the support for fault-tolerant real-time communication in CAN, as proposed in Chapter 5, and is structured as follows. The *Configuration* package provides the interface to the message streams' configuration data (in the *Application* package).

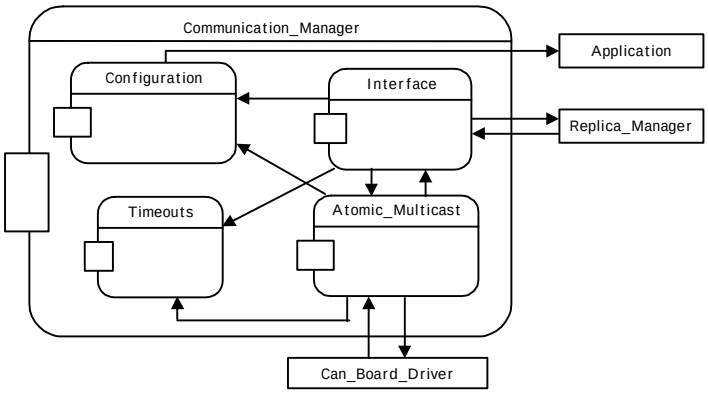

Figure 6.19. *Communication_Manager* structure

The *Interface* hierarchy provides the required interface with the *Replica_Manager*, while the *Atomic_Multicast* hierarchy provides the atomic multicast communication protocols. The *Timeouts* package provides a support for the communication timeouts, since the Ravenscar profile prohibits the use of the Ada timed entry call mechanism.

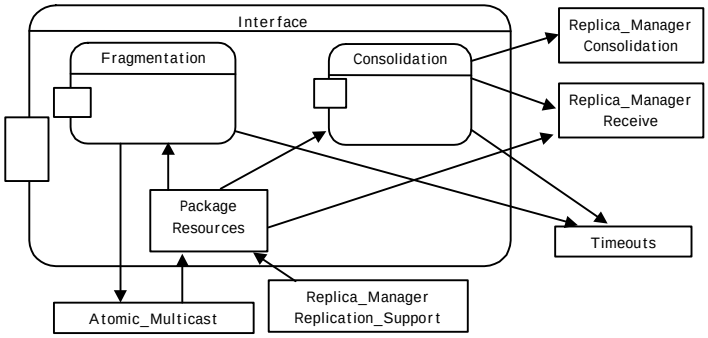

Figure 6.20. *Interface* structure

The *Interface* hierarchy (Figure 6.20), in addition to providing the group communication support to the *Replica_Manager*, also supports the *Fragmentation* and *Consolidation* protocols. The existence of a group communication interface causes a greater complexity in the framework implementation, since it is an extra layer between the replication mechanisms and the communication infrastructure. However, as referred in Chapter 4 (Section 4.5.2), it allows providing an abstract interface, facilitating changes in the communication infrastructure without a greater impact in the replication mechanisms.

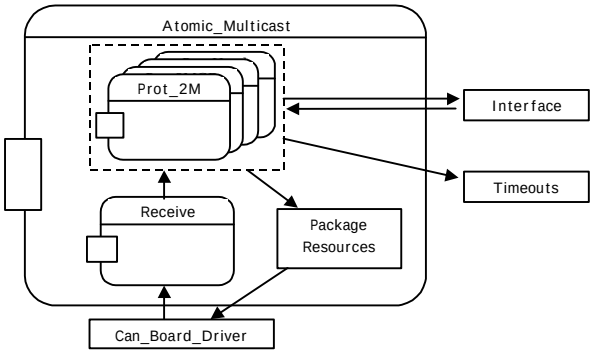

Figure 6.21. *Atomic_Multicast* structure

The *Atomic_Multicast* hierarchy (Figure 6.21) provides the support for the atomic multicast protocols proposed in Chapter 5. The *Receive* package is responsible for performing the filtering of received messages (the Filtering module) and for delivering the message for the correspondent atomic multicast protocol package.

As the protocols proposed in Chapter 5 (atomic multicast, fragmentation and consolidation) are based on delaying the message delivery, it is necessary to provide support for timeouts. However, the Ravenscar profile forbids the use of the timed entry call mechanism. It is therefore necessary to devise a mechanism that allows to start a timer and to, if necessary, abort the timer before completion. This is provided by the *Timeouts* package (Figure 6.22), that can be used to request, launch, and cancel a timer.

```
1:  package Communication_Manager.Timeouts is
2:     package FT renames Framework_Types;
3:     package A_RT renames Ada.Real_Time;
4:     type Timeout_Callback is access procedure (
                  Id: FT.Timeout_Id_Type);
5:     procedure Request_Timer ( Id: out FT.Timeout_Id_Type;
                                 Ok: out Boolean);
6:     procedure Launch_Timer (Id: FT.Timeout_Id_Type;
                               Time: A_RT.Time;
                               Call: Timeout_Callback);
7:     procedure Cancel_Timer (Id: FT.Timeout_Id_Type);
8:     procedure Init;
9:  end Communication_Manager.Timeouts;
```

Figure 6.22. *Timeouts* package

The handling of timers (Figures 6.23 and 6.24) is provided through the use of timer tasks. The package has a set of timer tasks (Figure 6.24), each one being a infinite loop, that in each iteration wait in an entry of a protected object (line 7) and then delays until the requested time has elapsed.

```
1:     protected Timer_Management is
2:        procedure Request_Timer ( Id: out FT.Timeout_Id_Type;
                                    Ok: out Boolean);
3:        procedure End_Timer (Id: FT.Timeout_Id_Type);
4:     private
5:        Available: Used_Timeout_Type := (others => True);
6:        pragma Priority (System.Any_Priority'Last-1);
7:     end Timer_Management;

8:     protected type Timer_Launch is
9:        entry Wait(T: out A_RT.Time; Call: out Timeout_Callback);
10:       procedure Release (T: A_RT.Time; Call: Timeout_Callback);
11:       procedure Cancel;
12:       function Is_Cancelled return Boolean;
13:    private
14:       Time: A_RT.Time;
15:       Callback: Timeout_Callback;
16:       Cancelled: Boolean := False;
17:       Released: Boolean := False;
18:       pragma Priority (System.Any_Priority'Last-1);
19:    end Timer_Launch;
```

Figure 6.23. Timer management

Since it is necessary to support the timer cancellation, the task queries its release object after the end of the delay, to determine if the timeout was cancelled. Only if not cancelled, the task calls the correspondent callback. A protected object (*Timer_Management*, Figure 6.23) is used to record which timers are in use, thus to manage timer requests. For each timer task there is a protected object (*Timer_Launch*, Figure 6.23) that is used to release the task, and to manage its cancellation.

```
1:    task type Timer_Task is
2:       pragma Priority (System.Any_Priority'Last-1);
3:    end Timer_Task;

4:    task body Timer_Task is
          -- ...
5:    begin
          -- ...
6:          loop
7:             Timer_Launch_Objs (Obj).Wait (Time, Call);
8:             delay until Time;
9:             if not Timer_Launch_Objs (Obj).Is_Cancelled then
10:               Call.all (Obj);
11:            end if;
12:            Timer_Management.End_Timer (Obj);
13:         end loop;
14:   end Timer_Task;
```

Figure 6.24. Timer tasks

Figure 6.25 presents an example of the use of the timer tasks (for the *2M* protocol). When a new message is received (line 10), it is necessary to start two timers for the *confirm* and *deliver* timeouts. Lines 12 to 15 present the required processing for the case of the *confirm* timeout. A timer is requested in line 12 (if there are no timers available, an error in the framework is signalled), and recorded by the receive object (line 13). Afterwards, this timer is launched (line 15).

When a duplicate message is received (line 17) it is necessary to restart the timeouts. However, since it is not possible to abort the timer tasks delay, they are cancelled (lines 18 to 21) and new timer tasks are requested (lines 22 to 26). It is clear that this approach is very inefficient, since it requires a greater set of timer tasks than the absolutely required (this will be further discussed in Section 6.4).

6.2.4. Prototype Limitations

This prototype implementation presents some limitations, mainly due to the goal of focusing on Ravenscar specific issues. At the present moment, the prototype does not support different applications in each node (no memory partitioning between the applications and the middleware), and it is targeted only to PC based systems, not supporting heterogeneous distributed systems. It also does not support several replicas of the same component in the same node (although it is not expected that an application would require such support).

```
1:  package body Communication_Manager.Atomic_Multicast.Prot_2M is
2:     package Timers renames Communication_Manager.Timeouts;

       -- ...
5:     protected body Received_Msg_Obj is

6:        procedure Received_Msg (Msg: i527.Message_Data;
                                  Size: CAN.Data_Length;
                                  Time:A_RT.Time) is
7:           Ok: Boolean;
8:           Id: FT.Timeout_Id_Type;
9:        begin
10:          if State = Empty then -- Received Message
11:             State := Unstable;

12:             Timers.Request_Timer (Id, Ok); -- Confirm Timer
                -- ...
13:             Used_Timers (Next_Used_Timer) := Id;
14:             Next_Used_Timer := Next_Used_TImer + 1;
15:             Timers.Launch_Timer (
                   Id,
                   Time + CM_C.Message_Confirm_Delay (Obj_Id),
                   Confirm_Timeout_Callback'Access);

16:             Timers.Request_Timer (Id, Ok); -- Deliver Timer
                -- The same processing for the deliver timer

17:          elsif State = Unstable then -- Received Duplicate

18:             while Next_Used_Timer > 1 loop   -- Cancel Timers
19:                Next_Used_Timer := Next_Used_Timer - 1;
20:                Timers.Cancel_Timer (
                              Used_Timers(Next_Used_Timer));
21:             end loop;

22:             Timers.Request_Timer (Id, Ok); -- New Confirm Timer
23:             Used_Timers (Next_Used_Timer) := Id;
24:             Next_Used_Timer := Next_Used_Timer + 1;
25:             Timers.Launch_Timer (
                   Id,
                   Time + CM_C.Message_Confirm_Delay (Obj_Id),
                   Confirm_Timeout_Callback'Access);

26:             Timers.Request_Timer (Id, Ok); -- New Deliver Timer
                -- ...
27:       end Received_Msg;
          -- ...
28:    end Received_Msg_Obj;

29:    Used: array (FT.Message_Identifier)
            of Boolean := (others => False);
30:    Received_Messages: array (FT.Message_Identifier)
            of Received_Msg_Obj;

31: end Communication_Manager.Atomic_Multicast.Prot_2M;
```

Figure 6.25. Example of timer tasks use

Furthermore, it is not possible to instantiate the generic objects with data types that do not have compile-time defined size (with access types or unconstrained arrays). However, this would only require the final configuration phase to instantiate the generic packages with two extra subprograms, to convert the data type to and from the framework Stream type.

Finally, the implementation of the generic *Shared_Data* objects only supports a single writer. This limitation is the one that is the most restrictive, since it precludes *Inter-Group Shared Data* objects to be written by more than one component. It also does not allow a *Shared Data* object to be written by more than one task of the same component. This restriction is related to the configuration data structures used, which can easily be modified.

6.3. Application Configuration

The configuration of the application is performed by instantiating different generic packages, providing the required parameters, which can easily be performed by a configuration tool. Therefore, application tasks are not changed, although being required to comply with a specific structure.

```
1:  package body Example_Application_Tasks is
2:     package RM_AS renames Replica_Manager.Application_Support;

3:     task body Sensor is
4:        Period: Ada.Real_Time.Time_Span := ...;
5:        Start: Ada.Real_Time.Time := ...;
          -- ...
6:     begin
7:        RM_AS.Wait_Initialization (Sensor_Task_Id);
8:        RM_AS.Register_Task (Sensor_Task_Id);
9:        loop
10:          Start := Start + Period;
11:          RM_AS.Request_Periodic(Start);
             -- ...
12:       end loop;
13:    end Sensor;

14:    task body Controller is
          -- ...
15:    begin
16:       RM_AS.Wait_Initialization (Controller_Task_Id);
17:       RM_AS.Register_Task (Controller_Task_Id);
18:       loop
19:          Device_Event_Data_P.Wait( Device_Event_Obj,
                                       Dev_Data);
          -- ...
20:       end loop;
21:    end Controller;
22: end Example_Application_Tasks;
```

Figure 6.26. Application tasks structure

Figure 6.26 presents the structure of a periodic (*Sensor*) and a sporadic (*Controller*) task (from the example in Chapter 4). Both tasks, before starting the main processing loop must perform two actions. They must wait for the framework initialisation (to guarantee that all data structures are configured) and they must register in the *Replica_Manager*. The *Wait_Initialisation* call (lines 7 and 16) simply blocks the task until the initialisation is completed. The *Register_Task* call (lines 8 and 17) is also required, in order to map the runtime task identifier to the application-specific task identifier.

As presented in Chapter 4 (Section 4.5.3), the periodic task requests its next release through a call to the *Replica_Manager* (line 11), while a sporadic task is released through a *Release* object (line 19).

6.3.1. Object Replacement

The configuration phase is performed by replacing the simple interaction objects with the appropriate objects, providing replication and distribution support. In order to demonstrate the configuration of an application through the use of different interaction objects, Figure 6.27 presents a possible implementation for the *Device_Data* type of the example presented in Chapter 4. This data type is required for a *Release Event With Data* object.

```
1:  package Device_Data_Package is

2:     type Device_Data is ...;
3:     type Device_Data_Array is
            array (FT.Rep_Id_Type Range <>) of Device_Data;

4:     Device_Data_Replicas: FT.Rep_Id_Type := ...;
5:     Device_Obj_Id: FT.Obj_Id_Type := ...;
6:     Device_Obj_Prio: System.Priority := ...;

7:     procedure Device_Data_Decide (
            Values          : Device_Data_Array;
            Value_Instants  : FT.Instant_Array_Type;
            Valid_Values    : FT.Boolean_Array_Type;
            Rejected_Values : out FT.Boolean_Array_Type;
            Release_Value   : out Device_Data;
            Release_Instant : out A_RT.Time;
            Release_Ok      : out Boolean);

8:  end Device_Data_Package;
```

Figure 6.27. Data handling package

The package *Device_Data_Package* specifies the type of the data (line 2) and an anonymous array type (line 3). It also provides the configuration information, such as the number of replicas of the releasing group (line 4), the identifier of the object (line 5) and the objects' ceiling priority (line 6). Finally, a *Decide* procedure (to be executed by the consolidation protocol) is also provided (line 7).

This procedure presents several parameters that must follow a defined interface. The *Values* parameter provides the array of the received proposals. The *Values_Instants* provides the instants when the values were delivered. As the *Values* array has always the size of the maximum number of proposing replicas, a *Valid_Values* array of booleans is used to provide information about which of the array positions have valid proposals.

The other parameters represent the information provided by the procedure to the framework. A *Rejected_Values* parameter is provided to notify the framework that some of the proposed values are to be rejected, thus the *Error_Manager* will be notified. The *Release_Value* parameter provides the decided value, the *Release_Instant* parameter provides the release instant to be given to the related task, and the *Release_Ok* parameter specifies if a decision was performed.

```
1:  package body Example_Application_Tasks is

2:     package DDP renames Device_Data_Package;

3:     package Device_Event_Data_P is
          new Object_Repository.Release_Event_With_Data (
                           Id          => DDP.Device_Obj_Id,
                           Prio        => DDP.Device_Obj_Prio,
                           Data_Type   => DDP.Device_Data );

4:     Device_Event_Obj: Device_Event_Data_P.Data_Release_Obj :=
              Device_Event_Data_P.Request_Data_Release_Obj;
       -- Other Objects and Application Tasks

5:  end Example_Application_Tasks;
```

Figure 6.28. Use of data handling package (before configuration)

```
1: package body Example_Application_Tasks is

2:    package DDP renames Device_Data_Package;

3:    package Device_Event_Data_P is
         new Object_Repository.Inter_Group.Release_Event_With_Data(
             Id              => DDP.Device_Obj_Id,
             Prio            => DDP.Device_Obj_Prio,
             N_Replicas      => DDP.Device_Data_Replicas,
             Data_Type       => DDP.Device_Data,
             Data_Array_Type => DDP.Device_Data_Array,
             Decide          => DDP.Device_Data_Decide);

4:    Device_Event_Obj: Device_Event_Data_P.Data_Release_Obj :=
              Device_Event_Data_P.Request_Data_Release_Obj;
      -- Other Objects and Application Tasks

5: end Example_Application_Tasks;
```

Figure 6.29. Use of data handling package (after configuration)

Figures 6.28 and 6.29 present the use of this *Device_Data_Package*, respectively before and after the configuration phase. The only difference is in line 3, in the instantiation of the generic package, where the *Release Event With Data* generic package only requires the identifier, priority and data type parameters. After configuration, the *Inter-Group Release Event With Data* generic package also requires the number of replicas, the data type array and the *Decide* procedure.

```
1: package body Application.Configuration is
      -- Task Configuration
2:    Periodic_Sensor_Task: Periodic_Task_Configuration_Record := (
                  Comp    => 1,
                  Rep     => 1,
                  Period  => ...,
                  Offset  => ...,
                  WCET    => ...,
                  BCET    => ...;
      -- Object Configuration
3:    Device_Release_Obj: Release_Obj_Configuration_Record := (
                  Obj_Type       => Inter_Group,
                  Data_Release   => True,
                  Releasing_Task => Sensor_Task_Id,
                  Used_Message   => 4);
      -- Message Configuration
4:    Sensor_Message_Node_1 : Message_Stream_Record := (
                  Used               => True,
                  Source_Group       => 1,
                  Source_Replica     => 1,
                  Dest_Groups        => (1, (others => 2)),
                  Obj_Id             => 1,
                  Protocol           => CM.Prot_2M,
                  D_Deliver          => ...,
                  D_Confirm          => ...,
                  D_Deliver_AE       => Null_Time,
                  Need_Consolidation => True,
                  Fragmented         => False);
5:    procedure Init_Configuration is
6:    begin
7:       This_Node_Id := ...;
8:       Periodic_Task_Array := (1 => Periodic_Sensor_Task,
                                 3 => Periodic_Actuator_Task,
                                 others => Null_Periodic_Task);
9:       Release_Obj_Configuration_Array := (
                  1      => Device_Release_Obj,
                  3      => Alarm_Release_Obj,
                  others => Null_Release_Obj);
10:      Message_Streams_Array := (
                  3      => Sensor_Message_Node_1,
                  4      => Sensor_Message_Node_2,
                  5      => Control_Message_Node_1,
                  6      => Control_Message_Node_2,
                  7      => Alarm_Message_Node_2,
                  others => Null_Stream);
         -- Initialization of other Configuration Data Structures
11:   end Init_Configuration;
12: end Application.Configuration;
```

Figure 6.30. Framework configuration example

6.3.2. Framework Configuration

Finally, the *Application.Configuration* package provides the required support to the configuration of the framework. A set of data structures is provided, which must be configured for each application and for each node. These data structures are used by the *Property_Recorder* and *Configuration* modules of the *Replica_Manager* and *Communication_Manager*, respectively. As an example, Figure 6.30 presents a possible configuration of the simple example presented in the Chapter 4. Line 2 presents the specification of a periodic task (*Sensor)*, line 3 the specification of an interaction object (*Device_Release*) and line 4 the specification of a CAN message stream (from the *Sensor* task in node 1). The framework data structures must then be configured in the *Init_Configuration* procedure, which is executed in the initialisation of the framework.

This implementation of the configuration structure is very inefficient and presents redundant information. Furthermore, it causes the configuration phase to be very complex and error prone. These data structures can be modified, and a better approach (with the appropriate tool support) can be easily devised, since only the *Property_Recorder* and *Configuration* modules are dependent of this structure. Nevertheless, the specification and implementation of such configuration tool is out of scope of this thesis.

6.4. Lessons Learnt

From the implementation of the framework prototype, some conclusions about the use of the Ravenscar profile for the implementation of complex middlewares can be easily drawn. It is possible to assess the increase of complexity in the framework code, and also the increase in the used resources, particularly when considering the interaction with the communication-related mechanisms.

Although Ravenscar presents an extensive number of restrictions, only four of them were considered to have a relevant impact in the implementation:

- The restrictions imposed to protected types, namely only one entry and no more than one task simultaneously calling a protected entry;
- Not allowing the dynamic allocation of tasks and protected objects;
- Not allowing dynamic priorities;
- Not allowing timed entry calls (no select statement);

These restrictions have also been previously identified as producing greater complexity and resource usage in Ravenscar compliant implementations (Audsley and Wellings, 2000; Audsley *et al.*, 2000; Vardanega and Caspersen, 2000), and were discussed in the 10th International Real-Time Ada Workshop (Wellings, 2000; Pinho, 2000). It is thus important to evaluate how their inclusion would impact both the complexity and resource usage of the framework implementation.

6.4.1. Protected Entries

In the Ravenscar profile, protected objects are restricted to have at most one protected entry and only one task simultaneously queued. It is clear that these restrictions will

impose an increased complexity of the middleware code. The reason is that the mechanisms that could be performed by a single object must now be jointly performed by a group of objects (for instance, the timer task management and release mechanism presented in Figure 6.23).

However, it is considered that this complexity increase is not sufficient to produce significant difficulties to the implementation of these mechanisms. There will be, however, an increase of the resource usage (higher number of protected objects). Nevertheless, this increase can be counterbalanced by the decrease of the complexity related to the management of the protected objects in the runtime.

6.4.2. Dynamic Allocation of Tasks and Protected Objects

Another restriction in the Ravenscar profile (which was not present in the initial specification (Burns, 1997)) is that the dynamic creation of tasks and protected objects is not allowed, even during the program elaboration. Thus, the initialisation of the middleware becomes more complex, since it is not possible to create just the resources required by the implementation. This can be seen in Figure 6.25, where an array of objects for managing the protocol must be created (line 30), one for each possible message identifier (the *Used* array in line 29 is used to, during initialisation, record the identifiers that will use the protocol).

If dynamic creation of objects were possible, only the message identifiers using the protocol would have a related object. Although not increasing the code complexity, this restriction increases the amount of resources required for the implementation.

6.4.3. Dynamic Priorities

A restriction with a significant impact in the implementation is the impossibility of dynamically changing the priority of a task. The main problem arises during the interaction of protected objects with communications. Tasks processing the messages' reception have to update values or to release entries in the protected objects of the *Repository*. However, it must be guaranteed that the priority of the task is not greater than the ceiling priority of the called object.

If dynamic task priorities were allowed, it would be easy to change the task priority to the ceiling priority of the object, before making the call. Since this is not possible, each object requiring the reception of communication streams has to declare its own handler task, which consequently increases the code complexity and the resource usage by the framework.

6.4.4. Timed Entry Calls

Another restriction with a relevant impact in the complexity and resource usage of the implementation is the impossibility of using the timed entry call mechanism of Ada. This mechanism allows a task to abort a protected entry call where it is suspended, when a specific time is reached. As this mechanism provides an efficient support for managing

timeouts, it would not be necessary to use the complex and inefficient mechanism presented in Section 6.2.

Nevertheless, it is considered that due to the nature of the delays used in the protocols proposed in Chapter 5, the most efficient approach would be to use a high resolution hardware timer. This timer could be reprogrammed each time a new timeout was required (as is usually done for task management), without the overhead related to the timed entry call mechanism.

6.4.5. Final Considerations

From this evaluation, it is possible to conclude that the Ravenscar profile, by decreasing the complexity and overhead of the Ada runtime, introduces a greater complexity and overhead in applications requiring distribution and replication support. However, within the restrictions that were found to have some impact, only dynamic task priorities and dynamic allocation of tasks and protected objects are considered to be useful in the implementation of the prototype. The availability of these mechanisms would reduce the overhead and resource usage of the framework.

The restrictions on protected entries do not present a relevant impact, since more complex mechanisms can be built with this basic block when needed. The lack of the timed entry call mechanism, although decreasing the related overhead, would not present a significant improvement, considering the nature of the required timeouts for the communication protocols.

The introduction of a transparent and generic approach allows applications to abstract from the requirements of replication and distribution, thus becoming simpler to develop. Nevertheless, this approach increases the complexity and resource usage of the framework, since the mechanisms for managing generics together with replication and distribution become more complex.

Nonetheless, the same reasoning that is applied to the Ada tasking model can also be applied to the framework. As the framework is not application-specific, it does not have to be designed for each application, thus much more attention can be given to its implementation. Note that, in the absence of the framework, the mechanisms of replication and distribution management would have to be implemented in the application. Although such approach could be more efficient, the application itself would be more complex, thus more difficult to develop and maintain.

6.5. Summary

This chapter presented the main issues related to the prototype implementation of the proposed framework for the development of fault-tolerant real-time applications conforming to the Ravenscar profile. Its main goal was to assess the expressiveness of the Ravenscar profile for the development of the mechanisms required by the framework.

From the implementation, it is considered that some of the restrictions present some impact, but that in general, the profile presents a suitable model for the development of

these mechanisms. It is also considered that by producing a more efficient runtime, the profile can compensate the greater complexity it causes in applications.

It is also clear that the proposed transparent and generic approach causes a greater complexity and resource usage. However, it is considered that the simpler programming model provided to applications in the presence of replication and distribution counterbalances these issues, as applications become simpler to develop and maintain.

Chapter 7

Conclusions

7.1. Introduction

This thesis proposes a transparent and generic programming model for the development of fault-tolerant real-time applications, conforming to the pre-emptive fixed priority computational model, and considering the utilisation of Commercial Off-The-Shelf (COTS) components. The main advantage of this model is that it allows the application to be developed with a focus on the requirements of the controlled system, abstracting the system developer from the implementation details of replication and distribution.

In order to provide such a model, it was considered that a suitable approach was to support the transparent replication of software components, considering a close integration of the programming mechanisms for replication and distribution with the underlying communication infrastructure. Concerning the underlying technologies, the focus was given to the support of Ada 95 (ISO/IEC, 1995) applications conforming to the Ravenscar profile (Burns, 1997), where replication and distribution are supported by a communication infrastructure based on the Controller Area Network (CAN) (ISO, 1993).

The emergence of the Ravenscar profile is creating an increasing eagerness towards the use of Ada in applications with fault tolerance and hard real-time requirements. The Ravenscar profile allows the use of the pre-emptive fixed priority computational model in application areas where the cyclic executive model has traditionally been the preferred approach. Nevertheless, it is also considered that further studies are still necessary for its use to support replicated and distributed systems. The interaction between multitasking software and replication introduces new problems, particularly for the case of a transparent and generic approach.

CAN was originally developed as an in-vehicle communication network. However, due to its real-time characteristics, CAN is increasingly used as a communication infrastructure for computer control systems. It is generally considered that CAN guarantees atomic multicast properties, through its extensive error detection/signalling mechanisms. Nonetheless, there are some well identified error situations where messages can be delivered in duplicate by some receivers or delivered only by a subset of the receivers. This misbehaviour may be disastrous when CAN networks are used to support replicated applications. It is therefore important to provide the adequate protocols to support replicated application components, whilst maintaining the real-time properties of CAN message transfers.

In order to provide a suitable framework for the development of fault-tolerant real-time applications, the major requirements imposed to such applications were identified. Based on these requirements, a Hard Real-Time Subsystem was specified to support to fault-tolerant real-time applications, within the DEAR-COTS architecture (Veríssimo *et al.*, 2000b).

This thesis proposes a framework for the development of fault-tolerant real-time applications, based on the transparent replication of application components. A set of generic task interaction objects is provided, which are to be used as the basic building blocks for application development, interfacing the application with a middleware layer providing the replication and distribution support. These objects are responsible for providing the required transparency during the application development.

As the integration of replication and distribution mechanisms with the communication infrastructure was also intended, a set of atomic multicast and consolidation protocols is also proposed. These protocols guarantee fault-tolerant and real-time communication on top of a CAN network. In addition, they maintain the predictability of CAN message transfers, in spite of CAN inconsistent message deliveries and considering the possible occurrence of temporary periods of network inaccessibility.

This thesis also discusses some of the lessons learnt during the implementation of a prototype of the proposed framework. The main goal of this implementation was to assess the expressiveness of the Ravenscar profile for the framework development, considering the complexity associated to the support of replication and distribution, particularly considering the proposed transparent and generic approach.

7.2. Research Contributions

This thesis provides some relevant contributions to the development of fault-tolerant real-time applications. The introduction of a generic and transparent approach (considering COTS components and multitasking replicas) allows application development to focus on the requirements of the controlled system, without the need to implement the low-level mechanisms of replication and distribution. Therefore, applications become easier to develop and maintain. This section tries to summarise the most relevant contributions of this thesis to the development of computer control applications.

7.2.1. DEAR-COTS Hard Real-Time Subsystem

In order to provide an adequate environment for computer control systems, this thesis specified the Hard Real-Time Subsystem (HRTS) of the DEAR-COTS architecture, responsible for providing a distributed environment intended for the transparent replication of real-time applications. This subsystem addresses requirements identified as being important in current computer control systems: real-time, fault tolerance, genericity, transparency and interconnectivity.

The use of a multitasking environment is proposed to support real-time applications, being the fault tolerance issues addressed through the transparent replication of application components. The subsystem also addresses the requirements of genericity

and transparency through the provision of a framework for the development of fault-tolerant real-time applications. As current computer control applications are also required to interconnect with other levels of the system, the HRTS also provides support for the interaction of the computer control applications with the DEAR-COTS Soft Real-Time Subsystem (STRS).

7.2.2. Transparent Framework for Application Replication

This thesis provides an abstraction for application replication, based on the concept of component, in order to support the transparent replication of applications. This concept allows applications to be configured only after being developed, thus allowing applications to be developed without considering replication and distribution issues. By using this approach, applications become easier to develop and maintain.

Although the goal was to transparently manage distribution and replication, it was considered that a completely transparent use of these mechanisms would introduce unnecessary overheads and difficulties for checking the real-time and fault tolerance properties of applications. Therefore, object-based resources (which transparently manage replication and distribution issues) were introduced to enable their use by the system developer (transparent approach). Later, in a configuration phase, the system developer configures the application components and their replication level, and allocates the application tasks and resources in the system.

This means that during the application development there is no consideration on how the application will be replicated or distributed. This methodology allows the system developer to abstract from replication and distribution issues, focusing on the requirements of the controlled system.

During the application configuration phase, transparency is only considered at the mechanisms level. Basically, this means that the application code is not constrained by the low-level details of replication and distribution mechanisms. On the other hand, by performing object replacement during configuration, the full behaviour of application (considering replication and distribution) is controlled and predictable. This allows the off-line determination of the real-time and fault tolerance properties of applications, which is of paramount importance in computer control systems.

Such object replacement is achieved by providing a repository of task interaction objects, which map the usual task interaction mechanisms available in hard real-time systems. The main advantage of this approach is that, by providing objects with different capabilities but with the same public interface, applications can be configured simply by replacing the used interaction objects, while during development applications just need to use objects without replication or distribution capabilities.

This object repository provides objects with different capabilities, namely with distribution support, replication support, or both. These objects provide the interface to a middleware layer, responsible for implementing the replication and distribution mechanisms. By reducing the set of mechanisms that need to be supported at the object level, an additional advantage is obtained, since the upgrade of the repository (to provide extra interaction objects) is simplified.

7.2.3. Fault-Tolerant Real-Time Communication

The replication and distribution of application components requires the communication infrastructure to support atomic multicast and replica consolidation mechanisms. Therefore, in this thesis a set of protocols is proposed to provide such mechanisms on top of CAN networks, while at the same time guaranteeing the real-time properties of CAN. These protocols address the problem of CAN inconsistency in message transfers, guaranteeing the required fault tolerance properties. One of the CAN characteristics that is considered in these protocols is that it may be disturbed by temporary periods of network inaccessibility (periods during which the network is unreachable due to on-going error detection and recovery mechanisms).

For the atomic multicast of messages, a set of different protocols is proposed, each one with different behaviours in the presence of errors. One, the *IMD* protocol, addresses the problem of inconsistent duplicates, without introducing extra overheads in the communication network. A second protocol, the *2M* protocol, addresses both duplicate and omission inconsistencies. It only requires the transmission of an extra message (without data) in error-free situations. Only in the presence of an inconsistent message omission, extra messages will be transferred in the bus, guaranteeing that none of the receivers will deliver the message. Finally, the *2M-GD* protocol offers a similar behaviour, but it also guarantees that all receivers of a message will deliver the message, even in the presence of an inconsistent message omission, if at least one correct node received the message.

For the consolidation of replicated messages, the *Consolidate* protocol, supported by the underlying atomic multicasts, is proposed. This protocol allows the implementation of replica agreement mechanisms, and does not require any extra messages in the bus.

Another protocol, the *Fragmentation* protocol, is also proposed to allow the transparent transmission of data units larger than the maximum size of the data field of CAN frames.

To guarantee the real-time properties of the CAN communication, a set of pre-run-time schedulability conditions was developed for all the proposed protocols, making it possible to determine the worst-case and best-case delivery times of messages. Since these conditions follow the current approach for response time analysis of CAN communications, they can be used for determining the schedulability of the applications, without the need to develop different schedulability analysis methods.

7.2.4. Evaluation of the Ravenscar Restrictions

As it is considered that a framework supporting replication and distribution of Ravenscar applications must also conform to the profile restrictions, a prototype implementation of the proposed framework was also developed. The goal of this implementation was to assess if the profile has the required expressiveness for the implementation of the proposed mechanisms, particularly considering the genericity and transparency requirements. From this implementation it is concluded that, although some of the Ravenscar imposed restrictions cause a more complex implementation and greater resource usage, in general the profile provides a suitable approach for the development of fault-tolerant real-time applications. Particularly, from the set of restrictions that were

identified to have impact on the framework's development, only the impossibility of using tasks with dynamic priorities and dynamic allocation of tasks and protected objects were considered to have a significant impact.

Furthermore, it is clear that the transparent and generic approach introduces some additional complexity and resource usage in the framework development. Nevertheless, it is considered that a positive trade-off is obtained, since a clearer and simpler programming model is provided to the application development, and as the framework is not application-specific, much more attention can be given to its implementation.

7.3. Future Work

The proposed programming model and associated replication/distribution mechanisms are considered to be an adequate tool for the development of computer control applications. However, new research directions have also been identified. In this section some improvements that can be applied to the results of this thesis, and future work that can be performed, are described.

1. Throughout the lifetime of a system it is often necessary to change the operational mode of the application (mode changes), in order to cope with a different set of requirements. The proposed framework provides a limited support to mode changes, through the support for application reconfiguration (at the component level). It would be interesting to provide a more extensive support to mode changes, by introducing the concept of a mode in the framework. This can be achieved by providing different application configuration data structures for different modes of operation and by integrating mode changes' support directly in the middleware.
2. In the framework, it is considered that error recovery mechanisms are implemented at the application level, since their efficient implementation is dependent on the application-specific structure and behaviour. Therefore, the framework provides mechanisms for application-specific modules to be notified by the error detection mechanisms implemented and to allow the reconfiguration of the application. It would be important to allow applications to tailor the behaviour of the framework, enabling reconfiguration procedures to be executed upon error detection. This would allow reconfiguration to be performed at the middleware level. Furthermore, in order to support application error recovery, it would be interesting to assess how the provided interaction objects could be used for state recovery operations.
3. At the communications level it is clear that the pre-run-time schedulability conditions provided for the response time analysis are somehow pessimistic, since they always assume the worst-case behaviour, even in cases where such behaviour can be demonstrated to have a negligible probability. It would be important to further evaluate the level of pessimism, in order to provide a less pessimistic analysis, while at the same time preserving the guarantees required by fault-tolerant real-time applications (as is briefly outlined in the Annex).
4. System configuration and component allocation in a replicated/distributed system is a complex task, since there is a trade-off between system efficiency and system reliability. Furthermore, when distributed systems are considered, the allocation of components to the different nodes is a difficult task, since the change of a single component may change the overall system properties. Therefore, the existence of a

configuration tool could ease the assessment of the system behaviour, in particular in what concerns the fault tolerance and real-time properties of the system. The tool would ease the reconfiguration of the system, since it would provide the required analysis to determine the impact of the alterations in the overall behaviour of the system. The direct interaction of the tool with the configuration of the application program would also reduce the possibility of configuration errors.

5. Finally, it was concluded that while the Ravenscar profile reduces the complexity and overheads of the runtime, it also introduces some additional complexity and higher resource consumption. It would be worthwhile to further investigate how much is gained from applying certain restrictions, at the cost of an increased complexity. To achieve this, a suitable platform (compiler and runtime) is required, in order to obtain relevant measures of performance.

Annex

CAN Behaviour in the Presence of Errors

A.1. Introduction

This annex presents a study (Pinho *et al.*, 2000a) performed in order to evaluate the behaviour of CAN networks in the presence of either bus or network interface errors. The results emphasise that, in the presence of temporary periods of network inaccessibility (periods during which nodes cannot communicate with each other, due to the existence of on-going error detection and recovery mechanisms), a CAN network is not able to provide different integrity levels to the supported applications, since errors in low priority messages interfere with the response time of higher-priority message streams. Furthermore, CAN is also not resilient to transceiver errors, since they can lead to large inaccessibility periods. However, a pessimism analysis also demonstrates that, for less strict failure assumptions, CAN may be used as the communication infrastructure for fault-tolerant real-time applications.

This annex is structured as follows. Section A.2 presents the analysis of a CAN network example, where temporary periods of network inaccessibility are considered. The chosen example is based on the SAE benchmark (SAE, 1993), which allows the comparative analysis of the results with a previously available study (Tindell *et al.*, 1995). Afterwards, Section A.3 evaluates how severe is the pessimism of the analysis presented in Section A.2, related to the identified sources of inaccessibility-related pessimism.

A.2. SAE Benchmark

This SAE benchmark specifies a set of messages that must be transferred, considering network data rates of: 125 Kbit/sec, 250 Kbit/sec, 500 Kbit/sec and 1 Mbit/sec. A simplification of this benchmark for the case of CAN networks was presented in (Tindell *et al.*, 1995), where the number of message streams is drastically reduced by piggybacking groups of data messages in single Data Frames, whenever possible. This simplification allows a reduction of the overall network load, due to the removal of the messages' overhead. Table A.1 presents the resulting set of message streams, ordered by decreasing priorities.

Table A.1. SAE benchmark

S_i	C_i (bytes)	T_i (ms)	D_i (ms)	S_I	C_i (bytes)	T_i (ms)	D_i (ms)
A	1	1000	5	J	2	10	10
B	2	5	5	K	1	100	20
C	1	5	5	L	4	100	100
D	2	5	5	M	1	100	100
E	1	5	5	N	1	100	100
F	2	5	5	O	3	1000	1000
G	6	10	10	P	1	1000	1000
H	1	10	10	Q	1	1000	1000
I	2	10	10				

Table A.2 presents the response time and the network load considering the message stream set of Table A.1 (evaluated using equations (5.1) and (5.7), respectively). The *0 errors* assumption is the assumption considered in (Tindell *et al.*, 1995), although with a slight difference: in (Tindell *et al.*, 1995) the authors assume that a message could be blocked by messages with 8 data bytes, although there is no such message in the benchmark. Thus, the response times presented in the 1st column of Table A.2 are slightly smaller than those presented in (Tindell *et al.*, 1995).

Table A.2. Response time of messages (125 Kbit/sec)

Msg.	Response Time (ms)				
	0 errors	1 error	2 errors	3 errors	Transc. Error
A	1.368	2.416	3.464	4.512	18.136
B	1.952	3.000	4.048	5.096	18.720
C	2.456	3.504	4.552	6.184	21.560
D	3.040	4.088	5.136	7.272	24.160
E	3.544	4.592	7.312	8.360	28.672
F	4.128	5.176	8.400	9.448	33.952
G	4.864	8.672	9.720	10.768	43.712
H	5.368	9.176	10.224	14.920	54.176
I	8.712	9.760	14.960	18.768	60.040
J	9.296	10.344	18.888	19.936	78.536
K	9.800	18.928	19.976	29.104	99.288
L	10.456	19.584	20.632	29.760	100.448
M	19.040	20.088	29.216	30.264	110.272
N	19.544	28.672	29.720	38.848	119.360
O	20.048	29.176	30.224	39.352	120.368
P	28.632	29.680	38.808	39.856	128.952
Q	28.656	29.704	38.832	39.880	128.976
U (%)	80.279	81.327	82.375	83.423	80.280

In this table, all the message streams that may miss their deadlines are highlighted. A network date rate of 125 Kbit/sec is considered (which leads to the highest network load) together with the following set of error assumptions:

- from 0 to 4 bus errors in each 100 ms time interval, resulting from a bit error rate of approximately 10^{-4} (for a data rate of 125 Kbit/sec, this results in considering 0-4 errors within 12500 bits), which is an expectable range for bit error rates in aggressive environments;
- a single transceiver failure (causing 16 failed transmissions), leading the related node to an Error-Passive state.

As it can be seen, a set of message streams that is completely schedulable without considering temporary periods of network inaccessibility, is no longer schedulable even assuming low bit error rates. The simple consideration of one bit error within an interval of 100 ms leads to a faulty timing behaviour in two of the message streams. Network load does not increase significantly since just 0-4 bus errors are considered within each interval of 100 ms.

An interesting result is that, conversely to what is common in fixed priority systems, the first message stream to miss its deadline is not the lowest priority one, but one with an intermediate priority (message streams **F** and **J**). The reason for this unusual behaviour is that the occurrence of a bus error results in the same inaccessibility period, whatever the message stream being considered. Therefore, message streams with smaller response times will have the larger percentage increase on their message's duration, resulting that the most penalised message streams will be the ones with the smallest slack time (smallest difference between response time and deadline).

Table A.3. Response time of messages (250 Kbit/sec)

Msg.	Response Time (ms)					
	0 errors	1 error	2 errors	3 errors	4 errors	Transc. error
A	0.684	1.208	1.732	2.256	2.780	9.068
B	0.976	1.500	2.024	2.548	3.072	9.360
C	1.228	1.752	2.276	2.800	3.324	9.904
D	1.520	2.044	2.568	3.092	3.616	10.992
E	1.772	2.296	2.820	3.344	3.868	11.828
F	2.064	2.588	3.112	3.636	4.160	12.624
G	2.432	2.956	3.480	4.004	4.528	13.576
H	2.684	3.208	3.732	4.256	4.780	14.272
I	2.976	3.500	4.024	4.548	5.072	14.816
J	3.268	3.792	4.316	4.840	6.744	16.780
K	3.520	4.044	4.568	5.092	6.996	17.324
L	3.848	4.372	4.896	6.800	7.324	17.652
M	4.100	4.624	6.528	7.052	7.576	17.904
N	4.352	4.876	6.780	7.304	7.828	18.156
O	4.604	5.128	7.032	7.556	8.080	18.408
P	4.856	6.760	7.284	7.808	8.332	18.660
Q	4.868	6.772	7.296	7.820	8.344	18.672
U (%)	40.140	40.664	41.188	41.712	42.236	40.140

This unusual behaviour is present even in the case of errors during the transfer of lower-priority messages. In this case, the mechanism needed to recover from the error prevents higher-priority messages from being transmitted.

Thus, in the presence of bus errors, a CAN fieldbus network is not able to provide different integrity levels to the supported applications, since errors in low priority messages interfere with the response time of higher-priority messages. This result proves that the scheduling of messages in the presence of errors is not equivalent to the scheduling of fixed priority systems in overload conditions (where tasks/messages with lower priorities do not interfere with the response time of higher-priority tasks/messages).

Table A.3 analyses the same scenario for the case of a network data rate of 250 Kbit/sec. Obviously, as the duration of messages is reduced by 50%, the overall network load is also reduced by 50%. As a consequence, considering such reduced network load for this particular set of message streams (with harmonic periodicities), the message stream set is now schedulable for the considered failure assumptions.

Also included in Tables A.2 and A.3 is the consideration of a single transceiver failure. In this situation, higher-priority messages miss their deadlines. It is interesting to notice that the response time of message stream **A** increases 13 times when a transceiver error is considered, but the network load does not suffer any increase. That is due to the assumption of an extremely low failure rate for transceivers, leading to a negligible increase in the network load.

Table A.4. Response time of messages considering one transceiver error

Msg.	Response Time (ms)			
	1 Mbit/sec	500 Kbit/sec	250 Kbit/sec	125 Kbit/sec
A	2.267	4.534	9.068	18.136
B	2.340	4.680	9.360	18.720
C	2.403	4.806	9.904	21.560
D	2.476	4.952	10.992	24.160
E	2.539	5.496	11.828	28.672
F	2.612	5.768	12.624	33.952
G	2.704	6.098	13.576	43.712
H	2.767	6.224	14.272	54.176
I	2.840	6.370	14.816	60.040
J	2.913	6.516	16.780	78.536
K	2.976	6.642	17.324	99.288
L	3.058	6.806	17.652	100.448
M	3.121	6.932	17.904	110.272
N	3.184	7.058	18.156	119.360
O	3.247	7.184	18.408	120.368
P	3.310	7.310	18.660	128.952
Q	3.313	7.316	18.672	128.976
U (%)	10.035	20.070	40.140	80.280

It is also clear that transceiver errors are extremely penalising for the scheduling of message stream sets, since a node with an erratic transceiver may signal up to 16 errors, preventing other nodes from accessing the bus.

Finally, Table A.4 analyses a scenario where there are no bus errors; instead, a single transceiver error for different network date rates is considered. It can be seen that, even without bus errors, the message stream set is only schedulable at 1Mbit/sec, that is, it is only schedulable for a network load as low as 10%.

A.3. Pessimism Analysis

Up to this moment, a set of worst case error assumptions has been assumed. It is, therefore, important to evaluate how severe is the pessimism inherent to the proposed approach. Considering the worst-case analysis presented in Chapter 5 (Section 5.4.1, equations (5.1) and (5.7)), some sources of inaccessibility-related pessimism can be identified:

- It has been assumed that the worst case error assumptions are always present. That is, that all the n_{bus} and n_{transc} are present in one round of messages;
- It has been assumed that bus errors are always detected in the last bit of the longest Data Frame;
- It has been also assumed that an Error Frame has always the maximum number of bits.

Although this set of assumptions is necessary for worst-case evaluations, it is also correct to say that it contains an important level of pessimism. In order to assess the impact of each one of these factors in the pessimism of the response time analysis, the following set of equations has been used:

$$Ina_{bus} = \mathbf{A} \times n_{bus} \times \left\lceil \frac{t_m + C_m}{T_{bus}} \right\rceil \times \left(\mathbf{B} \times C_{MAX} + (0.7 + 0.3 \times \mathbf{C}) \times C_{error} + C_{IFS} \right) \quad \text{(A.1)}$$

$$U_{ina\_bus} = \mathbf{A} \times n_{errors} \times \frac{\left(\mathbf{B} \times C_{MAX} + (0.7 + 0.3 \times \mathbf{C}) \times C_{error} + C_{IFS} \right)}{T_{bus}} \quad \text{(A.2)}$$

where **A** stands for the percentage of assumed errors in a period of T_{bus} (maximum of 4 errors), **B** stands for the percentage of the longest message to be transmitted and **C** is the percentage of the error frame length. As Error Frames have at least 14 bits, **C** can only be applied to the remaining 6 bits.

Figures A.1 and A.2 illustrate the impact of each one of these factors on the network load and on the response time of Message Stream **F** (Message Stream **F** is chosen for the analysis, since it is the one with the smallest slack time). The variation of parameter **A** is made considering a value of 1 for parameters **B** and **C**. Variation of parameters **B** and **C** is made considering the existence of 3 bus errors.

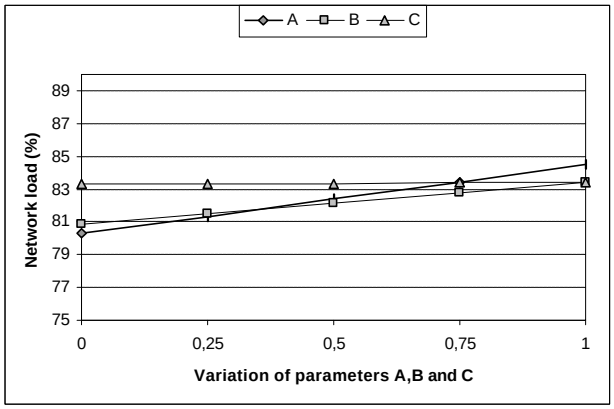

Figure A.1. Variation of the network load with parameters **A**, **B** and **C**

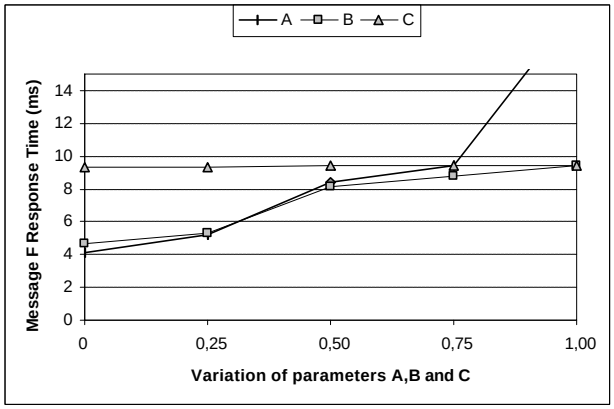

Figure A.2. Variation of message stream **F** response time with parameters **A**, **B** and **C**

As it can be seen, the parameter that has the strongest influence is the bus error rate. However, network load is only slightly penalised by errors. That is due to the assumption of a low failure rate in the network, since just 0-4 bus errors are considered within each interval of 100 ms.

The analysis presented in Section A.2 showed that message stream **F** is only schedulable in the absence of errors (Table A.2). In Figure A.2, such non-schedulability is reflected in the sudden increase of its response time, which is due to the increasing interference of higher-priority message streams (with 5 ms period). As shown in Figure A.2, the response time of this message stream is highly dependent on the assumed error rate, and also on the assumed inaccessibility time caused by such errors. However, with smaller periods of temporary inaccessibility, the message stream is schedulable even for larger error rates.

In order to assess the pessimism of considering that errors always occur in the last bit of the largest message, Figure A.3 shows the impact of parameter **B** for different bus errors assumptions (parameter **A**).

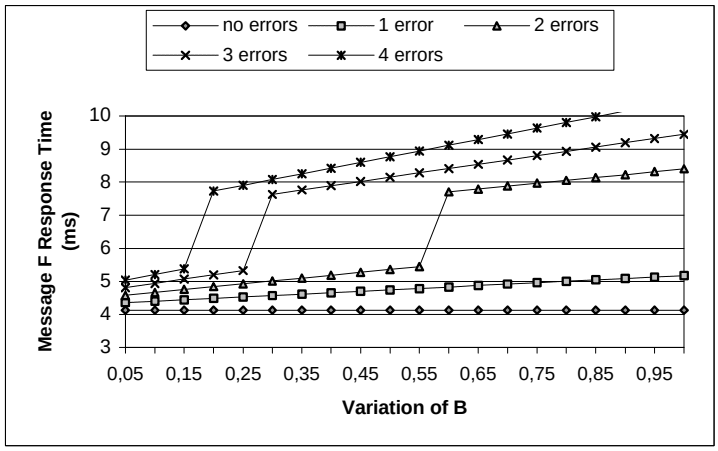

Figure A.3. Variation of message stream **F** response time with parameter **B**

Considering just one bus error per 100 ms, when parameter **B** is set to 0.5, the response time of message stream **F** will be just 4.744 ms, which compared to 5.176 ms (Table A.2) gives a reduction of 8%. Furthermore, for this scenario, message stream **F** becomes schedulable. If greater error rates are assumed, the decrease of the response time is even more relevant.

The scenario is quite realistic since there is only one message that takes 6 bytes of data, while the majority of the messages have 1 or 2 bytes of data. Therefore, the inherent pessimism of worst case analysis can be reduced, if less strict failure assumptions are accepted.

A.4. Summary

This annex presents a study performed in order to evaluate the behaviour of CAN networks, considering temporary periods of network inaccessibility. From the achieved results, it can be concluded that message streams with smaller response times will have the larger relative increase on their duration, resulting that the most penalised message streams will be the ones with the smallest slack time.

An important conclusion is also that a CAN fieldbus network is not able to provide different integrity levels to the supported applications, since errors in low priority messages interfere with the response time of higher-priority messages. This result proves that the scheduling of messages in the presence of errors is not equivalent to the scheduling of fixed priority systems in overload conditions (where tasks/messages with lower priorities do not interfere with the response time of higher-priority tasks/messages).

Another conclusion is that CAN is not resilient to transceiver errors, since they can lead to large inaccessibility periods. A faulty network interface can cause a sequence of (at most) 16 erroneous messages, causing the network to become unreachable for all nodes for a large period.

The inherent pessimism of the proposed analysis has also been evaluated, and it is concluded that the message set response times' are highly dependent on the considered error rates and inaccessibility periods. It is also concluded that assuming smaller periods of temporary network inaccessibility, the system becomes schedulable even for greater bus error rates. This assumption is quite realistic, since the majority of the considered messages carry only 1 or 2 bytes of data. Therefore it is considered that, for less strict failure assumptions, CAN may be used as the communication infrastructure for fault-tolerant real-time applications.

References

ANSI (1983). Reference Manual for the Ada Programming Language. ANSI/MIL-Std-1815a.

Aonix. (1998). ObjectAda/Raven. Aonix Inc.

Audsley, A., Burns, A., Richardson, M., Tindell, K., and Wellings, A. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. In *Software Engineering Journal*, 8(5):285-292.

Audsley, A. and Wellings, A. (2000). Issues with using Ravenscar and the Ada Distributed Systems Annex for High-Integrity Systems. In *Proc. of the 10th International Real-Time Ada Workshop*, Avila, Spain, September 2000. Ada Letters, XXI(1):33-39, ACM Press.

Audsley, A. Burns, A. and Wellings, A. (2000). Implementing a High-Integrity Executive using Ravenscar. In *Proc. of the 10th International Real-Time Ada Workshop*, Avila, Spain, September 2000. Ada Letters, XXI(1):40-45, ACM Press.

Barrett, P. A., Hilborne, A. M., Bond, P. G., Seaton, D. T., Veríssimo, P., Rodrigues, L. and Speirs, N. A. (1990). The Delta-4 XPA Extra Performance Architecture. In *Proc. 20th Int. Symposium on Fault-Tolerant Computing Systems*, UK, pp. 481-488.

Barrett, P. A., Burns, A., and Wellings, A. J. (1995). Models of Replication for Safety Critical Hard Real-Time Systems. In *Proc. 20th IFAC/IFIP Workshop on Real-Time Programming*, USA, pp. 181-188.

Bondavalli, A., Giandomenico, F. D., Grandoni, F., Powell, D., and Rabejac, C. (1998). State Restoration in a COTS-based N-Modular Architecture. In *Proc. First International Symposium in Object-Oriented Real-Time Distributed Computing*, Japan, pp. 174-183.

Burns, A. (1991). Scheduling hard real-time systems: a review. In *Software Engineering Journal*, 6(3):116-128.

Burns, A. (1997). Session Summary: Tasking Profiles. In *Proc. of the 8th International Real-Time Ada Workshop*, Ravenscar, England, April 1997. Ada Letters, XVII(5):5-7, ACM Press.

Burns, A. and Wellings, A. (1995a). HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems. Elsevier.

Burns, A. and Wellings, A. (1995b). Advanced fixed priority scheduling. In *Real-Time Systems: Specification, Verification and Analysis*, Joseph, M. (Ed.), Prentice Hall, pp. 32-65.

Burns, A. and Wellings, A. (1998). Concurrency in Ada. 2nd Ed. Cambridge University Press.

Cristian, F., Aghili, H., Strong, R. and Dolev, D. (1995). Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Information and Control*, 118(1):158-179.

Dobbing, B. and Burns, A. (1998). The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Proc. of SIGAda'98 Conference*, Washington, USA, pp. 1-6.

Guerraoui, R. and Schiper, A. (1997). Software-based replication for fault tolerance. In *IEEE Computer*, 30(4):68–74.

Hadzilacos, V. and Toueg, S. (1993). Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, Mullender, S. (Ed.), 2nd Ed., Addison-Wesley.

Heras-Quirós, P., González-Barahona, J. and Centeno-González, J. (1997). Programming Distributed Fault Tolerant Systems: The ReplicAda Approach. In *Proc. of Tri-Ada'97*, St. Louis, USA, pp. 21-29.

Hilmer, H., Kochs, H.-D. and Dittmar, E. (1998). A CAN-Based Architecture for Highly Reliable Communication Systems. In *Proc. of the 5th CAN Conference*, San Jose, USA, pp. 6.10-6.16.

IEEE (1995). Portable Operating System Interface: Amendment 2: Threads Extension. IEEE. Std 1003.1c.

Intermetrics (1995). Ada 95 Rationale - The Language - The Standard Libraries. Intermetrics, Inc.

ISO 11898 (1993). Road Vehicle - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. ISO.

ISO 8652 (1987). Reference Manual for the Ada Programming Language. ISO.

ISO/IEC 8652 (1995). Information technology – Programming languages – Ada. Ada Reference Manual. ISO/IEC.

Johnson, S., Jahanian, F., Ghosh, S., VanVoorst, B., and Weininger, N. (2000). Experiences with Group Communication Middleware. In *Proc. International Conference on Dependable Systems and Networks*, New York City, USA, pp. 37-42.

Joseph, M. and Pandya, P. (1986). Finding Response Times in a Real-Time System. In *The Computer Journal*, 29(5):390-395.

Kaiser, J. and Livani, M. (1999). Achieving Fault-Tolerant Ordered Broadcasts in CAN. In *Proc. of the 3rd European Dependable Computing Conference*, Prague, Czech Republic, September 1999, pp. 351-363.

Keickhafer, R. M., Walter, C. J., Finn, A. M., and Thambidurai, P. M. (1988). The MAFT Architecture for Distributed Fault Tolerance. In *IEEE Transactions on Computers*, 37 (4):398-404.

Kienzle, J. (2001). Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming. *Swiss Federal Institute of Technology (EPFL)*, PhD Thesis n. 2393, Switzerland.

Kopetz, H. (1997). Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic Publishers.

Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., and Zainlinger, R. (1989). Distributed Fault-Tolerant Real-Time Systems: The Mars Approach. In *IEEE Micro*, 9(1):25–41.

Laprie, J. L. (1992). Dependability: Basic Concepts and Terminology. Dependable Computing and Fault-Tolerant Systems, Vol. 5. Springer-Verlag.

Leung, J. and Whitehead, J. (1982). On the Complexity of fixed-priority Scheduling of Periodic Real-Time Tasks. In *Performance Evaluation*, 22(4): 237-250.

Livani, M. and Kaiser, J. (1999) Evaluation of a Hybrid Real-Time Bus Scheduling Mechanism for CAN. In *Proc. of the 7th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS ´99)*, San Juan, Puerto Rico, pp. 425-429.

Liu, C. and Layland, J. (1973). Scheduling Algorithms for Multiprograming in Hard-Real-Time Environment. In *Journal of the ACM*, 20(1):46-61.

Locke, C. D. (1992). Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. In *Real-Time Systems Journal*, 4(1):37-53.

Lundqvist, K. and Asplund, L. (1999). A Formal Model of a Run-Time Kernel for Ravenscar. In *Proc. of 6th International Conference on Real-Time Computing Systems and Applications*, Hong Kong, China, pp. 504-507.

Melliar-Smith, P. M., and Schwartz, R. L. (1982). Formal Specification and Mechanical Verification of SIFT: a Fault–Tolerance Flight Control System. In *IEEE Transactions on Computers*, 31(7):616-630.

Miranda, J., Alvarez, A., Arévalo, S. and Guerra, F. (1996). Drago: An Ada Extension to Program Fault-Tolerant Distributed Applications. In *Proc. of Ada-Europe'96 Conference*, Montreux, Switzerland, Lecture Notes on Computer Science 1088, Springer-Verlag, pp. 235-246.

Pinho, L. (2000). Session Summary: Distribution and Real-Time. In *Proc. of the 10th International Real-Time Ada Workshop*, Avila, Spain, September 2000. Ada Letters, XXI(1):14-16, ACM Press.

Pinho L. and Vasques, F. (2000). An Architecture for Reliable Distributed Computer-Controlled Systems. *IFIP Workshop on Distributed and Parallel Embedded Systems (DIPES 2000)*, Paderborn, Germany, October 2000. In *Architecture and Design of Distributed Embedded Systems*. B. Kleinjohann (Ed.), Kluwer Academic Publishers, Boston, April 2001.

Pinho, L., Vasques, F. and Tovar E. (2000a). Integrating Inacessibility in Response Time Analysis of CAN Networks. In *Proc. of the 3rd IEEE International Workshop on Factory Communication Systems*, Porto, Portugal, September 2000, pp. 77-84.

Pinho, L., Vasques, F. and Ferreira, F. (2000b). Programming Atomic Multicasts in CAN. In *Proc. of the 10th International Real-Time Ada Workshop,* Avila, Spain, September 2000, Ada Letters, XXI(1):79:84, ACM Press.

Pinho, L. and Vasques, F. (2001a). Reliable Real-Time Communication in CAN Networks. *Polytechnic Institute of Porto*, Technical Report HURRAY-TR-0109, April 2001. (Submitted to the *IEEE Transactions on Computers*).

Pinho, L. and Vasques F. (2001b). Reliable Communication in Distributed Computer-Controlled Systems. In *Proc. of Reliable Software Technologies - Ada-Europe 2001*. Leuven, Belgium, May 2001, Lecture Notes on Computer Science 2043, Springer, pp. 136-147.

Pinho, L. and Vasques, F. (2001c). Timing Analysis of Reliable Real-Time Communication in CAN Networks. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001, pp. 103-112.

Pinho, L. and Vasques, F. (2001d). Improved Fault-Tolerant Broadcasts in CAN. To appear in the *Proc. of the 8th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'01),* Antibes, France, October 2001.

Pinho, L. Vasques, F. and Wellings, A. (2001a). Replication Management in Reliable Real-Time Systems. *Polytechnic Institute of Porto*, Technical Report HURRAY-TR-0110, March 2001. (Submitted to the *Real-Time Systems Journal*).

Pinho, L. Vasques, F. and Wellings, A. (2001b). The DEAR-COTS Replication Framework. *Work in Progress of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001, pp.17-20.

Poledna, S. (1994). Replica Determinism in Distributed Real-Time Systems: A Brief Survey. In *Real-Time Systems*, 6(3):289-316.

Poledna, S. (1998). Deterministic Operation of Dissimilar Replicated Task Sets in Fault-Tolerant Distributed Real-Time Systems. In *Proc. Dependable Computing for Critical Applications 6*, Grainau, Germany, pp. 103-119.

Poledna, S., Burns, A., Wellings, A., and Barret, P. (2000). Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems. In *IEEE Transactions on Computers*, 49(2):100-111.

Powell, D. (Ed.). (1991). Delta-4 - A Generic Architecture for Dependable Distributed Computing. ESPRIT Research Reports. Springer Verlag.

Powell, D. (1992). Failure Mode Assumptions and Assumption Coverage. In *Proc. of the 22nd Symposium on Fault-Tolerant Computing*, Boston, USA, pp. 386-395.

Powell, D. (1994). Distributed Fault Tolerance – Lessons Learnt from Delta-4. In *Hardware and Software Architectures for Fault Tolerance. Experiences and Perspectives*. Banatre, M. and Lee P. A. (Eds.). Lecture Notes in Computer Science 774. Springer-Verlag, pp. 199-217.

Powell, D. (Ed.). (2001). A Generic Fault-Tolerant Architecture for Real-Time Dependable Systems. Kluwer Academic Publishers.

Powell, D., Bonn, G., Seaton, D., Veríssimo, P. and Waeselynck, F. (1988). The Delta-4 Approach to Dependability in Open Distributed Computing Systems. In *Proc. of the 18th International Symposium on Fault-Tolerant Computing Systems*, Tokyo, Japan, pp. 246-251.

Puente, J, Ruiz, J, and Zamorano, J. (2000). An Open Ravenscar Real-Time Kernel for GNAT. In *Proc. of Ada-Europe'2000 Conference*, Potsdam, Germany, Lecture Notes in Computer Science 1845, Springer-Verlag, pp. 5-15.

Pradhan, D. K. (1996). Fault–Tolerant Computer System Design. Prentice Hall.

Rajkumar, R., Sha, L., Lehoczky, J., and Ramamritham, K. (1995). An Optimal Priority Inheritance Policy for Synchronization in Real-Time Systems. In *Advances in Real-Time Systems*, Son, S (Ed.), Prentice Hall, pp. 249-271.

Rockwell (1997). DeviceNet Product Overview. Publication DN-2.5, Rockwell Automation.

Rodrigues, L., Guimarães, M. and Rufino, J. (1998). Fault-Tolerant Clock Synchronisation on CAN. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 420-429.

Rogers, P. and Wellings, A. (1999). An Incremental Recovery Cache Supporting Software Fault Tolerance. In *Proc. of the Ada-Europe'99 Conference*, Santander, Spain, Lecture Notes on Computer Science 1622, Springer-Verlag, pp. 385-397.

Rufino, J. and Veríssimo, P. (1995). A Study on the Inaccessibility Characteristics of the Controller Area Network. In *Proc. of the 2nd CAN Conference*, London, United Kingdom, pp. 7.12-7.21.

Rufino, J., Veríssimo, P. and Arroz, G. (1999). Design of Bus Media Redundancy in CAN. In *Proc. of the Fieldbus Conference (FeT'99)*, Magdeburg, Germany, pp. 375-380.

Rufino, J., Veríssimo, P., Arroz, G., Almeida, C. and Rodrigues, L. (1998). Fault-Tolerant Broadcasts in CAN. In *Proc. of the 28th Symp. on Fault-Tolerant Computing*, Munich, Germany, pp. 150-159.

Rushby, J. (1996). Reconfiguration and Transient Recovery in State Machines Architectures. In *Proc 26th Symposium on Fault-Tolerant Computing*, Sendai, Japan, pp. 6-15.

RTCA (1992). Software Considerations in Airborne Systems and Equipment Certification. DO-178B/ED-12B, RTCA.

SAE (1993). Class C Application Requirement Considerations. Technical Report J2056/1, SAE.

Schneider, F. (1990). Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. In *ACM Computing Surveys*, 22(4):299-319.

Sha, L. and Goodenough, J. (1990). Real-Time Scheduling Theory and Ada. In *IEEE Computer* 23(4):53-62.

Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990). Priority Inheritance Protocols: An Approach to Real-Time Synchronisation. In *IEEE Transactions on Computers*, 39(9):1175-1185.

Stankovic, J. (1988). Real-Time Computing Systems: the Next Generation. In *Tutorial: Hard Real-Time Systems*, Stankovic, J. and K. Ramamritham (Eds.), IEEE Computer Society Press, Los Alamitos, USA, pp. 14-38.

Tindell, K. and Clark, J. (1994). Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. In *Microprocessors and Microprogramming*, Vol. 40, pp. 117-134.

Tindell, K., Burns, A. and Wellings, A. (1995). Calculating Controller Area Network (CAN) Message Response Time. In *Control Engineering Practice*, 3(8):1163-1169.

Vardanega, T. and Caspersen, G. (2000). Using the Ravenscar Profile for Space Applications. In *Proc. of the 10th International Real-Time Ada Workshop*, Avila, Spain, September 2000. Ada Letters, XXI(1):96-104, ACM Press.

Veríssimo, P., Casimiro, A., and Fetzer, C. (2000a). The Timely Computing Base: Timely actions in the presence of uncertain timeliness. In *Proc. International Conference on Dependable Systems and Networks*, New York, USA, pp. 533–542.

Veríssimo, P., Casimiro, A., Pinho, L. M., Vasques, F., Rodrigues, L., and Tovar, E. (2000b). Distributed Computer-Controlled Systems: the DEAR-COTS Approach. In *Proc. 16th IFAC Workshop on Distributed Computer Control Systems*, Sydney, Australia, pp. 128-135.

Wellings, A. (2000). Session Summary: Status and Future of the Ravenscar Profile. In *Proc. of the 10th International Real-Time Ada Workshop*, Avila, Spain, September 2000. Ada Letters, XXI(1):4-8, ACM Press.

Wellings, A. and Burns, A. (1996). Programming Replicated Systems in Ada 95. In *The Computer Journal*, 39(5):361-373.

Wellings, A. and Burns, A. (1997). Implementing Atomic Actions in Ada 95. In *IEEE Transactions on Software Engineering*, 23(2):107-123.

Wellings, A. J., Beus-Dukic, Lj., and Powell, D. (1998). Real-Time Scheduling in a Generic Fault-Tolerant Architecture. In *Proc. IEEE Real-Time Systems Symposium*, Madrid, Spain, pp. 390-298.

Wolf, T. and Strohmeier, A. (1999). Fault Tolerance by Transparent Replication for Distributed Ada 95. In *Proc. of the Ada-Europe'99 Conference*, Santander, Spain, Lecture Notes on Computer Science 1622, Springer-Verlag, pp. 412-424.

Yeh, Y. (1995). Dependability of the 777 Primary Flight Control System. In *Proc. Dependable Computing for Critical Applications 5*, USA, pp. 1-13.

Zuberi, K. and Shin, K. (1997). Scheduling messages on Controller Area Network for Real-Time CIM Applications. In *IEEE Transactions on Robotics and Automation*, 13(2):310-314.

List of Publications Related to this Thesis

Published in International Journals

1. Pinho, L. and Vasques, F., To Ada or not To Ada: Ada*ing* vs. Java*ing* in Real-Time Systems. In *ACM Ada Letters*, XIX(4):37-43, December 1999.

Submitted to International Journals

2. Pinho, L. Vasques, F. and Wellings, A., Replication Management in Reliable Real-Time Systems. *Polytechnic Institute of Porto*, Technical Report HURRAY-TR-0110, March 2001. (Submitted to the *Real-Time Systems Journal*).

3. Pinho, L. and Vasques, F., Reliable Real-Time Communication in CAN Networks. *Polytechnic Institute of Porto*, Technical Report HURRAY-TR-0109, April 2001. (Submitted to the *IEEE Transactions on Computers*).

Published in Conference Proceedings

4. Pinho, L. and Vasques, F., Multi-µ: an Ada 95 Based Architecture for Fault Tolerance Support of Real-Time Systems. In *Proc. of the ACM SIGAda Annual International Conference (SIGAda'98)*, Washington DC, USA, November 1998, pp. 52-60.

5. Pinho, L. and Vasques, F., Replica Management in Real-Time Ada Applications. In *Proc. of the 9th International Real-Time Ada Workshop*, Tallahassee FL, USA, March 1999. Ada Letters, XIX(2):21:27, ACM Press.

6. Pinho, L., Vasques, F. and Tovar E., Integrating Inacessibility in Response Time Analysis of CAN Networks. In *Proc. of the 3rd IEEE International Workshop on Factory Communication Systems (WFCS'2000)*, Porto, Portugal, September 2000, pp. 77-84.

7. Pinho, L., Vasques, F. and Ferreira, F., Programming Atomic Multicasts in CAN. In *Proc. of the 10th International Real-Time Ada Workshop*, Avila, Spain, September 2000. Ada Letters, XXI(1):79:84, ACM Press.

8. Pinho L. and Vasques, F., An Architecture for Reliable Distributed Computer-Controlled Systems. *IFIP Workshop on Distributed and Parallel Embedded Systems*

(DIPES 2000), Paderborn, Germany, October 2000. In *Architecture and Design of Distributed Embedded Systems*. B. Kleinjohann (Ed.), Kluwer Academic Publishers, Boston, April 2001.

9. Veríssimo, P., Casimiro, A., Pinho, L. M., Vasques, F., Rodrigues, L., and Tovar, E., Distributed Computer-Controlled Systems: the DEAR-COTS Approach. In *Proc. 16th IFAC Workshop on Distributed Computer Control Systems,* Sydney, Australia, pp. 128-135.

10. Pinho, L. and Vasques, F., Merging Reliability and Timeliness Requirements in Distributed Computer-Controlled Systems. In *Proc. of the 9th International Conference on Real-Time Systems (RTS 2001)*, Paris, France, March 2001, pp.79-92.

11. Pinho, L. and Vasques, F., Atomic Multicast Protocols for Reliable CAN Communication. In *Proc. of the 19th Brazilian Symposium on Computer Networks (SBRC 2001)*, Florianópolis, Brazil, May 2001, pp. 194-209.

12. Pinho, L. and Vasques F., Reliable Communication in Distributed Computer-Controlled Systems. In *Proc. of Reliable Software Technologies - Ada-Europe 2001*. Leuven, Belgium, May 2001, Lecture Notes on Computer Science 2043, Springer, pp. 136-147.

13. Pinho, L. and Vasques, F., Timing Analysis of Reliable Real-Time Communication in CAN Networks. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001, pp. 103-112.

14. Pinho, L. Vasques, F. and Wellings, A., The DEAR-COTS Replication Framework. *Work in Progress of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001, pp.17-20.

To appear in Conference Proceedings

15. Pinho, L. and Vasques, F., Improved Fault-Tolerant Broadcasts in CAN. To appear in the *Proc.* of the *8th IEEE Conference on Emerging Technologies and Factory Automation (ETFA'01)*, France, October 2001.