View metadata, citation and similar papers at core.ac.uk

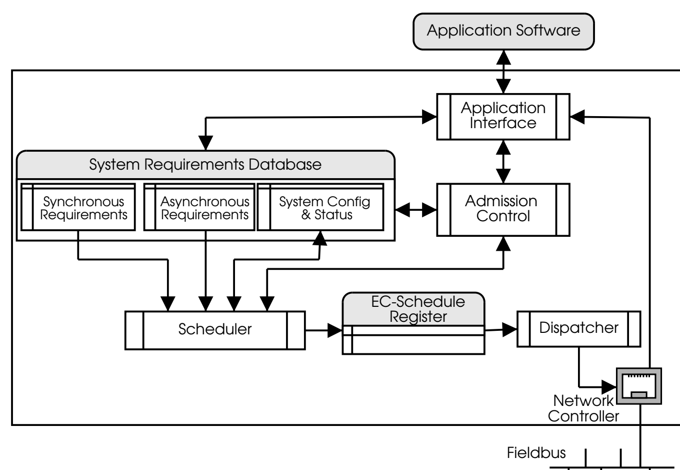
brought to you by CORE

provided by Repositório Institucional da Universidade de Aveiro

Paulo Bacelar
Reis Pedreiras

Suporte de Comunicações Tempo-Real Flexíveis em Sistemas Distribuidos

Supporting Flexible Real-Time Communication on Distributed Systems



Version 4

Paulo Bacelar Reis Pedreiras

Suporte de Comunicações Tempo-Real Flexíveis em Sistemas Distribuidos

Supporting Flexible Real-Time Communication on Distributed Systems

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

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

Júri

presidente

Prof. Doutora Maria Helena Vaz de Carvalho Nazaré
Reitora da Universidade de Aveiro

Prof. Doutor Paulo Jorge Esteves Veríssimo
Professor Catedrático da Faculdade de Ciências da Universidade de Lisboa

Prof. Doutor José Alberto Gouveia Fonseca
Professor Associado da Universidade de Aveiro

Prof. Doutor Luís Miguel Pinho de Almeida
Professor Auxiliar da Universidade de Aveiro

Prof. Doutor Eduardo Manuel de Médicis Tovar
Professor Coordenador do Instituto Superior de Engenharia do Porto

Prof. Doutor Josep Maria Fuertes Armengol
Professor Catedrático da Universidade de Catalunya, Espanha

Prof. Doutor Giorgio Buttazzo
Professor Associado da Universidade de Pavia, Itália

Agradecimentos

O trabalho realizado no âmbito da preparação desta dissertação envolveu directa e indirectamente diversas pessoas. A todas elas expresso o meu profundo e sincero agradecimento. Todavia, devido ao seu especial envolvimento, gostaria de agradecer em particular:

a Luís Miguel Pinho de Almeida, Professor na Universidade de Aveiro, por ter desempenhado exemplarmente o seu papel, primeiro como co-orientador e posteriormente como orientador científico principal. Para além de ter excedido largamente tudo o que é exigível à função de orientador, quer em termos técnicos quer científicos, conseguindo reunir todas as condições para que os trabalhos pudessem decorrer da melhor forma, não posso também deixar de realçar o humanismo que pauta todas as suas acções, o que propicia um ambiente onde trabalho e satisfação pessoal se conjugam harmoniosamente. Pela grande amizade demonstrada, pelo apoio incondicional prestado nos momentos difíceis e pelo modo exemplar como exerceu a sua função de orientação, o meu sincero e profundo agradecimento.

a José Alberto Gouveia Fonseca, Professor na Universidade de Aveiro, em primeiro lugar por me ter lançado este desafio, e em segundo lugar pelo modo irrepreensível com que desempenhou o seu papel, quer como orientador científico principal na fase inicial dos trabalhos, quer mais tarde como co-orientador. Para além da sua inestimável contribuição em termos técnicos e científicos para a realização deste trabalho, o seu empenho foi também decisivo na criação das condições necessárias à sua realização. Não posso deixar também de realçar o seu carácter profundamente humanista, o que faz com que trabalhar com ele seja sempre fonte de enorme satisfação pessoal. Pela grande amizade e confiança demonstrada de há longos anos a esta parte, pelo incondicional apoio prestado em todos os momentos e pelo modo exemplar como exerceu a sua função de orientação, o meu sincero e profundo agradecimento.

a Giorgio Buttazzo, Professor na Universidade de Pavia, Itália, por ter acompanhado os trabalhos realizados no âmbito desta tese desde o seu início. Para além da sua relevante contribuição em termos científicos, também me proporcionou um estágio no Laboratório de Sistemas de Tempo-Real (Retis Lab) da *Scuola Superiore S. Anna*, Pisa, Itália, que me permitiu não só realizar avanços significativos numa fase importante da implementação de um dos protocolos criados no âmbito desta tese, como também adquirir valiosos conhecimentos na área de sistemas operativos de tempo-real.

ao grupo de Sistemas Electrónicos Distribuídos, no seio do qual fui acolhido e procedi ao desenvolvimento deste trabalho. Em particular agradeço a Pedro Fonseca, Alexandre Mota e Ernesto Martins não só pelas profícuas discussões técnicas e científicas, que inequivocamente enriqueceram este trabalho, como também pela amizade e consideração demonstradas ao longo deste período. Gostaria ainda de agradecer a Joaquim Ferreira, com quem colaborei frequentemente na realização de trabalhos efectuados no âmbito desta tese. Para além da mútua amizade, as discussões técnicas e científicas também marcaram indelevelmente esta tese.

ao Laboratório de Sistemas de Tempo-Real (Retis Lab) da *Scuola Superiore S. Anna*, Pisa, Itália, que me acolheu para a realização de formação complementar em sistemas operativos tempo-real, que permitiu realizar avanços significativos relativamente à implementação de um dos protocolos criados no âmbito desta tese. Em particular agradeço a Paolo Gai e a Giuseppe Lipari, pelo empenho que tiveram em proporcionar-me não só todas as condições técnicas e logísticas para a realização da formação, mas também pela grande amizade e companheirismo que demonstraram, o que tornou simultaneamente agradável e proveitosa a estada em Pisa.

a todos os familiares, colegas e amigos pela amizade, apoio e encorajamento que sempre me dedicaram, o que contribuiu decisivamente para ultrapassar rapidamente os inevitáveis maus momentos. Sem eles este percurso teria sido certamente bem mais árduo. Em particular agradeço à Cristina e à Sofia pela (quase) infinita paciência que tiveram para comigo, e pelos sacrifícios, a vários níveis, a que as obriguei ao longo desta jornada.

Resumo

Os sistemas distribuídos controlados por computador (*Distributed Computer-Control Systems* / DCCS) encontram-se largamente disseminados, cobrindo aplicações que vão desde automação e controlo de processos industriais à aviónica, robótica e controlo automóvel. Muitas destas aplicações incluem actividades com características de tempo-real, i.e., actividades que têm de ser executadas durante janelas temporais bem definidas. Pela sua natureza distribuída, estes sistemas compreendem múltiplas unidades de processamento as quais, apesar de autónomas, necessitam de comunicar entre si para assegurar o controlo global do sistema. Assim, a troca de dados entre nodos encontra-se também sujeita a restrições temporais, donde o sistema de comunicação tem de garantir que esta ocorre de acordo com as restrições temporais requeridas pela aplicação.

Muitas aplicações de DCCS são complexas e heterogéneas, incluindo diferentes conjuntos de actividades, as quais exibem diferentes propriedades e requisitos. Por exemplo, encontram-se frequentemente actividades periódicas, resultando por exemplo de controladores operando em malha fechada, e actividades esporádicas resultantes de eventos que ocorrem em instantes imprevisíveis no ambiente a controlar. Todavia, a importância e tipos de requisitos temporais destas actividades são independentes da natureza da sua activação. Por outro lado, em sistemas DCCS a flexibilidade tem vindo a crescer de importância, em resultado quer da necessidade de reduzir custos de instalação, configuração e manutenção, quer do uso deste tipo de sistemas em aplicações emergentes, como manufactura ágil (*flexible manufacturing*), bases de dados de tempo-real com número variável de clientes, robótica móvel em ambientes não estruturados e controlo automático de tráfego, que têm de lidar com ambientes que são inerentemente dinâmicos.

Aplicações exibindo este grau de complexidade e dinamismo requerem sistemas suportando serviços activados quer pela passagem do tempo (*time-triggered)* quer por eventos (*event-triggered*) com garantias temporais e ao mesmo tempo exibindo flexibilidade operacional, suportando alterações dinâmicas às características das actividades que compreendem.

No que respeita especificamente ao sistema de comunicação, os protocolos existentes genericamente não preenchem estes requisitos. Em sistemas eminentemente *time-triggered*, os serviços *event-triggered* não existem ou são implementados de uma forma ineficiente, enquanto em sistemas eminentemente *event-triggered* algumas das propriedades mais interessantes exibidas pelos sistemas *time-triggered* são perdidas. Por outro lado flexibilidade e garantias temporais têm sido consideradas como propriedades conflituosas; sistemas que providenciam serviços com garantias temporais frequentemente requerem a especificação estática dos requisitos de comunicação, enquanto sistemas que suportam alterações dinâmicas aos requisitos de comunicação usualmente não fornecem garantias temporais.

O paradigma de comunicação apresentado nesta tese, denominado *Flexible Time-Triggered communication* (FTT), concentra os requisitos de comunicação e o escalonamento de tráfego num único nodo e utiliza uma técnica para distribuição do escalonamento denominada *master/multi-slave*. Esta caracteriza-se por consumir pouca largura de banda e por ser independente do algoritmo de escalonamento utilizado. Esta arquitectura facilita não só a implementação de escalonamento *on-line*, suportando portanto alterações aos requisitos de comunicação durante o funcionamento do sistema, como também a implementação *on-line* de controlo de admissão, o que permite rejeitar alterações que comprometam as garantias temporais do sistema, assegurando assim um comportamento previsível.

Em alguns domínios específicos de aplicação de DCCS, verifica-se uma necessidade crescente de suporte a gestão *on-line* de Qualidade de Serviço (*Quality of Service* / QoS). Genericamente, esta funcionalidade permite aumentar a eficiência da exploração dos recursos do sistema, pois habitualmente verifica-se uma relação directa entre o grau de recursos alocados às actividades de um sistema e o respectivo QoS. A gestão dinâmica de QoS requer um alto grau de flexibilidade, donde esta tese também descreve como o paradigma FTT suporta este tipo de serviço no que concerne ao tráfego.

Esta tese apresenta o paradigma FTT e defende que este permite combinar no mesmo sistema de comunicação diferentes tipos de tráfego, com a possibilidade de alterar as suas propriedades, executar gestão de QoS e alterar a politica de escalonamento durante o funcionamento, sem comprometer as garantias temporais granjeadas ao tráfego e atingindo uma elevada eficiência no uso da largura de banda.

O paradigma FTT apresentado nesta tese teve a sua génese no protocolo FTT-CAN. Após algum trabalho realizado sobre este protocolo verificou-se que os conceitos principais poderiam ser abstraídos, resultando um paradigma de comunicação genérico, passível de implementação em diversos meios de comunicação. Para verificar a performance do paradigma FTT, esta dissertação inclui algumas contribuições relativas ao protocolo FTT-CAN, nomeadamente no que concerne ao estudo do desempenho em termos de escalonamento e análise de tempos de resposta. Por outro lado é também apresentada a implementação do paradigma FTT sobre Ethernet (FTT-Ethernet), a qual se destina a aplicações mais exigentes no que respeita a poder de processamento e largura de banda, por exemplo aplicações integrando tráfego multimédia. No que respeita a este último protocolo explora-se essencialmente assuntos como a gestão dinâmica de QoS.

Abstract Distributed computer-control systems (DCCS) are widely disseminated, appearing in applications ranging from automated process and manufacturing control to automotive, avionics and robotics. Many of these applications comprise real-time activities, that is, activities that must be performed within strict time bounds. Due to its distributed nature, these systems comprise multiple autonomous processing units that, despite being autonomous, need to exchange data in order to achieve control over the environment. For this reason the data exchange among different nodes is also subject to real-time constraints, and thus the communication subsystem must be able to deliver data within specific time bounds. Many DCCS applications are complex and heterogeneous, comprising different sets of activities with different properties and requirements. For instance, they commonly include periodic activities, e.g. resulting from closed loop control, and sporadic activities resulting from events that occur at unpredictable instants in time in the environment under control. These types of activities can have distinct levels of criticalness and timeliness requirements, independently of their activation nature. On the other hand, flexibility is becoming increasingly important in DCCS, due both to the need of reducing the costs of set-up, configuration changes and maintenance, and also to the recent use of DCCS in new types of applications, such as agile manufacturing, real-time databases with variable number of clients, automotive, mobile robotics in unstructured environments and automatic traffic control systems, that must deal with environments that are inherently dynamic.

To cope with such high degree of complexity and dynamism, distributed real-time systems must support both time and event-triggered communication services under timing constraints and, at the same time, they must be operationally flexible, supporting on-the-fly changes to the computational activities they execute. Concerning specifically the communication subsystem, existing real-time protocols do not generally fulfill these requirements. In systems eminently time-triggered, event-triggered services are either non-existing or handled inefficiently, while in systems eminently event-triggered, interesting properties of time-triggered services are normally lost. On the other hand, flexibility and timeliness are often considered as conflicting: systems that provide timeliness guarantees are based on a static configuration of the communication activities while systems that support dynamic changes to the communication activities do not provide timeliness guarantees.

The communication paradigm herein presented, the Flexible Time-Triggered communication (FTT) paradigm, centralizes the communication requirements and scheduling of synchronous traffic in a single node and uses a master/multi-slave schedule distribution technique that requires low overhead and is independent of the particular scheduling algorithm employed. This architecture facilitates the implementation of on-line scheduling, which supports dynamic changes to the message set properties, and the implementation of on-line admission control, which permits to ensure that changes to the message set are only accepted if the timeliness requirements are all met.

In some application domains DCCS are also facing a trend towards higher flexibility in order to support on-line Quality-of-Service (QoS) management. This feature is generally useful to increase the efficiency in the utilization of system resources since typically there is a direct relationship between resource utilization and delivered QoS. On-line QoS management requires a high level of flexibility, and thus this dissertation also describes how the FTT communication paradigm can support such type of services.

This dissertation presents the FTT paradigm and argues that this paradigm allows to combine in the same communication system different types of traffic, with the ability to change their properties and the respective scheduling policy at run-time, without relinquishing predictability guarantees and achieving efficient use of network bandwidth.

The FTT paradigm presented in this thesis has its roots in the FTT-CAN protocol. After some work performed over the FTT-CAN protocol, it was realized that the main concepts could be abstracted and used to build a generic communication paradigm, which could be implemented in distinct communication networks. To assess the performance of the FTT paradigm, this dissertation includes some contributions to the FTT-CAN protocol, mainly in what concerns scheduling and response-time analysis. Moreover, it also presents an implementation over Ethernet (FTT-Ethernet), which aims at more resource demanding applications, supporting for instance multimedia activities. For this reason, in the scope of the FTT-Ethernet protocol most of the work presented is related to on-line QoS management.

Apoios

Este trabalho foi apoiado pelas seguintes instituições:

Ministério da Ciência e do Ensino Superior, por meio da Fundação para a Ciência e a Tecnologia, que me concedeu uma bolsa de Doutoramento no âmbito do III Quadro Comunitário de Apoio, programa POSI - Desenvolver Competências - Medida 1.2 (PRAXIS XXI / BD / 21679 / 99), o que possibilitou a realização dos trabalhos em regime de dedicação exclusiva.

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

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

Dedicatória Dedico este trabalho em particular:

à Cristina e à Sofia,
à memória de minha mãe e a meu Pai,
a todos os familiares e amigos.

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Flexible real-time distributed systems 2

1.3 Central proposition and contributions 3

1.3.1 Improvements on the FTT-CAN protocol: 4

1.3.2 Specification of the FTT paradigm 4

1.3.3 The FTT-Ethernet protocol 5

1.4 Organization of the dissertation 6

2 Real-time systems fundamentals **9**

2.1 Basic concepts on real-time systems 9

2.2 Scheduling real-time systems tasks 11

2.3 Schedulability analysis 13

2.4 Examples of scheduling algorithms 14

2.4.1 Task model 14

2.4.2 On-line scheduling algorithms 16

2.4.3 Schedulability tests 18

2.5 Conclusion 24

3 Distributed real-time systems **27**

3.1 Real-time communication 28

3.1.1 Event and Time-triggered communication paradigms . 29

3.1.2 Combining event and time-triggered traffic 31

3.1.3 Message Scheduling 31

3.1.4 Co-operation models 34

3.2 Fieldbus Protocols - brief survey 36

3.2.1 The Controller Area Network (CAN) protocol 37

3.2.2 WorldFIP 39
3.2.3 Profibus 42
3.2.4 P-Net 44
3.2.5 DeviceNet 45
3.2.6 TT-CAN 47
3.2.7 TTP/C 48
3.2.8 FF-H1 50
3.2.9 FlexRay 53
3.2.10 Fieldbus properties summary 55
3.3 Ethernet-based RT protocols - brief survey 56
3.3.1 The Ethernet protocol 58
3.3.2 Modification of the Medium Access Control 60
3.3.3 Addition of transmission control over Ethernet 61
3.3.4 Ethernet-based protocols properties summary 66
3.4 Conclusion 67

4 The FTT paradigm 69
4.1 Why a new protocol 70
4.2 The Flexible Time-Triggered paradigm 72
4.2.1 System architecture 73
4.2.2 The Elementary Cycle 75
4.2.3 Master node architecture 77
4.2.4 Station node architecture 81
4.3 Synchronous Traffic Analysis 87
4.3.1 Synchronous Message Model 88
4.3.2 Utilization-based schedulability analysis 90
4.3.3 A necessary and sufficient schedulability test 93
4.4 Asynchronous traffic analysis 95
4.4.1 Worst-case response time for AT1 asynchronous message class 96
4.4.2 Worst-case response time for AT2 asynchronous message class100
4.5 Conclusion 103

5 QoS management based on FTT 105
5.1 Adding a QoS manager 106
5.2 Examples of QoS management policies 108
5.2.1 Priority-based QoS management 108
5.2.2 Elastic Task Model based QoS management 109

5.2.3 Applying the Elastic Task Model to message scheduling 110
5.3 QoS management case study: a mobile robot 112
5.3.1 Communication requirements 112
5.3.2 Using the priority-based QoS manager 115
5.3.3 Using the Elastic Task Model QoS manager 116
5.4 Conclusion 117

6 Contributions to FTT-CAN 119
6.1 The FTT-CAN Elementary Cycle 119
6.1.1 Message Arbitration 120
6.1.2 Enforcing temporal isolation 121
6.1.3 FTT-CAN message types 122
6.2 Synchronous traffic 126
6.2.1 Schedulability analysis 126
6.2.2 Experimental results 127
6.3 Asynchronous traffic 134
6.3.1 Schedulability analysis 134
6.3.2 Experimental results 135
6.4 Using a Planning Scheduler 138
6.4.1 Responsiveness limits 139
6.4.2 Improving the responsiveness 141
6.4.3 Implementation issues 143
6.4.4 Performance analysis 144
6.5 Dependability issues 148
6.5.1 FTT-CAN Master replication 148
6.5.2 Master replica synchronization protocol 149
6.5.3 Computing the worst-case synchronization time 150
6.5.4 Active master replacement 152
6.5.5 Experimental results 153
6.6 Conclusion 154

7 The FTT-Ethernet protocol 157
7.1 The FTT-Ethernet Elementary Cycle 159
7.1.1 Message Arbitration 160
7.1.2 Enforcing temporal isolation 161
7.1.3 FTT-Ethernet message types 162
7.2 Schedulability analysis 166

7.2.1 Message's transmission time computation 166
7.2.2 Synchronous traffic 168
7.2.3 Asynchronous traffic 170
7.3 FTT-Ethernet implementation 172
7.3.1 S.Ha.R.K. brief overview 172
7.3.2 Implementing FTT-Ethernet on top of Shark 173
7.4 Experimental results 175
7.4.1 Experiment characterization 176
7.4.2 Results with FTT-Ethernet 178
7.4.3 Results with hub-based Ethernet 179
7.4.4 Results with switched Ethernet 180
7.4.5 Experimental results analysis 180
7.5 Conclusion 181

8 Conclusions and future work 183
8.1 Contributions 183
8.2 Future research 188

A List of publications and communications 205
A.1 Journal articles 205
A.2 Conference papers 206

B List of acronyms 209

C FTT-Ethernet sample application 213

List of Figures

2.1 Generic computer-based control system block diagram 9
2.2 Taxonomy of real-time scheduling algorithms 12
2.3 Exact, sufficient and necessary schedulability tests 14
2.4 Schedule generated by RM 17
2.5 Schedule generated by EDF 18

3.1 Layered communication architecture 33
3.2 CAN 2.0A message frame 38
3.3 Periodic message properties and resulting BAT 41
3.4 Profibus token-passing and master-slave relations 43
3.5 TT-CAN system matrix 48
3.6 TTP/C architecture 49
3.7 Foundation Fieldbus link 51
3.8 FlexRay communication cycle structure 54
3.9 Ethernet frame 59

4.1 The FTT paradigm system architecture 73
4.2 The Elementary Cycle structure 75
4.3 FTT master internal architecture 77
4.4 FTT station internal architecture 81
4.5 FTT station network software architecture 85
4.6 Expanding the synchronous window to allow using the blocking-free non-preemptive model 90
4.7 Modeling the effect of the inserted idle-time, asynchronous window and trigger message 92
4.8 Maximum dead-interval (σ_i) and level-i busy window (w_i) . . 97
4.9 Calculating the level-i busy window 101

5.1 Adding QoS management to FTT 108
5.2 Rounding of periods in FTT-Ethernet. 111

5.3 Increasing the effective utilization factor in FTT-Ethernet. . . 111
5.4 Robot components 113

6.1 FTT-CAN Elementary Cycle 120
6.2 Preventing synchronous window overrun 121
6.3 Experimental set-up 128
6.4 Schedulability versus bus utilization under RM and EDF . . . 130
6.5 Percentage of schedulable message set using EDF scheduling on CAN133
6.6 SMS Responsiveness bounds 140
6.7 Using the AMS to temporarily convey a new synchronous message141
6.8 Operational flowchart 143
6.9 Transition from SSP to SMS 145
6.10 Timeline of the scheduling synchronization process 150
6.11 Master replacement process 153

7.1 Layer model of factory communications 158
7.2 FTT-Ethernet Elementary Cycle 160
7.3 Asynchronous message arbitration scheme 161
7.4 Preventing window overrun 162
7.5 FTT-Ethernet frame 163
7.6 Ethernet propagation delay 168
7.7 Unwanted collision between synchronous messages 169
7.8 Including the propagation delays in the schedule 170
7.9 Asynchronous arbitration overhead 171
7.10 Master node: time-critical activities 174
7.11 Slave node: time-critical activities 175
7.12 Packets sent using FTT-Ethernet. 178

List of Tables

2.1 Periodic task set properties 17

5.1 Message set and properties 114
5.2 Message set network utilization 114
5.3 Message set utilization: priority-based QoS manager 116
5.4 Message set network utilization: ETM QoS manager 117

6.1 Message type identification 123
6.2 EC Trigger Message structure 123
6.3 Communication overhead imposed by the EC Trigger Message 124
6.4 Synchronous Data Message structure 124
6.5 Asynchronous Data Message structure 125
6.6 Control Message structure 126
6.7 Synchronous communication requirements 136
6.8 Asynchronous communication requirements 136
6.9 Results from experiment 1 137
6.10 Results from experiment 2 137
6.11 Synchronous message properties. 154

7.1 EC Trigger Message structure 163
7.2 Synchronous Data Message structure 164
7.3 Asynchronous Data Message structure 165
7.4 Control Message structure 166
7.5 Communication overhead imposed by the EC Trigger Message 167
7.6 Task set parameters used in the experiments. (Periods and transmission times in milliseconds)177
7.7 Periods of each message (ms) during the experiments. 177
7.8 Message jitter with FTT-Ethernet. 179
7.9 Message jitter (shared Ethernet). 180

7.10 Message jitter (switched Ethernet). 180

Chapter 1

Introduction

1.1 Overview

In the last decades distributed computer control systems (DCCS) became widely disseminated, appearing in many application fields such as automated process and manufacturing control, automotive systems, avionics and robotics [Pim90, LA99, Kop97]. Many of these applications pose stringent constraints to the properties of the underlying control system, which arise from the need to provide predictable behavior during extended time periods. Depending on the particular type of application, failure to meet these constraints can cause important economic losses or even put human lifes in risk [Kop97].

To cope with these requirements, early DCCSs have been developed based on static off-line scheduling, i.e., all activities are modeled and analyzed during system design, based on a complete a priori knowledge about the system properties (e.g. [Kop99]). The resulting static schedule is used during system run-time to coordinate all system activities. This framework provides a high level of predictability, since all activities and respective activation instants are known beforehand, and so a correct system will perform as planned in all anticipated circumstances. For this motive, many safety critical applications employ static off-line scheduling.

Frequently, complete knowledge about the system is hard or even impossible to gather at design time [SLST99]. In this case, the use of static off-line scheduling of activities would be impossible at all, or, even when possible, would result in poor resource efficiency, because it would require the use of

an extended range of conservative approaches. Thus, to be able to deploy such kind of application in a more effective way, system activities should be dynamically scheduled during run-time, as they are required. In this case it is also possible to provide a priori guarantees about the system predictability, however the amount of information required is lower than in the case of static off-line scheduling.

1.2 Flexible real-time distributed systems

Many real-world systems are complex and dynamic, evolving during time and consequently changing their requirements that nevertheless must be always fulfilled by the control system. Furthermore, the adoption of DCCSs in markets such as the automotive, in which economic issues are of paramount importance, requires highly efficient systems. To cope with the requirements of such applications, DCCS systems must be able to adapt themselves to the evolving requirements of the environment they are attached to. However, high resource efficiency frequently conflicts with static scheduling approaches, according to which resources are permanently allocated based on worst-case requirements.

An initial step to improve efficiency consists in the provision of several modes of operation during system design. At run-time, the particular mode of operation that better fits the operational requirements is selected. Issues concerning the timeliness during mode changes have been addressed in previous scientific work [Ped99, Foh93]. Some communication protocols support the mode changes semantic to provide some level of flexibility (e.g. Time-Triggered Protocol (TTP) [KG94]). Nevertheless, mode changes are still restrictive, since all the modes are required to be completely known and characterized during system design. For complex highly dynamic systems, this degree of knowledge can be unavailable, or can result in an explosion on the number of possible modes, making their implementation cumbersome or even impossible at all.

To be able to support applications having such high complexity and high degree of dynamism, a distributed real-time system must be operationally flexible, meaning that it must support on-the-fly changes to the computational activities carried on. In distributed systems, computation activities imply the execution of tasks, eventually residing in distinct nodes, as well as

data exchanges between them using an appropriate communication network. Both task execution and data exchange activities are closely related. In a distributed environment tasks require as input and/or produce as output data, which must be distributed by the underlying communication network within constrained time boundaries [TC94, GH98]. Failing to meet such time constraints can result in feeding tasks with outdated data, which in its turn can compromise the entire system behavior. From this strong interdependency between tasks and communication activities within distributed systems, it follows that changes in the properties of real time activities can lead to changes both in the task and message scheduling.

Another requirement found in real-time distributed systems is the capacity to deliver both time and event-triggered communication services under timing constraints [LA99]. In time-triggered systems the communication activities are triggered at pre-defined time instants, according to a global schedule, thus requiring a global time synchronization. This approach allows setting the different message streams out of phase, which in some cases may result in a reduction in the number of message streams that become ready for transmission simultaneously. Therefore, this type of systems is well suited to convey periodic updates of state data. On the other hand, in event-triggered systems communication activities occur only when required, thus these systems are more adapted to convey alarms and management data. Most DCCSs privilege either one or the other type of services. In systems eminently time-triggered, event-triggered services are either non-existing or handled inefficiently in terms of either response time or network utilization. On the other hand, in systems eminently event-triggered, interesting properties of time-triggered services such as global synchronization and composability with respect to the temporal behavior are normally lost. Thus, another aspect that should be addressed by a flexible system is the efficient integration of both these traffic paradigms, with mechanisms providing temporal isolation between them, in order to prevent mutual interference.

1.3 Central proposition and contributions

This work introduces a communication paradigm deemed to support the requirements of flexible distributed real-time systems. It is our thesis that the proposed communication paradigm allows combining in the same commu-

nication system different types of traffic, with the ability to change traffic properties and/or the respective scheduling policy during system run-time, without relinquishing predictability guarantees and achieving efficient use of network bandwidth. More specifically, the envisaged traffic types are time and event-triggered with distinct timeliness requirements (hard/soft/non-real-time). The proposed communication paradigm meets the following objectives:

- Support for on-line message scheduling of time-triggered messages based on dynamic requirements;
- Support for on-line changes between different scheduling policies, both with fixed and dynamic priorities, concerning the time-triggered traffic;
- Timeliness guarantees concerning the real-time traffic, based on on-line admission control;
- Support for distinct traffic types (time and event-triggered) with temporal isolation;
- Low protocol overhead;

The contributions found in this thesis relate to the specification, analysis and implementation of such communication paradigm, and are the following:

1.3.1 Improvements on the FTT-CAN protocol:

The FTT-CAN protocol was developed at the University of Aveiro ([AFF98]) and relies on the Controller Area Network (CAN) [Rob91] as the base communication network protocol. The initial implementation of the FTT-CAN protocol comprised a planning scheduler and an on-line admission control protocol based on a schedulability analysis for the periodic traffic assuming fixed priorities. The research made in the scope of this thesis addresses on one hand the scheduling of periodic messages using dynamic priorities and respective feasibility analysis, and on the other hand the support for aperiodic traffic, both real and non-real-time, and respective timeliness analysis.

1.3.2 Specification of the FTT paradigm

Based on the set of requirements resulting from the main proposition of this thesis, the major contribution consists on the definition of a framework

able to support the communication requirements of flexible distributed real-time systems. This framework is designated Flexible Time-Triggered (FTT) paradigm and defines a communication system architecture. The system architecture herein referred to is generic in the sense that it does not rely on any particular network protocol. The only requirement posed by the FTT paradigm with respect to the underline communication protocol is the ability to exchange broadcast messages. The FTT paradigm defines a centralized scheduling architecture, where a particular node, designated by Master, is responsible for managing a database with all the relevant communication requirements, performs on-line feasibility tests concerning the real-time traffic, executes a dynamic scheduler and finally distributes the generated schedules to the network devices. From the device side, the FTT paradigm also defines the rules to perform communications. Furthermore, all these functions are abstracted from the respective implementation, thus allowing applications to be developed independently of the particular implementation and MAC. To support such architecture, suitable scheduling and on-line admission protocols were also developed.

1.3.3 The FTT-Ethernet protocol

One important aspect of flexibility is related to scalability. Distributed real-time systems are used in a wide range of applications, with different requirements in many aspects, namely bandwidth. Observing that some applications require greater bandwidth than the one made available by traditional fieldbus protocols like CAN, the FTT paradigm was also implemented over Ethernet, leading to the FTT-Ethernet protocol. With respect to this protocol, besides the implementation of the functions strictly related with the FTT paradigm, a further research was developed in the field of dynamic Quality of Service (QoS) handling and support for multimedia message streams. Concerning the dynamic QoS management, an implementation of the Elastic Task Model [BLA98] was performed, providing support for message streams characterized by ranges of acceptable QoS concerning the network utilization, as well as a method to assign dynamically the best possible QoS to each such message, according to the available network resources.

1.4 Organization of the dissertation

In this chapter we have outlined the scope of the thesis and briefly discussed the need for further research on the flexibility of the communication networks supporting distributed real-time systems. Finally, it was presented the central proposition of this thesis and its main contributions. The reminder of this thesis provides background information on this research field and presents the work done in order to support the proposition made above, being organized as follows:

Chapter 2 includes a brief overview of the area of real-time systems, with special emphasis on the issues that are addressed in this dissertation. Starting with an informal presentation of the main concepts on real-time systems, the focus then moves to an overview of the most relevant results in the field of scheduling algorithms and schedulability analysis.

Chapter 3 is devoted to distributed real-time systems. This chapter starts by a characterization of distributed real-time systems, task activation and co-operation models and message scheduling. Then it presents an overview of some of the more relevant communication protocols used in DCCS systems. Besides the dedicated communication protocols, developed specifically for use in DCCSs, are also addressed real-time protocols based on Ethernet, which recently has been target of interest both from the scientific and industrial communities. This chapter includes two tables that summarize the properties of these protocols in issues ranging from the support of different types of traffic to timeliness guarantees and operational flexibility.

Chapter 4 presents the Flexible Time-Triggered communication paradigm. This chapter is the heart of this dissertation and starts by presenting a set of requirements that flexible real-time communication networks must fulfill, as well as the justification for the proposal of a new paradigm. Then the FTT paradigm is presented in detail, both from an architectural and functional point of view. Furthermore, this chapter also presents a generic schedulability analysis, both concerning the synchronous and asynchronous traffic, adapted to cope with the FTT constraints. Although generic, the analysis herein presented must be slightly adapted to handle the peculiarities of the underline commu-

nication network, issue that is addressed in Chapters 6 and 7, for the FTT-CAN and FTT-Ethernet implementations, respectively.

Although chronologically the FTT paradigm as appeared after the FTT-CAN protocol, the presentation becomes more clear and understandable if the paradigm is presented before the implementations. For this reason the FTT paradigm is presented in Chapter 4, while the FTT-CAN and FTT-Ethernet implementations are presented in Chapters 6 and 7, respectively.

Chapter 5 discusses the suitability of the FTT paradigm to support systems that benefit or even require dynamic QoS management. This chapter starts by discussing the internal implications of supporting this type of service. Then two illustrative QoS management policies are presented, which are used in a simple case study.

Chapter 6 and 7 present two FTT implementations, one based on the Controller Area Network protocol (Chapter 6), and another based on Ethernet (Chapter 7). Although from the application point-of-view the set of services provided by any of the implementations is basically the same, their internals must cope with the particularities that each one of the underline communication protocols presents. Such particularities become specially visible in what concerns the message arbitration, access-control and arbitration techniques employed in each case, which are carefully discussed. Moreover, these chapters also include the small adaptations that must be performed in the generic schedulability analysis presented in Chapter 4.

Both of these chapters include simulation and experimental results that allow, in some extent, to assess the performance of the protocols.

Chapter 8 contains a brief summary and discussion about the contributions presented in this dissertation and suggests some lines of future research that seem promising.

Chapter 2

Real-time systems fundamentals

2.1 Basic concepts on real-time systems

Computer-based control systems are becoming a commonplace. They are often found in applications ranging from bread toasters, washing machines, automatic doors and access control systems to automotive, avionics, robotics and process and manufacturing industries. A computer-based control system comprises at least a sensory system to gather data about the state of the system under control, or environment, a computer able to execute a control algorithm and an actuation system.

The nature of the computations made in this kind of systems is very broad, ranging from complex numerical computations required to imple-

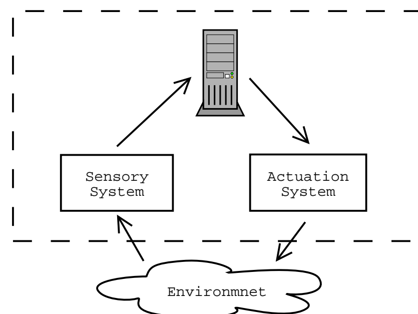


Figure 2.1: Generic computer-based control system block diagram

ment advanced control algorithms or image processing used for instance in robotics, to basic operations like turning some device on or off according to a binary input fed by some sensor. A broad range of values is also found concerning the time granularity. For example, in industrial environments it is usual to find control loops in the range of seconds to milliseconds.

Systems are considered to produce logically correct results when its outputs are related to the actual inputs according to the laws determined during system specification. However, for some systems, this requirement is not enough. For instance, if the bread toaster controller takes an excessive amount of time to turn it off after detecting that the bread is enough toasted, the output of the process can become a piece of charcoal. Such kind of systems, in which computations must be carried within specific time boundaries, are referred as having real-time requirements. More concisely, a real-time computer system is a computer system in which *the correctness of the system behavior depends not only on the value of the computation but also on the time at which the results are produced* [SR88]. Thus, a real-time system must react to changes in the state of the object under control within time boundaries, which depend on the dynamics of the controlled object. The last instant at which a result can be produced is called deadline.

Depending on the particular application, failing to meet deadlines can have dissimilar consequences. For example, to be able to reach some geographical position, a mobile robot must collect data from the environment and use it to perform trajectory planning. However, to be able to deal with real environments, it must also be able to detect and avoid obstacles. If due to some system overload, the trajectory planning task sometimes does not have enough computational resources to execute, the robot will take more time to reach its goal, but eventually will reach it, provided that the deadline miss ratio is not too high. On the other hand, if, in the course of the same overload, the robot fails in timely detecting the presence of an obstacle, it can collide with it. This failure can cause economical losses, for example if the robot or the object with which it collides becomes damaged, or it can also put human lifes in risk, for example if the undetected object is a person. In [Kop97] deadlines are classified as *firm* or *soft*. If a result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. Whenever failing to meet a firm deadline can lead to a catastrophe, the deadline is called *hard*. Whenever a computer system executes at

least one activity having an hard deadline it is called a *hard real-time system* or *safety-critical real-time system*. If no hard real-time deadlines exist, the system is called *soft real-time system*.

2.2 Scheduling real-time systems tasks

In the scope of real-time systems, processes (or logical units of concurrency within the system, interacting to achieve a common goal [Aud93]) in a real-time application are mapped on software tasks. Tasks thus represent activities handled by the computational system. Usually computational systems execute several activities, eventually with different deadline constraints. Some of these activities are independent of each other, with no precedence constraints or shared resources. Other activities must be executed in some specific order, or share access to some entities, such as data structures or I/O devices.

To be able to perform correctly, the resources required by all the activities should be granted in a way that they can be completely served within their respective deadlines, while respecting any other requirements, such as precedence constraints. The procedure of selecting which task should be executed at a particular point in time is called scheduling and the set of rules that, at any time, determines the order in which tasks are executed is called a scheduling algorithm. More accurately, a scheduling problem can be defined [But97] by three sets: a set of n tasks $J = \{J_1, J_2, ..., J_n\}$, a set of m processors $P = \{P_1, P_2, ..., P_m\}$ and a set of s types of resources $R = \{R_1, R_2, ..., R_s\}$. Furthermore, precedence relations among tasks can be specified through a directed acyclic graph and each task can have associated timing constraints. In this context scheduling means to assign processors from P and resources from R to tasks from J in order to complete all tasks under the imposed constraints.

Real-time scheduling is perhaps the research topic that deserved most attention from the real-time research community. A common taxonomy (e.g. [But97]) of real-time task scheduling is presented in Figure 2.2:

Off-line. All scheduling decisions are made prior to system execution. The resulting schedule is stored in a table, called dispatcher table, which contains the list of tasks and the respective activation instants. During run-time a cyclic executive, called dispatcher, sequentially and repeatedly scans

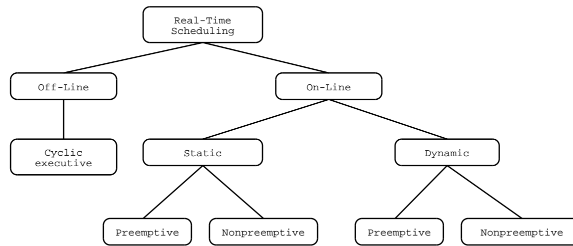


Figure 2.2: Taxonomy of real-time scheduling algorithms

the list and activates the tasks at the appropriate instants. To be able to use this approach, a complete characterization of the properties of the task set is required in advance. Therefore, this method cannot handle systems that require runtime changes to the task set. On the other hand, such systems require low runtime overhead and support complex scheduling algorithms. The former property results from the fact that, during runtime, the overhead is due only to the dispatcher execution, which in turn only needs to read data sequentially from a table. The latter feature results from the fact that the scheduling is performed prior to system execution. Thus, the time required to build the schedule is not tightly constrained. Moreover, the scheduling algorithm can be (and usually is) executed in a computational system other than the one used to deploy the system, which can have more adequate resources to perform this function.

On-line. Scheduler decisions are taken during system runtime, upon the occurrence of some event that requires rescheduling. Such events can be for instance the arrival of new tasks, a blocking, or the termination of the currently executing task. To select the next task to execute among the ready ones, a particular parameter, usually called priority, is used. The priority is derived by some specific methodology, resulting for instance from the temporal properties of the task or its relative importance. This approach supports runtime changes to the message set, since in each invocation the scheduler considers only the set of ready tasks. On the other hand, the runtime processing required to find a schedule can be substantial. Since the time required to build the schedule is overhead in what concerns the execution of application tasks, the complexity of the scheduling algorithms must be bounded.

Static. Scheduling decisions are based on fixed information that is avail-

able at pre-runtime, e.g. fixed priorities.

Dynamic. Scheduling decisions are based on information that is available at runtime, only, e.g. the release instants of aperiodic tasks.

Non-preemptive. A running task executes until it decides to release the allocated resources, usually on completion, irrespectively of other tasks becoming ready, eventually with higher priority. In this case scheduling decisions are only required after task's completion instants.

Preemptive. A running task can be suspended or interrupted during its execution, if at some instant a task with higher priority becomes ready. In non-preemptive systems, when a task becomes ready, it must wait at least for the completion of the running task, independently of their relative priorities. This effect is called blocking. Preemptive systems are more responsive concerning higher priority tasks, since these tasks do not suffer blocking from lower priority ones. However, in this case, scheduling events are generated more often, in all task activation instants, resulting in higher overhead when compared with non-preemptive systems.

2.3 Schedulability analysis

Hard real-time systems demand a high degree of predictability, thus the feasibility of the schedule should be guaranteed in advance. On the other hand, soft real-time systems have less stringent requirements, and missing deadlines have no catastrophic consequences. Scheduling algorithms fall into two classes, guarantee-oriented and best effort [SR92]. In off-line scheduled systems task properties such as activation instants, worst-case computation times, etc. are known a priori, and the schedule is built before runtime. Provided that the assumptions concerning the task properties are accurate, if a feasible schedule is found the tasks are guaranteed to meet their deadlines during system runtime. Thus, this kind of algorithms fall into the guaranteed-oriented class. However, in on-line scheduled systems, that knowledge might not be available, e.g. when tasks are created and removed dynamically during runtime. In this case, if there is an on-line admission control mechanism based on a schedulability test, responsible for rejecting changes to the task set that compromise the system timeliness, the scheduling algorithm also falls into the guarantee-oriented class. This scheduling paradigm is known as dynamic planning based [RS94], because the resources

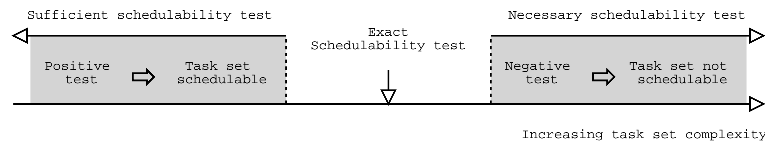


Figure 2.3: Exact, sufficient and necessary schedulability tests

of accepted tasks are reserved into the future. On the other hand, if changes to the task set are always accepted without any kind of assessment, it is not possible to guarantee the system timeliness, and thus such algorithms fall in the best effort category .

The schedulability test algorithms are closely related to the particular scheduling algorithm. The schedulability test result must reflect the ability of the particular scheduling algorithm to find or not a feasible schedule. In some cases, the schedulability test is exact, meaning that, if a feasible schedule can be built, the test result is positive, and conversely, a negative result implies that the scheduling algorithm is unable to find a feasible schedule. However, exact schedulability tests can be too complex to execute on-line, or even be computationally intractable [GJ75]. Sufficient schedulability test algorithms can be simpler. However, a sufficient schedulability test can reject feasible sets. On the other hand, sets rejected by a necessary schedulability test algorithm are not certainly schedulable, but tasks sets that are not rejected may be not schedulable. Figure 2.3 depicts the guarantees delivered by these types of schedulability tests.

2.4 Examples of scheduling algorithms

This section briefly presents some paradigmatic scheduling algorithms and respective schedulability analysis. Particular attention is devoted to Rate Monotonic and Earliest Deadline First scheduling algorithms because later on these algorithms will be re-used for message scheduling.

2.4.1 Task model

Tasks are activated in response to some event. For instance, in a computer controlled system, whenever a sensor detects a change in a particular environment variable, the task that implements the control algorithm can be

activated and executed when possible. In this case the activation instants of the tasks cannot be predicted. These tasks are called *aperiodic*. If there is a minimum inter-arrival time between any two consecutive activations, tasks are called *sporadic*. Some other tasks are required to be activated regularly. This situation is often found in computer control systems, to enforce the sampling of data at some desired rate. These tasks are known as *periodic*. To be able to schedule a set of tasks, scheduling algorithms need to have a minimum level of knowledge about each task properties. A set of periodic tasks Γ can be denoted by:

$$\Gamma = \{\tau_i(C_i, T_i, Ph_i, D_i, Pr_i), i = 1, ..., n\} \tag{2.1}$$

where:

- C_i is the worst case computation time required by task τ_i;
- T_i is the period of task τ_i;
- Ph_i, is the initial phase of task τ_i;
- D_i is the relative deadline of task τ_i;
- Pr_i is the priority or value of task τ_i.

The activation instant ($a_{i,k}$) and absolute deadline value ($d_{i,k}$) of the generic k^{th} instance of the periodic task τ_i can be computed as:

$$a_{i,k} = Ph_i + (k-1) * T_i$$
$$d_{i,k} = a_{i,k} + D_i$$

The same notation is valid for sporadic tasks, except that the period (T_i) becomes the minimum inter-arrival time (mit_i) and the initial phase is not defined. In this case the activation instant and absolute deadline instants can be computed as:

$$a_{i,k} \geq a_{i,k-1} + mit_i$$
$$d_{i,k} = a_{i,k} + D_i$$

2.4.2 On-line scheduling algorithms

The seminal work by Liu and Laylan [LL73] includes two of the most important scheduling algorithms for independent task scheduling in single CPU systems. These algorithms are the Rate Monotonic, for static priorities systems and Earliest Deadline First for dynamic priorities systems. The relevance of these algorithms results from the fact that they are optimal among their classes. An algorithm is optimal if it is able to generate a feasible schedule whenever some other algorithm of the same class is able to do it.

Rate Monotonic algorithm

The Rate Monotonic (RM) algorithm [LL73] is an on-line preemptive algorithm based on static priorities.

According to the RM algorithm, priorities are assigned monotonically with respect to the tasks period; the shorter the period, the greater the priority:

$$\forall \tau_i, \tau_j \in \Gamma \, : \, T_i < T_j \, \Rightarrow \, Pr_i > Pr_j \tag{2.2}$$

At runtime, whenever a task instance is activated or the running task finishes executing, the scheduler selects the task with highest period among the ready ones. The overall complexity of this algorithm is $O(n)$ since inserting a new task instance in an order queue of n elements may take up to n steps. At dispatching time, selecting the highest priority ready task just requires to get the first element of the head of the queue.

Earliest Deadline First Algorithm

The Earliest Deadline First (EDF) [LL73] algorithm is an on-line preemptive algorithm based on dynamic priorities. According to the EDF algorithm, the earliest the deadline the highest the priority of the task. During runtime the following relation holds:

$$\forall \tau_i, \tau_j \in \Gamma_R \, : \, d_i < d_j \, \Rightarrow \, Pr_i > Pr_j \tag{2.3}$$

where Γ_R is the subset of Γ comprising the ready tasks and (d_i ,d_j) are the absolute deadlines of tasks τ_i and τ_j.

Task	T	C
1	4	2
2	6	2
3	11	1

Table 2.1: Periodic task set properties

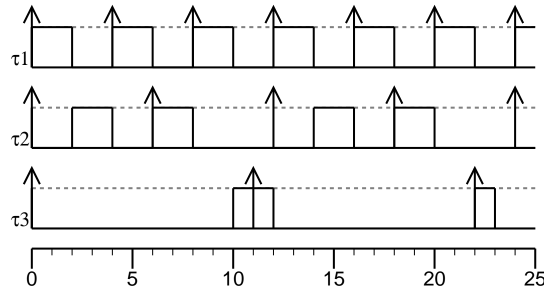


Figure 2.4: Schedule generated by RM

At runtime, whenever a task instance is activated or the running task finishes executing, the scheduler selects the task with highest period among the ready ones. Since the task priorities are dynamic, it is necessary to sort the ready task queue whenever new task instances are activated. Thus, the time complexity of this algorithm is $O(n * log(n))$. If follows that EDF scheduling requires higher runtime overhead than the RM scheduling algorithm, which can be problematic in systems based on low processing power CPUs, often found in some embedded distributed control applications. However, as it will be seen further on, compared to RM, the EDF algorithm is able to achieve higher utilization factors and, at the same time, the number of preemptions can be potentially lower. This results in a trade-off between runtime overhead and schedulability level, which must be evaluated case by case. Figures 2.4 and 2.5 depict the timeliness relative to the schedules generated both by an RM and EDF schedule algorithms for a periodic task set with the properties stated in table 2.1.

In Figure 2.4, concerning the RM scheduler, it can be observed that task τ_1 always executes first, since it has the shortest period among all tasks, and thus the highest priority. Task τ_2 always executes before task τ_3 because it has a shorter period. However, in Figure 2.5, concerning the EDF scheduler, the priority depends on the distance to the deadline, and thus it changes

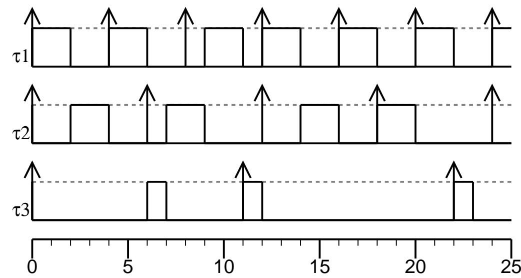


Figure 2.5: Schedule generated by EDF

during runtime. For instance, at time t=6 task τ_3 has the shortest deadline and thus executes before task τ_2.

Other scheduling algorithms

Many other scheduling have been developed along the years. Two other well-known algorithms are the Deadline Monotonic (DM) [LW82] and the Least-Laxity (LL) algorithms [MD78]. The DM algorithm belongs to the class of the static priorities preemptive algorithms and uses the same assumptions as the RM algorithm except that relative deadlines can be shorter than the periods. In this algorithm task priorities are assigned according to the task relative deadlines instead of periods. The DM algorithm is also optimal in its class [LW82]. The LL algorithm makes the same assumptions as the EDF algorithm. However, the priority assignment is made according to the laxity of the task, i.e., the amount of time that a task can wait to be able to finish within the deadline. The LL algorithm also is optimal in its class [MD78].

2.4.3 Schedulability tests

Most of the schedulability tests fall in one of two classes: utilization-based and response-time based. The former ones have a lower computational complexity than the latter ones, thus from this point of view are more suitable to be used in on-line scheduled systems. However, response-time based schedulability tests are usually less pessimistic and can provide individual response-time bounds for each task.

Utilization-based schedulability tests

Liu and Layland present a sufficient schedulability condition for the RM algorithm [LL73]. The following assumptions are assumed:

- Task set only comprises periodic tasks;
- Relative deadlines of all tasks are equal to the tasks periods;
- Independent tasks, i.e., no precedence or mutual exclusion constraints;
- All task instances have the same worst-case execution time.

Moreover, it is implicitly assumed that, once started, task instances execute until completion or preemption and that the operating system overhead (e.g. time required for context switching and tick handling) is small and can be ignored. However, when required, the operating system overhead can be accounted for in the analysis.

The processor utilization factor of a task set is defined as the fraction of the processor time spent in the execution of the task set. The ratio between the computation time of a task and its period gives the fraction of the processor time spent in executing that task. Thus, the utilization factor U of a task set composed by n tasks is:

$$U = \sum_{i=1}^{n} (\frac{C_i}{T_i}) \tag{2.4}$$

The sufficient schedulability analysis presented in [LL73] consists in the computation of the least upper bound for the task set utilization. For all task sets having a utilization factor below this bound there exist a feasible schedule. The least upper bound is given by the following equation:

$$U = \sum_{i=1}^{n} (\frac{C_i}{T_i}) < n(2^{\frac{1}{n}} - 1) \tag{2.5}$$

This function approaches ($\simeq 0.69$) as n goes to infinity. For task sets with harmonic periods the least upper bound is one, the maximum attainable in single processors. To perform this feasibility test it is required to sum the utilizations of each task. For a task set with n messages this takes n steps, thus the computational complexity of this method is $O(n)$.

Other utilization-based analysis for the RM scheduling algorithm have been proposed, some of them providing exact results ([LSD89]) even for arbitrary deadlines ([Leh90]). However, despite being more complex to compute, they still do not provide timing information for individual tasks, as response-time based schedulability tests do.

An extension of the original analysis of Liu and Layland for non-preemptive systems was presented in [SS93]. In this case high priority tasks can be blocked by running lower priority tasks. This blocking occurs at most once in each task instance activation if a suitable resource access protocol is used (e.g. Priority Ceiling Protocol). For these assumptions, a set of n periodic tasks is schedulable by RM if:

$$\forall i,\ 1 \leq i \leq n,\ \sum_{j=1}^{i-1}(\frac{C_j}{T_j}) + \frac{C_i + B_i}{T_i} \leq i(2^{\frac{1}{i}} - 1) \tag{2.6}$$

where B_i is the time during which task τ_i is blocked by lower priority tasks (priority inversion). The task set is supposed to be ordered by decreasing priorities, i.e., $\forall i, j : 1 \leq i, j \leq n,\ i < j \Rightarrow P_i \geq P_j$.

B_i is determined as follows:

$$\begin{cases} B_i = 0, & P_i = \min_{j=1..n}\{P_j\} \\ B_i = \max_{j \in lp(i)}\{C_j\}, & P_i \neq \min_{j=1..n}\{P_j\} \end{cases} \tag{2.7}$$

where $lp(i)$ denotes the set of tasks having lower priority than task τ_i .

In [LL73] it is also presented a schedulability condition for the EDF algorithm. It relies on the same assumptions of the RM schedulability test above referred. This condition is exact (necessary and sufficient):

$$U = \sum_{i=1}^{n}(\frac{C_i}{T_i}) \leq 1 \tag{2.8}$$

As in the case of RM schedulability test, it is required to sum the utilizations of each task. For a task set with n messages this takes at most n steps, thus the complexity of this method is also $O(n)$.

Response-time based schedulability tests

Several response-time based schedulability tests have been proposed. Particularly interesting approaches are [JP86] and [ABRW91, ABR+93], since they

not only provide schedulability tests for task sets with arbitrary fixed priority ordering, but also provide estimations of the actual worst-case response time of each task.

According to the method presented in [ABR+93], the longest response time of a periodic task τ_i, denoted as R_i, is given by the sum of its computation time (C_i) with the amount of interference that it can suffer from higher priority tasks (I_i), calculated in the critical instant, i.e., the instant in which the combination of the activations of the tasks causes the maximum interference.

$$R_i = C_i + I_i \tag{2.9}$$

The amount of interference due to higher priority tasks is:

$$I_i = \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.10}$$

where $hp(i)$ is the set of tasks with higher priorities.

Combining equations 2.9 and 2.10 results:

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.11}$$

Unfortunately, the response time R_i appears in both sides of equation 2.11. However, it can be used an interactive technique to solve it. Let r_i^n be the nth approximation of the real value of r_i. The successive approximations are generated by:

$$r_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \tag{2.12}$$

The iteration starts with $r_i^0 = 0^+$ and stops when $r_i^{n+1} = r_i^n$. As referred in [ABR+93], it can be shown that $r_i^{n+1} \geq r_i^n$ and so the iteration can be stopped either when $r_i^{n+1} = r_i^n$ or when r_i^n exceeds the task deadline or period (for Deadline Monotonic or Rate Monotonic scheduling policy, respectively). Moreover, in each iteration of Equation 2.12 either $r_i^{n+1} = r_i^n$ and the process is finished, or $r_i^{n+1} > r_i^n$ meaning that (at least) an instance of an higher priority task became ready. Thus, iteration steps are lower-bounded by the lower execution time among the higher-priority task, which

implies that the termination condition is reached in a finite number of steps.

The analysis presented in [ABR+93] also includes the effect of non-preemption due to resource sharing. Moreover, it can be extended to independent non-preemptive systems. In this case Equation 2.9 can still be used but the interference equation must be redefined to include the blocking factor due to lower priority tasks, as follows:

$$I_i = B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{I_i}{T_j} \right\rceil C_j \tag{2.13}$$

The blocking term B_i is still given by 2.7. As in the case of Equation 2.11, Equation 2.13 is also solved iteratively. Note however that Equation 2.13 does not include the computation time of the task τ_i itself, since in non-preemptive systems, once a task is dispatched it cannot be interrupted by other tasks.

Contrarily to what happens in fixed priority systems such as DM or RM, the worst-case response times of a general task set scheduled by EDF are not necessarily obtained with a synchronous pattern of arrival, i.e., when all tasks become ready at the same (arbitrary) time instant. In fact, the worst-case response time of a task τ_i is found in a *deadline busy period,* in which all tasks but τ_i are released synchronously from the beginning of the deadline busy period and at their maximum rate [GRS96]. In order to find the worst-case response time of τ_i, it is necessary to consider several scenarios, in which τ_i has an activation released at time a, while all other tasks are released synchronously, at an arbitrary time instant, usually $t = 0$ [Spu96]. Thus, for a given value of a, the response time of a τ_i instance released at time a is given by:

$$R_i(a) = \max\{C_i, L_i(a) - a\} \tag{2.14}$$

where $L_i(a)$ is the length of the busy period that includes τ_i activation. To compute $L_i(a)$ the following iterative computation is performed:

$$L_i^{(0)}(a) = 0, L_i^{(k+1)}(a) = W_i(a, L_i^{(k)}(a)) + (1 + \left\lfloor \frac{a}{T_i} \right\rfloor)C_i \tag{2.15}$$

where $W_i(a, t)$ includes the contributions of all instances of all tasks except τ_i having absolute deadlines smaller or equal to $a + D_i$, i.e.:

$$W_i(a,t) = \sum_{\substack{j \neq i \\ d_j \leq a + D_i}} min\left\{ \left\lceil \frac{t}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - d_j}{T_j} \right\rfloor \right\} C_j \quad (2.16)$$

The issue of EDF task scheduling analysis on non-preemptive systems was addressed in [GRS96]. As in the case of fixed priorities addressed above, also in systems based on EDF, the schedulability analysis is similar in both the preemptive and non-preemptive cases. The only two differences are:

- Due to the absence of preemption, a task instance with a later absolute deadline can cause blocking, thus inducing priority inversions;
- The calculation of the busy period must be performed until the start time of the task instance instead of its completion time, since, once dispatched, the task instance always executes until completion.

Therefore, Equations 2.14, 2.15 and 2.16 for non-preemptive systems become respectively:

$$R_i(a) = \max\{C_i, L_i(a) + C_i - a\} \quad (2.17)$$

$$L_i^{(k+1)}(a) = \max_{D_j > a + D_i}\{C_j - 1\} + W_i(a, L_i^{(k)}(a)) + \left\lfloor \frac{a}{T_i} \right\rfloor C_i \quad (2.18)$$

$$W_i(a,t) = \sum_{\substack{j \neq i \\ D_j \leq a + D_i}} min\left\{ 1 + \left\lceil \frac{t}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor \right\} C_j \quad (2.19)$$

As in the case of preemptive systems, Equation 2.18 is a monotonic non-decreasing step function, and can be solved iteratively, starting with $L_i^0(a) = 0$.

2.5 Conclusion

This chapter presents a brief overview about the major concepts and challenges concerning real-time systems. Starting from a generic perspective about real-time computer-based control systems, the chapter evolves to issues like task scheduling, scheduling algorithms and schedulability analysis.

Computer-based control systems comprise sensors to gather data from the environment, computers to execute control algorithms and actuators to drive the environment. Some of these activities may have to be performed within strict time bounds. In this case the system is called a real-time system. Moreover, if failing to meet these temporal constraints can be tolerated, the system is called soft real-time, while if such failure can lead to catastrophic results the system is called hard real-time.

For hard real-time systems it is necessary to assign the resources required by the computational activities so that they can be completely served within the required time bounds. Moreover, other requirements commonly found, such as precedence constraints, must also be fulfilled. Scheduling has been a fertile research field, with a large variety of methodologies described in the literature. One important aspect concerns the instant where the schedule decisions are performed. In off-line scheduled systems, scheduling decisions are made prior to system execution, and their results are stored in a table that is used during run-time to trigger the system activities. In on-line scheduled systems the schedule is built during system run-time, based on the instantaneous system requirements.

While in some real-time systems it is possible to characterize in advance all the activities, in others this is either difficult or even impossible at all. In the former case it is possible to schedule the activities off-line. However, the latter type of systems are more efficiently supported by on-line scheduling, since in this case the activities are scheduled for execution based only on the instantaneous system state.

In off-line scheduled systems, once a feasible schedule is found, the real-time behavior of the system in assured. If no such schedule is found, the system designer can tune some of the system parameters and repeat the process the number of times necessary to achieve positive results, since this job is carried before system runtime. However, in on-line scheduled systems, this is not possible, since scheduling is carried during system run-time, and

thus the scheduler must promptly select the activity that should be executed next. Continued real-time behavior can be achieved in this latter case by the execution of appropriate schedulability tests, which reject the admission of activities that may compromise the system real-time behavior.

Distinct schedulability analysis differ in their accuracy and computational cost. Some techniques require less computational resources (e.g. utilization-based) when compared to others that produce more exact results, but incur in higher computational overhead (e.g. response-time based). The issue of computational cost is particularly relevant in on-line scheduled systems that must respond promptly to changes in the system requirements during run-time. To assure continued real-time behavior the schedulability analysis must be performed whenever the requirements change. Thus, in this case, the system responsiveness to such changes depends directly of the complexity of the schedulability analysis.

Chapter 3

Distributed real-time systems

Several definitions of the term "distributed system" can be found in the literature. None of them is completely in agreement with any one of the others, and they depend heavily on the particular “environment and background”. For example, in the COSI project [PD00], meant to assess critically and develop new ways of thinking about social processes, distributed systems *are systems made of a collection of entities (humans, technical systems, insects, etc.) and where decision (control) is totally or partially taken by these entities.* Moving to the field of computer science, Tannenbaum [Tan95] defines a distributed system as *a collection of independent computers that appear to the users of the system as a single computer.* On its hand, Coulouris *et al* [CDK94] go deeper and define a distributed system as a system consisting of a *collection of autonomous computers linked by a computer network and equipped with distributed system software. Distributed system software enables computers to coordinate their activities and to share the resources of the system – hardware, software, and data. Users of a well-designed distributed system should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations.*

The bottom line is that distributed systems comprise multiple autonomous processing units (or entities), cooperating to achieve a common objective or goal. To achieve their goal the processing units need to exchange information, thus each one is attached or integrates a network interface unit providing access to a suitable communication system. This type of system is loosely coupled in the sense that all information exchange is performed exclusively via the communication system using messages.

A distributed real-time system is a distributed system in which there exist real-time activities, i.e., activities that must be carried within specific time bounds. To be accomplished, these time-constrained activities require the execution of tasks in some processing units, which, in its turn, may eventually require the exchange of data with other task(s) that may be executing in different processing units. Thus, to be able to perform real-time activities, the distributed real-time system must be able to execute both tasks and data exchanges strictly within the time boundaries imposed by the timeliness requirements of each of the real-time activities carried out in the system [GH98, TC94].

Distributed real-time systems are required to closely interact with the environment under control. In some circumstances the environment can be completely characterized and its requirements are considered as time invariant. This situation is typically found in distributed computer control systems, in which control engineers specify the control loops based on system dynamics and then generate the timing requirements of the corresponding tasks and messages. However, real systems often do not fit within these restrictive assumptions: complete knowledge about the environment is sometimes too costly or even impossible to gather, environments evolve and thus change their properties during lifetime, upon overload or failure conditions the best possible functionality level must be delivered, etc.. Typical applications fitting in this category are mobile robotics, multimedia and adaptive control systems. To cope with this framework, a distributed real-time system must be operationally flexible, i.e., must be able to adapt itself to the evolving requirements during runtime, without disruption of the services delivered to the system. The flexibility can have several forms: use of adequate scheduling policies, in order to deliver best possible performance in normal situations, but with the capability to change to more robust scheduling policies upon errors or overloads; capacity to accommodate new activities and remove or change the properties of existing activities, in order to adapt to the evolving requirements.

3.1 Real-time communication

Distributed systems comprise a set of autonomous processing devices, which, to accomplish their mission, need to exchange information across the net-

work. Thus, the temporal behavior of the whole distributed system depends not only on the timeliness of tasks executing on each processing device but also on the capability of the underlying communication system to provide message delivery within specific timing requirements [GH98, TC94]. Communication systems able to support such temporal requirements are called real-time communication systems. The remainder of this section addresses some important issues concerning real-time communication.

3.1.1 Event and Time-triggered communication paradigms

Over the last years, a recurring debate concerns the paradigm used for application architectures, with event-triggered (ET) ones being opposed to those based on time-triggering (TT) [Kop93, APF02]. One of the main aspects of this debate concerns the communication infrastructure in distributed applications. This discussion has been fostered by the appearance of the Time-Triggered Protocol - TTP [KG94, Kop99] that highlighted the advantages of that paradigm in real-time communication systems. More recently, such paradigm has also been addressed by the ISO Technical Committee TC22/SC3/WG1 that, in 1999, set up a task force (TF6) to work on the definition of a new CAN-based standard, TT-CAN [Int00], which is a time-triggered profile for CAN. In event-triggered communication, messages are sent by the application upon the occurrence of some event, such as a change in the value of some input. On the other hand, according to the time-triggered paradigm, messages are sent only in precise pre-defined time instants.

Event-triggered communication does seem more ergonomic and even more resource efficient. However, when worst-case requirements are considered, that efficiency is not verified. Since events are asynchronous by nature, a typical worst-case assumption is that all events that must be handled by the system will occur simultaneously. In order to cope with such situation in a timely fashion, the required amount of resources (e.g. network bandwidth) is very high.

On the contrary, the time-triggered approach forces the communication activity to occur at pre-defined instants in time at a rate determined by the dynamics of the environment under control. One of the features of this approach is that it allows relative phase control among the streams of messages to be transmitted over the communication system. By using this feature,

messages of different streams can be set out-of-phase allowing a reduction on the number of messages that become ready for transmission simultaneously.

This feature is responsible for one of the most important properties of time-triggered communication as stressed by Kopetz [Kop97], i.e. the support for composability with respect to the temporal behavior. This property assures that, when two subsystems are integrated to form a new system, the temporal behavior of each of them will not be affected.

This does not hold true for event-triggered communication. In this case, the level of contention at the network access that each subsystem feels before integration is always increased upon integration due to the traffic generated by the other subsystems. Furthermore, the relative phase control allowed by the time-triggered approach may lead to two other positive effects. Firstly, it improves the control over the transmission jitter felt by periodic message streams. Secondly, it supports higher network utilization with timeliness guarantees. Therefore, when considering worst-case requirements, the time-triggered approach is more resource efficient than the event-triggered one. However, when considering average-case requirements, time-triggered communication is considerably greedy when compared to event-triggered one. Consequently, by dimensioning a system according to its worst-case requirements, as typical in hard real-time systems, the time-triggered approach tends to be less expensive than the event-triggered one. Nevertheless, since the average network utilization of event-triggered systems is normally lower, such systems can easily support other types of communication with less stringent or no timing constraints (e.g. traffic associated with the management of either remote nodes or network) without any additional cost. This fact can have a positive impact on the overall efficiency of the communication system utilization, reducing its exploitation costs. Apart from the above considerations on network utilization, it is commonly accepted [TC99] that time-triggered communication is well adapted to control applications that typically require regular transmission of state data, with low or bounded, jitter (e.g. motion control, engine control, temperature control, position control). On the other hand, event-triggered communication is well adapted to the monitoring of alarm conditions that are supposed to occur sporadically and seldom, and also to support asynchronous non-real-time traffic e.g. for global system management.

3.1.2 Combining event and time-triggered traffic

Despite their different characteristics, many applications do require joint support for both event and time-triggered traffic (e.g. automotive [LA99]) and thus, a combination of both paradigms in order to share their advantages is desirable. An important aspect is that temporal isolation of both types of traffic must be enforced or, otherwise, the asynchrony of event-triggered traffic can spoil the properties of the time-triggered one. This isolation is achieved by allocating bandwidth exclusively to each type of traffic.

A typical implementation makes use of bus-time slots called elementary cycles, or micro-cycles (e.g. [RN93]), containing two consecutive phases dedicated to one type of traffic each. The bus time becomes, then, an alternate sequence of time-triggered and event-triggered phases. The maximum duration of each phase can be tailored to suit the needs of a particular application. If each type of traffic is forced to remain within the respective phase then temporal isolation is guaranteed. This concept is used, for example, in the WorldFIP [IEC00], Foundation Fieldbus-H1 [IEC00] and FlexRay [MF02] fieldbuses.

Even protocols relying in a pure TDMA approach usually support event-triggered communications semantics, usually by reserving time for pooling this type of traffic, as in the case of TTP/C [Kop99]. However, in this case, if no transmission request for the respective message is pending the slot is wasted, i.e. unused. This time-based polling mechanism for each event-triggered message causes these ones to be undifferentiated from the time-triggered traffic, inheriting the properties referred in the previous section, particularly high efficiency under worst-case requirements and low efficiency under average-case requirements whenever these are substantially lower than the former ones.

3.1.3 Message Scheduling

Distributed systems usually rely on a shared medium network to interchange data among nodes. Therefore, as for the case of tasks in microprocessors, to be able to meet their timing constraints, messages access to the communication network must also be properly scheduled. Other similarities can be found between message scheduling in communication networks and task scheduling in microprocessors [CM95]; messages can also have distinct time-

liness requirements (soft, hard, best effort) and activation patterns (periodic, sporadic). This resemblance allows the application of several results obtained for task scheduling into message scheduling (e.g. [TBW95] and [Nat00]). Moreover, some of the paradigms found in real-time task scheduling (off-line, on-line with fixed/dynamic priorities) are also found in real-time message scheduling [AF98].

However, message scheduling in distributed real-time systems has its own challenges, due to the particularities of this environment. On one hand, the resource requests are issued by entities spread among the system nodes and thus can not be immediately known, as in the case of centralized systems. Moreover, also due to the systems distributed nature, complete knowledge about the system state is sometimes unavailable, and thus scheduling decisions must be taken based on incomplete information [SS93]. Due to the lack of complete information about the system state or the substantial overhead required to get such information, optimal scheduling techniques developed for microprocessors, when transported to distributed systems frequently do not keep their optimality [MZ95].

Another issue is related to the lack of preemption during message transmissions. Preemptive systems are known to have higher level of schedulability than non-preemptive ones, thus the lack of this feature in message transmission can penalize efficiency. A partial solution to this problem is implemented by most of the available communication protocols and consists in limiting the maximum length of each message, thus avoiding "long" periods of blocking. Long messages sent by the application are broken is several "short" messages (i.e., messages respecting the maximum length defined by the particular communication protocol), transmitted and reassembled at the destination. The counterpart is an increased overhead in systems nodes, required by the break and reassembling procedures.

Real-time communications are usually implemented based on some kind of multiple access networks [MZ95], within somehow limited geographical spaces (e.g. a manufacturing cell, an enterprise building, a ship). System nodes comprise the hardware required to handle the communications (a Network Interface Card) and usually have a layered communication architecture. Each layer has a set of protocols responsible for carrying out the specific operations that must be made available to other layers. Figure 3.1 shows the architecture of the ISO Reference Model for Open Systems Interconnection

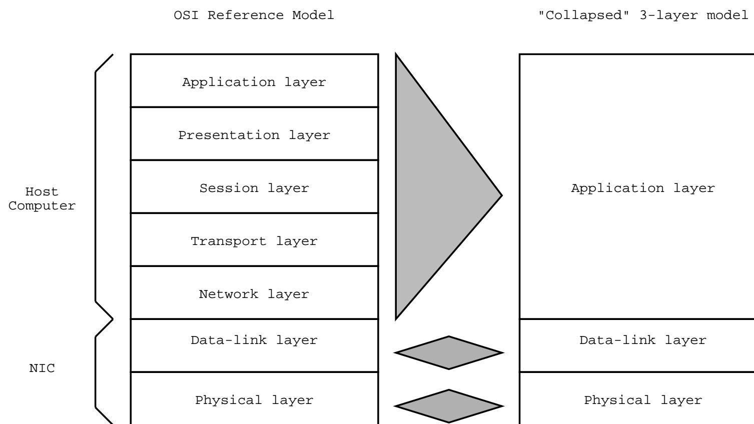


Figure 3.1: Layered communication architecture

[Zim80]. Frequently, real-time communication networks employ a “collapsed” OSI-based architecture, in which the upper 5 layers are merged into a single application layer, as shown also in Figure 3.1. The OSI Reference model was developed for generic communication systems. Many distributed applications are implemented on constrained hardware resources, and thus the implementation of the full OSI reference model can be too expensive in terms of both CPU power and memory, thus the need to some lightweight protocol stack.

Nevertheless, independently of the architectural peculiarities, a communication stack comprises some or all of the following functions: an *application* interface, providing common services required to the particular applications; a *presentation* layer, to provide an uniform data access, independently of the equipment (interoperability); a *session* layer, allowing to open and close dialogs between senders and receivers; a *transport* layer which handles the end-to-end communication; a *network* layer which handles the node addressing and message routing; a *data-link* layer responsible for the access to the communication medium and logical data transfer; and finally a *physical* layer, which handles the way the messages are transmitted physically over the communication medium (pins assignments, number of wires, electrical characterization, repeaters).

The performance of the communication system as a whole strongly depends on the performance of each one of its layers. New techniques have been recently proposed to enhance the performance concerning the time spent in

the internal processing at the different protocol layers, for example by providing distinct queues and paths for real-time and non-real-time traffic (e.g. [SJH02]). However, the data-link layer is of utmost importance, since it is this layer that is responsible for deciding when nodes can access the bus and for how long. Medium access control (MAC) protocols can be classified in two classes: controlled access and uncontrolled access ([Tho98]). In the former class, access to the communication channel is handled by a particular mechanism which is responsible for ensuring that collisions (i.e. simultaneous message transmission by two or more distinct nodes) cannot occur. Commonly used mechanisms are: master-slave, token passing and Time-Division Multiple Access. Concerning the latter class, uncontrolled access, no global arbitration method exists and thus collisions can occur. However, special mechanisms are used to detect these events and resolve them. Carrier-Sense Multiple Access based protocols, such as Ethernet, use this method.

A comprehensive study and classification of access protocols suited to real-time communication over multiple-access networks is presented in [MZ95]. In this work the MAC protocols are described as consisting of two processes: access arbitration and transmission control. The access arbitration process determines when a node can access the communication channel to send messages; the transmission control process determines for how long a node can continue to use the channel to send messages. Examples of protocols relying either in access arbitration or transmission control are also presented. Furthermore, in this work it is also presented and analyzed, in terms of efficiency and message timeliness, the implementation of several scheduling policies (e.g. Rate Monotonic, Minimum-Laxity-First).

3.1.4 Co-operation models

As referred in the beginning of this section, distributed systems comprise multiple autonomous processing units, cooperating to achieve a common objective or goal. Information exchange is carried by a suitable communication system and consists not only in the physical transmission of the message across the network but also in the way it is distributed by the nodes in the network, i.e., co-operation model. Depending on the particular application, nodes may need data that resides in one or more other nodes. Moreover, the same data can also be needed in several distinct nodes. Thus, communication can be one-to-one, one-to-many, many-to-one and many-to-many.

Two well-known co-operation models are the **producer-consumer** and **client-server** [VR01].

The **producer-consumer** model associates unique logical handles to each message type. Messages are generated and received based only on these logical handles, without any explicit reference to the particular source or destination nodes of the messages. Consumer nodes select the logical handle(s) related to the data they need to perform their own computations and receive all messages identified with those handle(s). Producer nodes need not to know who and how many are the consumers of its data, and conversely receiver nodes need not to know which particular node is the producer of the data.

The producer-consumer co-operation model inherently supports one-to-one and one-to-many communication, without incurring in spatial data consistency problems, since the same data message is used to update all the local images in all the consumer nodes in the network. However, this property can be lost if the underline communication network does not support atomic broadcasts. In this case, due to errors during transmission, some nodes can receive correctly a message while others can receive the very same message with errors. If this situation happens, different nodes can end up with different images of the same entity, i.e., spatial inconsistency.

On the other hand, this model does not solve the problem of temporal consistency. Whenever there are several producer nodes, there is contention for message transmission on the network among the several producers, and therefore some messages are delayed in this process, which can result in outdated values sent to the bus. This problem has been solved by the **producer-distributor-consumer** (PDC) model [Tho93], which adds a coordination layer to the producer-consumer model. In the PDC model the producers behave as slaves with respect to an arbitrator node (called master), and thus only transmit messages when authorized. The master node is fed with the properties and temporal requirements of the messages that are exchanged by the bus and builds a suitable schedule, which, then is used to grant the producers the right to transmit their messages.

Another approach is the **client-server** model. In this case, nodes that are producers of some data that can be required elsewhere in the network behave as servers. The nodes that need the data (clients) issue requests to the respective server, which in its turn replies with the demanded data value.

This communication model is inherently one-to-one, and can lead to both spatial and temporal data inconsistency problems when used to support one-to-many or many-to-one communication. For instance, if the same data is required in several nodes, different nodes issue the respective requests to the server. If the data value changes during this period, the successive replies of the server will carry different values for the same entity, resulting in spatial inconsistency. On the other hand, when a node needs data from different servers, it must issue the requests sequentially, one after the other, which can result in temporal inconsistency. Another problem posed by this model is related with the internal scheduling and processing of requests inside the servers. The requests reach the servers asynchronously and take some time to be processed, thus the time required to handle a particular request depends on the request arrival pattern, which is not deterministic [VR01].

3.2 Fieldbus Protocols - brief survey

Over the last 30 years a large number of real-time communication protocols for distributed computer-controlled systems have emerged, developed by different companies and organizations all over the world. These protocols, known as fieldbuses, are used at the field level to interconnect devices such as sensors, PLCs and actuators. Although fieldbuses are to some extent similar to general-purpose local area network protocols, they are tailored to fulfill the specific requirements of real-time computer-controlled distributed systems, such as [Pim90, Dec01]:

- Handle short messages in an efficient manner;
- Support for periodic traffic with different periods as well as aperiodic traffic;
- Bounded response time;
- No single point of failure;
- Low cost, both at the device level as well as at the infrastructure installation and maintenance levels.

In the remainder of this section it will be presented a brief overview of some of the most relevant fieldbus protocols, with special emphasis on the scheduling

paradigms, support for dynamic communication requirements, handling of event and time-triggered traffic and temporal isolation between TT and ET traffic. Particular attention is devoted to the CAN protocol, since one of the FTT paradigm implementations is based on it.

3.2.1 The Controller Area Network (CAN) protocol

The Controller Area Network [Rob91] (CAN) protocol was developed in the mid 1980s by Robert Bosch GmbH, aiming at automotive applications, to provide a cost-effective communications bus for in-car electronics and as an alternative to expensive and cumbersome wiring looms. It is standardized as ISO 11898-2 [ISO93] for high speed applications (1Mbps) and ISO 11519-2 [ISO94b] for lower speed applications (125Kbps). The transmission medium is usually a twisted pair cable and the network maximum length depends on the data rate. Due to its bitwise arbitration mechanism, it is required that the bit time must be long enough to allow the signal propagation along the entire network as well its decoding by other stations, which imposes a fundamental limit to the maximum speed attainable (e.g. 40m @ 1 Mbps; 1300m @ 50 Kbps).

CAN uses a multi-master bus architecture and employs the Carrier Sense Multiple Access with Non-destructive Bitwise Arbitration (CSMA/NBA) mechanism. It uses a priority arbitration scheme based on numerical identifiers to resolve collisions between nodes trying to transmit at the same time. A logical zero on the bus is dominant (dominant bit) and overwrites a one (recessive bit). Therefore, if a node transmits a logical one whilst another transmits a logical zero, the resulting logical level on the bus is zero (the one is overwritten). A node wishing to transmit must first sense the bus, and it can start to transmit the message only when there is no activity (CSMA), starting by the identifier, most significant bit first. During the transmission the nodes also monitor the bus state. If a node transmits a recessive bit and senses a dominant bit in the bus, it infers that an higher priority message is also being transmitted and thus gives up from the arbitration process. Therefore, the node transmitting the message with the highest priority among the ones that where ready in the beginning of the arbitration process wins the arbitration process. Nodes that loose the arbitration process must wait for the bus to become free again before trying to re-transmit its message. This arbitration scheme does not consume bandwidth, i.e., the transmission time

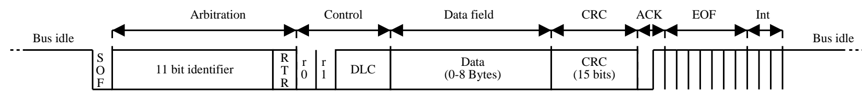


Figure 3.2: CAN 2.0A message frame

of the message that wins the arbitration process does not depend on the number of messages that contend for the bus access.

A CAN message frame (Figure 3.2) consists of: identifier, data, error, acknowledgment and CRC fields. The identifier field consists of 11 or 29 bits (CAN 2.0A/2.0B respectively) and the data field can carry between 0 and 8 bytes. When a device transmits a CAN message it first transmits the identifier field followed by the data field. The identifier field determines which node gains access to the bus. Individual nodes can be programmed to accept messages with specific identifiers. In this case, an internal data transfer will occur if the identifier of the transmitted message matches the identifier of the message which the node is configured to receive. On the other hand, nodes that are not programmed with the same identifier as the transmitted CAN message will not receive the message. This is known as acceptance mask filtering and is normally performed by the CAN hardware. The RTR bit is used for a remote transmition request. When this bit is set, the CAN frame has an empty data field. The node that transmits messages with that identifier will send a message, carrying the requested data, in reply to this request.

CAN controllers have transmit and receive error counters which register errors during transmission and reception respectively. These counters are implemented in hardware and are incremented or decremented (with different weights) by each erroneous or correct message transmission or reception events. During system runtime the error counters may increase even if there are fewer corrupted frames than uncorrupted ones. During normal operation the CAN controller is in its error-active state. In this state, the node is able to transmit an active error frame every time a CAN frame is found to be corrupted. If one of the error counters reaches a warning limit of 96 error counts, indicating significant accumulation of errors, this is signaled by the controller usually using an interrupt. The controller then operates in its error active mode until a limit of 127 error counts has been exceeded. Once 128 error counts has been reached, the CAN controller enters an error-passive

state. In this state, an error-passive controller is still able to transmit and receive messages but signals errors by transmitting a passive error frame. If the error count reaches or exceeds a limit of 256, the controller enters its Bus-OFF state. In this state the controller can no longer transmit or receive messages until it has been reset by the host processor, resetting its hardware counters back to zero.

In real-time message scheduling, the computation of the transmission time of messages is of paramount importance, since it is required to perform any kind of analysis. To provide clock information embedded in the bit stream, CAN does not allow the transmission of more than 5 consecutive bits of the same polarity. When such situation occurs in the data to be transmitted, CAN automatically inserts a bit of opposite polarity. By reversing the procedure, these bits are removed at the receiver side. This technique, called *bit-stuffing*, implies that the actual number of transmitted bits not only can be larger than the size of the original frame, but also can vary in consecutive instances of the same message, depending on the particular message instance contents. According to the CAN standard [Rob91], the total number of bits in a CAN frame without bit-stuffing is given by Equation 3.1, where DLC is the number of bytes of payload data in a CAN frame ([0, 8]) and 47 is the number of control bits (Figure 3.2).

$$CAN_LEN_{No_Stuff} = 47 + 8 * DLC \tag{3.1}$$

The CAN frame layout is defined such that only 34 of these 47 bits are subject to bit-stuffing. Therefore the worst-case number of bits after bit-stuffing is given by Equation 3.2 ([NHNP01]).

$$CAN_LEN_{Stuff} = 47 + 8 * DLC + \left\lfloor \frac{34 + 8 * DLC - 1}{4} \right\rfloor \tag{3.2}$$

3.2.2 WorldFIP

The WorldFIP protocol (European fieldbus standard EN50170 ([CEN96]) and international standard IEC61158 ([IEC00])) is based on the producer-distributor-consumers (PDC) communication model [Tho93] according to which process variables are made available by producer nodes, one at a time, and are distributed to consumer nodes that use them.

The distributor function is performed by the bus arbitrator (BA) which schedules the producers access to the bus. The addressing method is based on identified variables, i.e., the addressed entities are variables to be exchanged and not nodes. At each network node, the protocol data link layer (DLL) manages a set of buffers holding the values for the variables to be exchanged. These buffers are available locally to the application software through application layer (AL) services, which allow writing to or reading from such buffers. The contents of the DLL buffers in the consumer nodes are automatically updated by the communication system through a network service called buffer transfer. Each buffer transfer corresponds to an atomic network transaction which includes an identification frame (ID_DAT) sent by the BA with the identification of the variable to be produced and a response frame (RP_DAT) sent by the node that produces the identified variable, containing the respective updated value. The consumer nodes receive the response frame and overwrite the respective DLL buffer of the identified variable with its new value.

As referred above, the bus access arbitration is centralized and performed by a particular node called Bus Arbitrator (BA). At run-time, the BA uses a static schedule table, the BAT, to schedule periodic transactions. This table is usually built off-line, prior to the system operation. Two important parameters are associated with a WorldFIP BAT: the elementary cycle (EC) and the macro-cycle (MC). The elementary cycle determines the resolution available to express the variable's scan periods. The inverse of its duration represents the maximum rate at which the BA may scan any variable. Usually, the EC duration is set equal to the maximum common divider of the variable's scan periods. The BAT contains the sequence of ECs that describe the network periodic traffic during one Least Common Multiple (LCM) period, which is called the macro-cycle.

Aperiodic message transfers are carried after the last periodic transaction of the EC, if enough room is available (*Aperiodic window* in Figure 3.3). The aperiodic buffer transfer takes place in three steps:

1. When transmitting a periodic data frame, a node having buffered aperiodic messages signals this status by setting the aperiodic request bit in the data frame (RP_DAT);

2. The BA collects the aperiodic requests and latter on, in the aperiodic

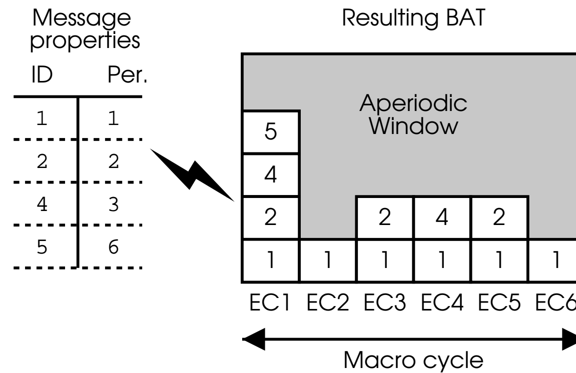


Figure 3.3: Periodic message properties and resulting BAT

window, queries the nodes for the list of pending aperiodic requests;

3. Finally, the BA processes the list of pending requests using the same mechanism as for periodic buffer transfers, but using the aperiodic window.

Over the last years scheduling and schedulability analysis issues of WorldFIP networks have been addressed is several academic works.

Concerning specifically the aperiodic requests, *Vasques and Juanole* [VJ94] derive an upper bound to the worst-case response time for the aperiodic requests, which includes the load due to the periodic transfers during the whole MC and the time required by all other aperiodic requests that can be issued during that period of time. In [PB97], *Pedro and Burns* propose a less pessimistic analysis based on a lower bound to the aperiodic window of each EC. *Almeida et al* present an improved schedulability analysis for both the periodic [AF99] and aperiodic traffic [ATFV01]. This work is based on the construction of a timeline, and can be used for on-line admission control.

Dworzecki [Dwo98] presents a scheduling technique which claims to be more efficient than RM and EDF. The computational complexity of the approach presented by the author is considerably higher than the RM/EDF schedulers, however, since the BAT is built offline, such impact has a limited relevance.

Kim et al [KJK98] present a methodology to reduce both the amount of memory required by the BAT and the message release jitter. An offline built BAT must hold the schedule for the duration of a macro-cycle. When the

message set has messages that have relative prime periods, the LCM, and thus the BAT size, can become very large. To reduce this effect, the authors reduce the larger scan periods, avoiding relative prime values. Concerning the message release jitter issue, the authors propose to reduce the message's scan periods until they become harmonic in powers of 2. In this case it becomes possible to build a jitter free schedule. However, both of these methods imply an increase in the bandwidth utilization, and thus reduce the system schedulability.

Some effort has also been devoted to add support for dynamic message sets to the WorldFIP protocol. For example, *Almeida et al* [APF99] propose on-line planning-based scheduling and admission control techniques. With this approach, the BAT is periodically built, based on the current message properties. Thus, if these change, in its next invocation the scheduler uses the updated values to build the BAT. On the other hand, changes are always subject to admission control. Therefore the timeliness guarantees are not compromised despite the dynamic environment.

3.2.3 Profibus

The Profibus protocol (European fieldbus standard EN50170 ([CEN96]) and international standard IEC61158 ([IEC00]) is a fieldbus network designed for deterministic communication between computers and PLCs and field devices such as sensors, valves, etc. The Profibus MAC protocol is based both on token passing between masters and master-slave between master and slave nodes. Token passing is used between master stations to grant the bus access to each other. When a particular master holds the token, it uses a master-slave procedure to communicate with slave stations.

The MAC is implemented at the layer 2 of the OSI reference model, and in Profibus is called Fieldbus Data Link (FDL). The FDL layer is responsible for controlling the bus access and for providing data transmission services.

The data transmission services supported by the Profibus protocol are message broadcasting (from masters) and one-to-one communication between masters and slaves. Only the master holding the token is allowed to send broadcast messages or initiate a transaction with one slave. Slave nodes, when pooled by a master, must respond in a bounded time ("immediate-response"). This is particularly important for the real-time operation of the protocol, since it allows to upper bound the transactions duration, and thus

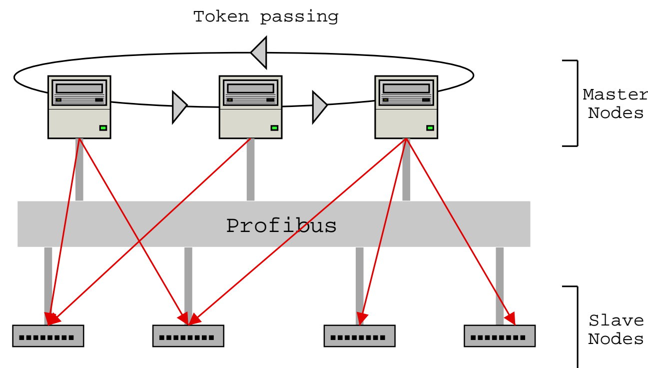


Figure 3.4: Profibus token-passing and master-slave relations

perform worst-case computations. A message cycle consists of a master's action frame (send, request or send/request frame) and the associated responder's acknowledgment or response frame, which, as referred above, is constrained to arrive within a predefined time, called slot time. If the response is not received by the master, the request is repeated. The number of retries before a communication error report is defined during system setup in all master stations. This is part of the cycle time and is the major source of the pessimism in the existing analysis.

The Profibus FDL layer supports a poll list, used for cyclically polling the network slaves (e.g. sensors). On the other hand, network sharing among masters is accomplished by a set of rules constraining the amount of time that each master can hold the token. After receiving the token, the measurement of the token rotation time begins and stops at the next token arrival, resulting in the real token rotation time (TRR). In a Profibus network, a target token rotation time (TTR), common to all masters, is pre-configured. The value of this parameter must be carefully choosen to meet the responsiveness requirements of all masters. When a master receives the token, it computes the token holding time (TTH), which is given by the value of the difference, if positive, between TTR and TRR.

In Profibus there are two distinct classes of messages, high-priority and low-priority, using two independent outgoing queues. If a late token is received, i.e. real token rotation time (TRR) greater than the target token rotation time (TTR), the master station may execute at most one high-priority message cycle. Otherwise, the master station may execute message

cycles while TTH is greater than zero. Note that the TTH is always tested at the beginning of the message cycle execution, therefore, once a message cycle is started, it is always completed, including any required retries, even if TTH expires during the transaction (TTH overrun). Low-priority message cycles are executed only if there are no high-priority messages pending and TTH is greater than zero.

Low-priority messages are further subdivided in three subtypes: poll list, non-cyclic low-priority and Gap List message cycles. As referred above, the poll list messages are used for cyclically polling the network slaves, and are processed after all the high-priority messages being handled. If the poll cycle is completed and the master still can hold the token (i.e. TTH $>$ 0), it then processes the non-cyclic messages, which are produced by the application layer and remote management services.

The Gap is the address range between two consecutive master addresses, and each master periodically checks the Gap addresses to handle dynamic changes in the logical ring.

The timeliness analysis of real-time traffic has been addressed in [TV99b]. The message queues in Profibus are First-Come-First-Served, which can cause timeliness problems in heavily loaded networks. Enhancements to the protocol consisting on local priority-based message scheduling have been presented in [TV99a] and [CMTV02].

3.2.4 P-Net

The P-NET protocol (European fieldbus standard EN50170 ([CEN96]) and international standard IEC61158 ([IEC00]) is a multi-master standard based on virtual token-passing scheme among masters and master-slave between masters and slaves. The system architecture is similar to the presented in Figure 3.4, relatively to the Profibus protocol.

In a P-NET system each master has a node address (NA), between 1 and the number of masters expected within a system. All masters contain an *Idle Bus Bit Period Counter* (IBBPC) which is incremented for each bit period the bus is idle and reset to zero when bus activity is detected. Each master also has an *Access Counter* (AC), which is incremented when the idle bus bit period counter reaches τ =40 bit periods ($520\mu s$ at 76.8Kbps). If a master has nothing to transmit, or indeed is not even present, the bus will continue to be inactive. Following a further period of σ =10 bit periods

($130\mu s$ at 76.8Kbps), the idle bus bit period counter will have reached 50, so all the access counters will be incremented again, allowing the next master to access the bus. The virtual token passing will continue every 10 bit periods, until a master does require access. When the access counter exceeds the maximum number of masters, it is preset to 1. To avoid loss of synchrony during long idle periods, when the IBBPC counter becomes higher or equal to 360, the token master should send a *sync* frame. This frame does not carry any meaningful data, but causes all the IBBPC counters to be cleared, resulting in AC counters synchronization.

The P-NET standard allows each master to perform at most one message cycle per token visit. After receiving the token, the master must transmit a request before a certain time has elapsed. This is denoted as the master's reaction time, and the standard imposes a worst-case value of up to $\rho = 7$ bit periods. A slave is allowed to access the bus between 11 and 30 bit periods after receiving a request. This delay is denoted as the slave's turnaround time. The limitation to one message cycle per token visit together with the upper bounds on the master's reaction time and slave's turnaround time allow to perform timeliness analysis, and thus evaluate if, for a given message set and system topology, the timing requirements are completely fulfilled.

P-Net has some interesting features, like the low overhead required in the nodes to implement the protocol and the simplicity of dynamically adding and removing nodes. However, the master-salve transmission control technique combined with the restriction of being possible only one message cycle per token visit limits the performance of this protocol.

The timeliness analysis of real-time traffic has been addressed in [TV98b]. As in the case of Profibus, P-NET message queues are First-Come-First-Served, thus potentially causing the same timeliness problems in heavily loaded networks. Enhancements to the P-NET protocol, also consisting on local priority-based message scheduling have been presented in [TV98a].

3.2.5 DeviceNet

DeviceNet [OODVA97] was developed by Rockwell Automation as an open fieldbus standard based on the CAN-protocol. It was designed specifically for automation technology. The Open DeviceNet Vendor Association, Inc (ODVA) is responsible for the specification and maintenance of the DeviceNet standard.

DeviceNet is part of a family of three open network standards (DeviceNet, ControlNet and EtherNet/IP) that use a common application layer (ISO Layer 7), designated by *Control and Information Protocol* (CIP). The control part of CIP handles the exchange of real-time I/O data, while the information part of the CIP defines the exchange of data for configuration, diagnosis and management.

DeviceNet defines two different types of messaging: I/O Messaging and Explicit Messaging.

I/O messages are for real-time control-oriented data and provide a dedicated communication path between a producing application and one or more consuming applications (one-to-many co-operation model). Typically high priority identifiers are assigned to these messages and use source addressing (i.e. the ID CAN field identifies the data and not the sender or destination devices). I/O messages are not constrained concerning their length, and thus fragmentation is supported. The DeviceNet Communication Protocol is based on connections, which must be established before the start of the communications. The process of connection establishment reserves system resources, such as CAN ID address ranges.

Explicit messages provide multi-purpose, point-to-point communication paths between two devices and are used to perform node configuration and diagnosis. Explicit messages typically use low priority identifiers and contain the specific meaning of the message right in the data field.

DeviceNet supports both periodically triggered traffic and event-based traffic.

The periodically triggered traffic (cyclic option) is used typically in control-loops. In this case the application associate a specific period to each state variable, and the protocol performs the transmission of the respective messages according to the respective period.

With event-based traffic, a device only produces its data when the variation on its value since the last transmission exceeds a given pre-defined value. DeviceNet provides a membership service for sources of this type of data by means of an adjustable background heartbeat rate. Devices send data whenever it changes or the heartbeat timer expires. With this method consumer nodes detect a failure in a producer node if no data is received during a period of time exceeding the heartbeat period.

By default, both change of state and cyclic are acknowledged exchanges.

However the protocol allows to selectively suppress acknowledges, which is useful for applications that exhibit fast changes of state or cyclic rates.

3.2.6 TT-CAN

Time-Triggered CAN (TT-CAN) [Int00] is another communication protocol based on CAN. As discussed in Section 3.2.1, CAN prioritizes messages according to their ID field using bitwise arbitration. Nevertheless, a CAN message can be delayed if some other message is already in the process of transmission, independently of their relative priorities, or if another message with higher priority also competes for the bus. Lower priority messages, due to interference of higher priority messages, can potentially suffer high latency jitter in the media access.

Considering these drawbacks, TT-CAN goals are to reduce latency jitters, guarantee a deterministic communication pattern on the bus and use the physical bandwidth of a CAN network more efficiently.

Communication in TT-CAN involves the periodic transmissions of a *reference message* by a special network device called time master. This reference message introduces a system-wide reference time. With synchronized nodes, messages can be transmitted at specific time slots, without competing with other messages for the bus (exclusive windows), thus contention on bus access is avoided and the latency time becomes predictable and independent of the message's CAN identifier. Exclusive windows ownership is defined at pre-runtime, during system design. Moreover, TT-CAN also allows to reserve time slots for shared access, in which several messages can try to be transmitted on the same time slot (arbitration windows). In this case the protocol relies on a contention resolution mechanism that is based on CAN, except that message transmission is made in *single-shot*, i.e., nodes do not try to retransmit the message when they loose arbitration. This mechanism is required to ensure that arbitrating windows do not overrun their respective pre-allocated time. Independently of being transmitted on exclusive or arbitrating windows, messages have the CAN standard format. Moreover, because TT-CAN preserves the original CAN CSMA/NBA channel access protocol for event messages, it is inherently limited to a 1 Mbps (or lower, depending on the bus length) data transmission rate.

The period between two consecutive reference messages is called the basic cycle. A basic cycle consists of several time windows, which can be of

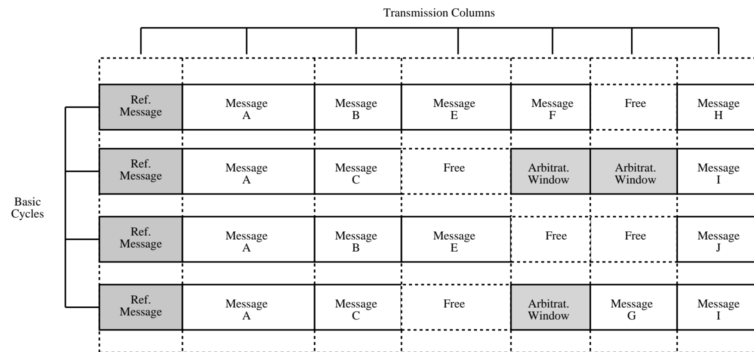


Figure 3.5: TT-CAN system matrix

different sizes and types (exclusive or arbitration). Several basic cycles may be combined to build the so-called system matrix, which completely characterizes the communication pattern (Figure 3.5). The sequence of basic cycles in the matrix cycle is controlled by the reference messages. A TT-CAN node is not required to know the whole system matrix, but instead it is only required to know the time marks that are necessary to define the time slots assigned to messages transmitted by the node itself and to check for received messages. The structure of the basic cycle is the same for all cycles within the system matrix, meaning that all the transmission columns have the same width, usually corresponding to the length of the longest message that is transmitted in the respective column.

3.2.7 TTP/C

The TTP/C [Kop99, TTT] protocol is a reliable and fault-tolerant communication protocol, designed to permit high performance data transmission, clock synchronization, membership services, fast error detection and consistency checks. A TTP/C network consists of a set of communicating nodes connected by a replicated interconnection network (Figure 3.6). A node computer comprises a host computer and a TTP/C communication controller with two bi-directional communication ports. Each of these ports is connected to an independent channel of a dual-channel interconnection network. Via these broadcast channels the nodes communicate using the services of the communication controller.

The TTP/C protocol implements broadcast communication that pro-

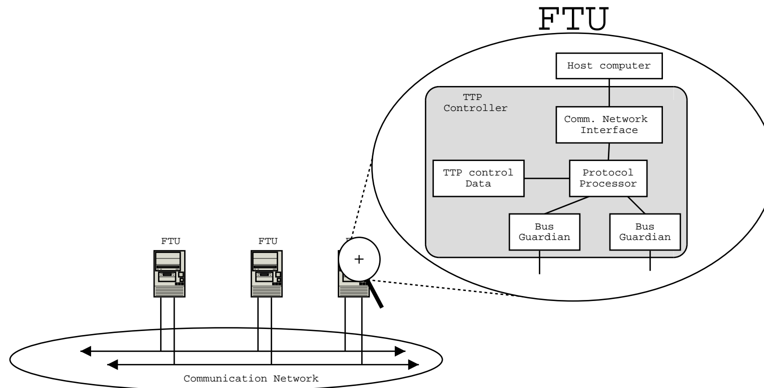


Figure 3.6: TTP/C architecture

ceeds according to an a priori established time-division multiple access (TDMA) scheme. This TDMA scheme divides time into slots, each being statically assigned to a particular node, and, during its slot, each node has exclusive write permission to the network. The slots are grouped in the so-called TDMA rounds. In a TDMA round every node is granted write permission in at least one slot, and the access pattern repeats itself in successive rounds.

A distributed fault-tolerant clock synchronization algorithm establishes the global time base needed for the distributed execution of the TDMA scheme. Nodes can send different messages in different TDMA rounds, although the slot length is constrained to be the same. To handle this feature, the protocol defines *cluster cycles,* comprising several TDMA rounds with all the possible message combinations.

Each node holds a data structure containing the message descriptor list (MEDL), where the data concerning the complete data communication pattern is stored. The MEDL contains the information relative to all messages exchanged on the system, which, combined with the global time-base, allows fast detection of missing messages.

The TTP/C protocol provides frames for application data (N-frames), protocol-state information exchange (I-frames) and mixed user data and protocol information (X-frames).

To allow for integration of nodes into an active cluster, some nodes of the cluster periodically broadcast actual network controller state (C-State) in I-frame or X-frame messages. Nodes willing to integrate listen to these

frames to acquire membership status, global time and the actual position within the TDMA round (synchronization process) and then become active.

Message scheduling in TTP/C is performed at pre-runtime, which turns out this protocol unsuited to handle dynamic message sets. Nevertheless a limited degree of flexibility still exists, both due to the possibility of pre-configuring several modes of operation and to the possibility of reserving TDMA slots for later expansion.

Up to 30 global modes can be pre-configured and can be requested by any node, out of a user-specified set of nodes, by using dedicated messages (Mode change request and Clear Mode change request). The execution of a mode change is globally synchronized by the communication protocol. Static information indicating which node may request which mode change at which time is also included.

When building TTP/C communication schedules, a certain percentage of the available bandwidth is assigned to the pre-defined communication requirements. The remaining bandwidth is statically assigned for future expansion of specific existing nodes, and/or nodes to be added at a later time.

During system configuration, the TTP/C protocol allows the reservation of an a priori specified number of bytes for the transmission of event-triggered messages in the time slots. This implies that the bandwidth assigned for aperiodic message transmission cannot be shared among nodes. Thus, effectively, event-triggered traffic is handled as the periodic one, which leads to a poor efficiency, since, typically, the occurrence of such events is seldom, and thus, most of the time, the allocated bandwidth is not used.

3.2.8 FF-H1

The Foundation Fieldbus H1 (FF-H1) protocol (international standard IEC 61158 [IEC00]) was developed to interconnect field devices such as sensors, actuators and controllers, both in manufacturing and process industries.

Foundation Fieldbus defines two device types: basic device and link masters. A link master is any device that can become a Link Active Scheduler (LAS). Conversely, a basic device does not have such property. At any instant each network link has one and only one Link Active Scheduler (Figure 3.7). At link boot or upon failure of the existing LAS, the link master devices on the segment bid to become the LAS. The link master that wins the bid

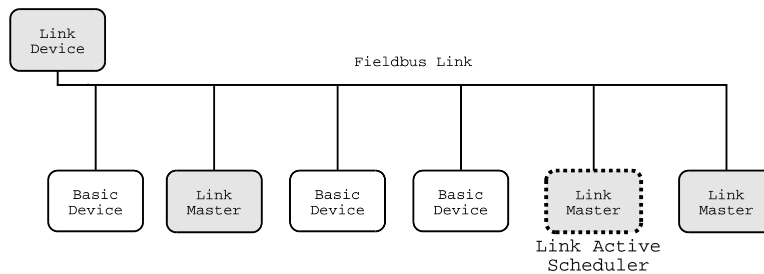


Figure 3.7: Foundation Fieldbus link

begins operating as the LAS. Link masters that do not become the LAS act as basic devices. Link masters can act as LAS backups by monitoring the LAS activity and then bidding to become the LAS when a LAS failure is detected.

The LAS operates as the bus arbiter for the link, and must perform the following tasks:

- To recognize and add new devices to the link;
- To remove faulty devices from the link;
- To distribute Data Link (DL) and Link Scheduling (LS) time;
- To poll devices for process loop data (scheduled transmission times);
- To distribute a priority-driven token to devices between scheduled transmissions.

Network time synchronization is achieved by means of Time Distribution (TD) messages, periodically broadcast by the LAS. The global network time-base is used both to perform the scheduled message transmissions and to schedule user application functions blocks, i.e., functions that describe device's functions and define how these can be accessed.

In each link only one device can communicate at a time. Permission to communicate on the bus is controlled by the LAS and granted to link devices by means of a token. Only the device with the token can communicate. The LAS uses four types of tokens.

A time-critical token, **compel data** (CD), which is sent by the LAS according to a schedule.

A non-time-critical token, **pass token** (PT), which is sent by the LAS to each device in ascending address order. Upon reception of the PT token, devices can send unscheduled messages.

An **execute sequence** (ES) token is used to pass a delegate token to other LM in the network, allowing them to initiate transactions during the period of time specified in the ES token.

The LAS maintains a list of all devices that need access to the bus and are active, which is called the Live List. **Probe node** (PN) messages are periodically sent to nodes that are absent from the live list, allowing their integration, for instance, when a device is connected during system operation. Changes to the live list are broadcast by the LAS to synchronize the other link master's live list according to the current system status.

The Foundation Fieldbus protocol supports several co-operation models:

Publisher/Subscriber: used to transfer critical process data, such as process variables. The publisher entity posts the data in a local buffer. This buffer only contains room for a single data instance, thus if the application updates the data, the old value is overwritten. The value of the data is broadcast to the subscribers when the publisher device receives the corresponding CD command from the LAS. Transfers of this type can be scheduled periodically.

Report Distribution: used to broadcast and multicast event and trend reports. Transfers of this type are queued and delivered to the receivers in the order transmitted. These transfers are unscheduled and occur between scheduled transfers. There is no flow control, therefore corrupted messages are not retransmitted.

Client/Server: used for request/response exchanges between two devices. These transfers are sent and received in the order submitted for transmission, according to their priority, and with queuing. In this case transfers are flow controlled and employ a retransmission procedure to recover from corrupted transfers.

Scheduled data transfers are typically used for the regular cyclic transfer of process loop data between devices on the fieldbus. Scheduled transfers use publisher/subscriber type of reporting for data transfer. The Link Active Scheduler maintains a list of transmit times for all publishers in all devices that need to be cyclically transmitted. When it is time for a device to publish data, the LAS issues a Compel Data (CD) message to the device. Upon

receipt of the CD, the device broadcasts (publishes) the data to all devices on the fieldbus. When the LAS uses one of the specified scheduling profiles known as *dynamic*, on-line change requests to the scheduling table can be performed, which are accepted only if the resulting schedule is feasible.

Unscheduled transfers are used for operations like set point changes, mode changes and software upload/download. Unscheduled transfers use either report distribution or client/server type of reporting for transferring data. All of the devices on the fieldbus are given a chance to send unscheduled messages between transmissions of scheduled data. The LAS grants permission to a device to use the fieldbus by issuing a pass token (PT) message to the device. When the device receives the PT, it is allowed to send messages until either it has finished or the maximum token hold time has expired. This kind of transfers is handled in a best-effort way, meaning that no timeliness guarantees are provided by the protocol. However, the protocol specifies three levels of priorities (urgent, normal and time-available), that correspond to distinct levels of QoS. The PT defines the priority level(s) being served, which depend on the token rotation time. The priority is lowered in case of early tokens, and increased in case of late tokens.

The Foundation Fieldbus protocol allows the interconnection of several fieldbus links into a Foundation HSE (High Speed Ethernet) backbone by means of Link Devices (Figure 3.7). This supports system-wide communication, even between devices residing on different links.

3.2.9 FlexRay

FlexRay [Con01] is a protocol that specifically aims at efficiently combine time-triggered and event-triggered communication. The latter type of communication is based on the ByteFlight [PBG99] communications link invented by BMW for airbag systems. This protocol was developed specifically for advanced automotive control applications, being supported by companies like BMW, GM, Bosch, Motorola and Philips. The constraints of such environment, namely the need to limit the number of different communication systems within vehicles, motivated the quest for a fieldbus providing high data rate, determinism and fault-tolerance, but also with some degree of flexibility, in order to support a broader range of in-vehicle subsystems.

Unlike most of the fieldbus protocols, FlexRay presents a 4-layer architecture, comprising:

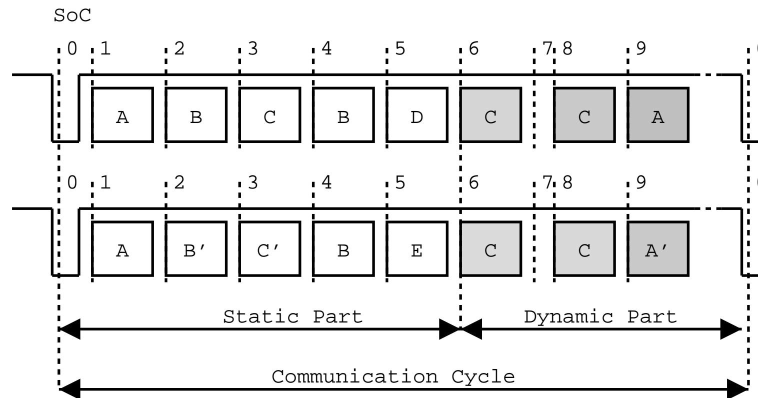


Figure 3.8: FlexRay communication cycle structure

Application Layer: application software;

Presentation Layer: frame filtering and frame status handling;

Transfer Layer: fault confinement, error detection and signaling, frame validation, frame format, synchronization, timing;

Physical Layer: fault confinement, error detection and signaling, error confinement in the time-domain, bit transmission.

Concerning the network topology, FlexRay supports both star and bus topologies, with optional redundant communication channels.

Both synchronous (time-triggered) and asynchronous (event-triggered) data transmissions are supported by FlexRay. Communication is done in fixed duration time slots, designated communication cycles (CC), which contain a static and a dynamic part (Figure 3.8). Synchronous traffic is transmitted within the static part and asynchronous traffic is transmitted in the dynamic part. Any of these parts may be empty, thus a CC can contain only synchronous traffic, only asynchronous traffic or a mix of both.

The communication cycle starts with special control symbol (SoC), followed by the so-called sending slots, where messages are effectively transmitted. The sending slots are represented by the ID numbers.

In the static part all the sending slots have the same length, defined at pre-runtime, and are pre-assigned according to a TDMA strategy. Therefore, bus access is made without contention. Sending slots can be multiplexed,

allowing nodes to send different messages in different communication cycles. Thus, regarding the static part, the communication pattern can be described by a matrix.

In the dynamic part the bus arbitration is based on waiting times, using a mini-slotting scheme (CSMA/CA). Message IDs have associated a unique priority, and sending slots are assigned in decreasing order of priority, thus higher priority messages are sent first. Contrarily to the static part, in the dynamic part messages are only sent when required by the application. A timer is used to detect vacant slots and increment the slot counters in case of such event.

3.2.10 Fieldbus properties summary

Table **3.2.1** summarizes some of the properties of the several fieldbus systems above discussed.

Fieldbus	Scheduling paradigm [1]	Dynamic comm. req.	ET traffic	TT traffic	TT/ET isolation	Efficient ET handling
WorldFIP	ST+(DBE/SP)	No	Yes	Yes	Yes	-
FF-H1	DP+(DBE/SP)	Yes[2]	Yes	Yes	Yes	-/+
TTP/C	ST	No	No	Yes	*****	*****
TT-CAN	ST+(DBE/SP)	No	Yes	Yes	Yes	+
ProfiBus	DBE/SP	Yes	Yes	Yes[3]	No	-/+
P-Net	DBE/SP	Yes	Yes	Yes[3]	No	-/+
DeviceNet	DBE/SP	Yes	Yes	Yes[3]	No	+
FlexRay	ST+SP	Yes[4]	Yes	Yes	Yes	+

Legend:

1 ***ST***- *Static Table-Driven;* ***SP***- *Static Priorities-Driven;*
DBE- *Dynamic Best Effort;* ***DP***- *Dynamic Planning-Based*
XX+YY : XX for TT traffic and YY for ET;
(XX/YY) : XX or YY for ET traffic depending on pre-analysis.

2 assuming a dynamic scheduling profile, only ("N" for all other profiles)

3 Automatic Cyclic Transmissions

4 Concerning the event-triggered traffic only.

Table 3.2.1: Fieldbus properties summary

3.3 Ethernet-based RT protocols - brief survey

Apart from the protocols designed specifically to operate at the field level, a lot of effort was also devoted to the possibility of using general-purpose communication protocols employed in other areas (e.g. Ethernet, ATM, FDDI) at the field level. Several reasons have fostered this line of research [Dec01, BM01, Mon00, VC94], but the main arguments are that, on one hand, traditional fieldbuses have difficulties in supporting the growing bandwidth demand felt in some DCCS applications and, on other hand, pose interoperability difficulties when integrated in more complex systems composed by layered network architectures. Ethernet, in particular, has received recently a considerable interest from the scientific and industrial communities. For this reason, this section presents a brief reasoning about the use of Ethernet at field level and then visits some of the most relevant contributions in this area.

The first question that should be answered is what makes Ethernet so appealing to convey time-constrained traffic, considering that its designer has not envisaged this kind of applications. Thus, some properties of this protocol, such as the non-deterministic arbitration mechanism, pose serious challenges concerning its use at this level. Several works address this subject (e.g. [Dec01, VC94, BM01]). Particularly, [Dec01] presents a thorough reasoning on the pros and cons of using Ethernet at the field level in industrial systems, culminating with two concise sets of arguments, one in favor and the other against the adoption of Ethernet as a fieldbus.

Commonly referred arguments favoring the use of Ethernet in this field, can be summarized as follows:

- It is cheap, due to mass production;
- Integration with Internet is easy;
- TCP/IP stacks over Ethernet are widely available, allowing the use of application layer protocols such as FTP, HTTP and so on;
- Steady increases on the transmission speed have happened in the past, and are expected to occur in the near future;
- Due to its inherent compatibility with the communication protocols used at higher levels, the information exchange at plant level becomes

easier;

- The bandwidth made available by existing fieldbuses is insufficient to support some recent developments, like the use of multimedia (e.g. machine vision) at the field level;
- Availability of technicians familiar with this protocol;
- Wide availability of test equipment from different sources;
- Mature technology, well specified and with equipment available from many sources, without incompatibility issues;

On the other side, Ethernet does not fulfill some fundamental requirements that are expected from a communication protocol operating at the field level. In particular, the destructive and non-deterministic arbitration mechanism has been regarded as the main obstacle faced by Ethernet concerning this application domain. The common solution to this obstacle, nowadays, is the use of Switched Ethernet that allows to bypass the native CSMA/CD arbitration mechanism. In this case, provided that a single NIC is connected to each port, and the operation is full-duplex, no collisions occur.

However, just avoiding collisions does not make Ethernet deterministic: for example, if a burst of messages is sent to a single port of a switch at a rate larger than its transmission rate, its buffers can be exhausted and messages lost. Therefore, even with Switched Ethernet some kind of higher level coordination is required. Moreover, bounded transmission is not the only requirement in a fieldbus.

Other important requirements commonly referred to in the literature [Dec01, ISO94a] are: temporal consistency indication, precedence constraints, efficient handling of periodic and sporadic traffic. These are not all intrinsically supported neither by shared Ethernet nor by switched Ethernet.

In the quest for achieving real-time behavior on Ethernet several approaches and techniques have been used. The remainder of this section presents and characterizes some paradigmatic efforts, some of which are general and others have been developed specifically for Ethernet. Particular emphasis is given to the latter ones.

3.3.1 The Ethernet protocol

Ethernet was born about 30 years ago, invented by Bob Metcalfe at the Xerox's Palo Alto Research Center. Its initial purpose was to connect two products developed by Xerox: a PC and a brand new laser printer. Along the time this protocol has evolved in many ways and it has become the IEEE 802.3 standard. Despite the standard presenting some differences relatively to the original Ethernet specification, we will consider the IEEE standardized version, only. Therefore, in the scope of this thesis the term "Ethernet" always refers to the IEEE 802.3 standard, unless explicitly stated otherwise.

In terms of transmission speed, it has grown from the original 2.94Mbps to 10Mbps [IEE82, IEEb, IEEa, IEEf], then to 100Mbps [IEEd] and more recently to 1Gbps [IEEg]. Ten Gbps specification is expected to become available soon.

Concerning the physical medium and network topology, Ethernet also has evolved: it started with a bus topology based firstly on thick coaxial cable [IEEb] and afterward on thin coaxial cable [IEEa]. In the mid 80's a more structured and fault-tolerant approach, based on a star topology, was standardized [IEEe], running however at 1Mbps, only. In the beginning of the 90's an improvement on this latter technology was standardized [IEEf], running at 10Mbps over category 5 unshielded twisted pair cable.

Along this journey over the last three decades, two fundamental properties have been kept unchanged:

- a single collision domain, i.e., frames are broadcast on the physical medium and all the network interface cards (NIC) on it receive the message, and
- the arbitration mechanism, which is called Carrier Sense Multiple Access with Collision Detection (CSMA/CD).

According to the CSMA/CD mechanism, a NIC having a message to be transmitted must wait for the bus to become idle. When this happens, it starts the transmission. Since other NICs can also have messages ready for transmission, a collision can occur. In this case, all the stations that detect the collision abort the transmission of the current message and transmit a jam sequence, to ensure that all other adapters become aware of the occurrence of a collision. Next, the nodes wait for a certain time before retry

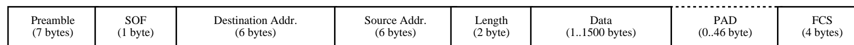


Figure 3.9: Ethernet frame

the message transmission. This waiting time is selected randomly from a discrete set of values. The upper bound of this set doubles its value by each consecutive collision (exponential back-off). After 10 collisions the upper bound of the waiting time interval does not grow anymore, which is the reason why the mechanism used by the Ethernet protocol is known as truncated exponential back-off. The number of retries is limited to sixteen.

The use of a single broadcast domain and the CSMA/CD arbitration mechanism has created a bottleneck in highly loaded networks: above a certain threshold, as the load increases the throughput of the bus decreases. A solution to this problem, known as thrashing, has been proposed in the beginning of the 90's, consisting on the use of switches in the place of hubs. A switch creates a single collision domain for each node connected to it. This way, collisions never occur unless they are intentionally created for managing purposes. Switches also keep track of the addresses of the NICs connected to each port, therefore each NIC only receives the traffic addressed to itself. This architecture allows the existence of multiple transmission paths simultaneously, between different network nodes. Since using switches the devices on the network no longer share the bandwidth and collisions don't occur, the throughput increases significantly.

Figure 3.9 presents the Ethernet frame format. Ethernet frames start with a preamble field meant to allow synchronization, followed by a start of frame (SOF) delimiter. Then the destination and source addresses are included, with 6 bytes each, to identify respectively the sender and recipient(s) of the message. The number of data bytes carried in the message is defined in the length field. The data itself is placed in the Data field, which can contain between 0 and 1500 bytes. To allow collision detection, the 10 Mbps Ethernet requires a minimum packet size of 64 bytes. So, shorter message must be padded with zeros (PAD field). Finally, the Ethernet frame ends up with a frame check sequence (FCS), meant for error detection. The FCS is performed on both address, length and data fields. The probability of undetected errors is 1 in $(2^{32} - 1)$ bits.

The IEEE controls the assignment of addresses by administering a por-

tion of the address field. The IEEE does this by providing 3 byte identifiers called "Organizationally Unique Identifiers" (OUIs), which are assigned to each manufacturer of Ethernet interfaces. The manufacturer in turn creates the full 6 byte addresses using the assigned OUI as the first 3 bytes of the address, and locally selecting the lower 3 bytes according to some internal policy. Provided that the locally assigned 3 bytes are unique, the full address becomes unique. This 6 byte address is also known as the physical address, hardware address, or MAC address, and is commonly pre-assigned to each Ethernet interface when it is manufactured.

As each Ethernet frame is sent onto the shared signal channel, all Ethernet interfaces look at the destination address field. The interfaces compare the destination address of the frame with their own address. The Ethernet interface with the same address as the destination address in the frame will read in the entire frame and deliver it to the networking software running on that computer. All other network interfaces will stop reading the frame when they discover that the destination address does not match their own address. This mechanism provides unicast communication.

A multicast address allows a single Ethernet frame to be received by a group of stations. Network software can set a station's Ethernet interface to listen for specific multicast addresses. This makes it possible for a set of stations to be assigned to a multicast group which has been given a specific multicast address. A single packet sent to the multicast address assigned to that group will then be received by all stations in that group.

There is also the special case of the multicast address known as the broadcast address, which has the 6 byte address filled with ones. All Ethernet interfaces that see a frame with this destination address will read the frame in and deliver it to the networking software.

3.3.2 Modification of the Medium Access Control

This approach consists on modifying the Ethernet MAC layer to achieve a bounded access time to the bus (e.g. [LR93, SS85, Cou92]). For instance, a method described in [LR93] (CSMA/DCR) consists in a binary tree search of colliding nodes, that is, there is a hierarchy of priorities. Whenever a collision happens the lower priority nodes voluntarily cease contending for the bus, and higher priority nodes try again. This process in repeated until a successful transmission occurs.

Two main drawbacks can be identified: in some cases the firmware must be modified, therefore the economy of scale obtained when using standard Ethernet hardware is lost; the worst-case transmission time, which is the main factor considered when designing real-time systems, can be orders of magnitude greater than the average transmission time. This forces any kind of analysis to be very pessimistic and thus, leads to an under-utilization of the bandwidth;

3.3.3 Addition of transmission control over Ethernet

Another way to achieve time-constrained communications over Ethernet consists in adding a layer above it, intended to control the instants of message transmissions, ending up with a bounded number of collisions or even a complete avoidance of them. The major advantage of this kind of approach, when compared to the modification of the MAC layer, is that standard Ethernet hardware can be used.

Several different ways of doing transmission control over Ethernet are referred below.

Master/Slave

In this case, all ordinary stations in the system transmit messages only upon receiving an explicit command message issued by one particular station called master. This approach supports relatively precise timeliness, depending on the master, but introduces a considerable protocol overhead caused by the master messages (notice the number of messages is duplicated). Also the time required by slaves to process the request and respective response (turnaround time) contributes to reduce the bus utilization efficiency. Moreover, with this approach the handling of event-triggered traffic is normally inefficient because the master must first become aware of any request before inquiring the respective station.

Token-passing

This method consists on circulating a token among the stations. Only the station currently holding the token is allowed to transmit and the token holding time is upper bounded (IEEE 802.4 timed-token is one example).

This scheme is still not very efficient due to the bandwidth used by the token and induces large jitter in the periodic traffic due to variations in the token holding time. Furthermore, token losses generally impose long periods of bus inaccessibility.

Timed Token

This particular technique is also based on token-passing and it is the basis for the RETHER protocol [VC94]. When in real-time mode, RETHER divides network nodes in two groups: the RT group for nodes with bandwidth reservations; the NRT group for all the others. The real-time messages are assumed to be periodic, and time is divided in cycles with the duration of one time unit. Access to the channel for both kinds of traffic is regulated by a token. First the token visits all the RT senders having messages to be produced in that cycle, and after the NRT nodes, if enough time is left until the end of the cycle.

An on-line admission mechanism is provided; only messages that can be timely handled and don't jeopardize the remaining RT set are accepted. The major drawbacks of this approach are: lack of support for real-time sporadic traffic; high overhead (similar to master/slave); lack of support for dynamic priorities concerning the periodic traffic;

TDMA

In this case, stations transmit messages at pre-determined disjoint instants in time in a cyclic fashion. This approach requires precise clock synchronization and does not lend itself well to dynamic changes in the message set because the communication requirements are distributed and thus, changes must be done globally. On the other hand, it uses the bus bandwidth efficiently since there are no control messages beyond those to achieve clock synchronization and also there is no need for explicit addressing.

Virtual Time Protocol

This protocol [MZ95, MK85] tries to reduce the number of collisions on the bus while offering the flexibility to implement different scheduling policies. It prioritizes messages by mapping different message parameters (e.g laxity or arrival time) in waiting periods, and operates in the following way.

When a node wishes to transmit a message, it waits for a given amount of time, counting from the moment the bus became idle. This amount of time is calculated according to the desired scheduling policy. When that time expires, and if the bus is still idle, the node tries to transmit the message. Collisions can still occur since there is no guarantee that two different nodes can have messages with the same priority. In this case the protocol uses a probabilistic approach, in which the nodes involved in the collision retransmit the message with a given probability p.

This kind of approach has some important drawbacks:

- Performance is highly dependent on the proportional constant value used to relate the waiting time with the scheduling policy in use, which leads to collisions if this factor it is too short, and to a large amount of idle time (low efficiency in bandwidth utilization), if the proportional constant is too long;
- Proportional constant is dependent on the properties of the message set, therefore on-line changes can lead to poor performance;
- Lack of support for time-triggered traffic;
- The unbounded worst-case transmission time, resulting from the probabilistic collision resolution mechanism, renders this protocol unsuitable for use in hard real-time systems.

One of the most interesting features of this approach it is its ability to achieve performances close to the theoretical model for some scheduling policies. For instance, in [ZR87] it is shown that the Virtual Time protocol performs close to the exact minimum laxity first policy under a wide range of load conditions.

Windows Protocols

This type of protocols has been proposed both for CSMA/CD and token ring networks [MZ95]. Concerning the CSMA/CD implementation, the operation is as follows. The nodes on a network agree on common predefined time interval named window, and the bus state is used to assess the number of nodes with messages to be transmitted within the time window. If only one message is ready within in the window, it will be transmitted. However;

if more than one node has ready messages within the window, a collision occurs. In this case the window size is successively reduced until only one message is in the window. Finally, if no nodes have ready messages within the window, then the window size is increased.

This method has some important drawbacks:

- The time and space required to maintain the window can incur is substantial overhead [MZ95];
- Lack of explicit support for time-triggered traffic;
- Since collisions make part of the protocol, worst-case transmission time is much higher than average transmission time, leading to bus underutilization when timeliness must be guaranteed (i.e. for hard real-time systems).

On the positive side, this approach, unlike priority-based protocols, is not limited by the number of available priority levels.

Traffic shaping

As opposed to transmission control, this technique follows an approach based on the fact that, if the bus utilization is kept low, then the probability of collisions is also low (although not zero). Therefore, if the network average load is kept below a given threshold and traffic bursts are avoided, a given probability of collisions can be obtained. Implementations of this technique are presented in [KSZ99, KS00, BM01, CCBM02]. An interface layer called traffic smoother is placed between the transport layer (TCP/UDP) and Ethernet. Real-time traffic is assumed to be event-triggered and generated pseudo-periodically, since it is generated by some kind of control system. Moreover, the real-time traffic is assumed to use a small fraction of the bus bandwidth and is transmitted on demand, without interference of the traffic smoother. Non-real-time (NRT) traffic can be bursty and it is handled by the traffic smoother, which keeps track of previous message transmissions (both RT and NRT) performed by the node. According to this history record, the traffic smoother releases NRT messages in a controlled fashion, in order to follow a desired node's traffic generation rate. This way, at the network level, the interference on the RT traffic due to NRT traffic is kept inside a

(probabilistic) bound. Several techniques have been developed to manage the behavior of the traffic smoother, such as the leaky bucket proposed by *Kweon et al* [KS00] and fuzzy logic in *Carpenzano et al* [CCBM02].

One major drawback of this approach is that all the guarantees are statistical - it cannot be guaranteed a priori that a specific message can be transmitted within a specific time interval. Therefore this approach is not well suited to support hard real-time traffic. Moreover, this approach lacks explicit support for time-triggered traffic.

Switched Ethernet

The use of switches became very popular recently, as a way to improve the performance of shared Ethernet. Switches provide a private collision domain for each one of their ports, i.e., unlike a hub, there is no "direct" connection between the ports. When a node transmits a message, this one is received by the switch and then buffered in to the ports where the receivers of the message are connected. If several messages addressed to a given port arrive in a short interval, they are buffered and then sequentially transmitted. The IEEE 802.1D standard defines 8 priority queues in output ports. The scheduling policy used at this level is a topic currently addressed in the scientific community (e.g. [JN01]).

Unfortunately the use of a switch in an Ethernet network is not enough to make it real-time, in the general case. For instance, output buffers can be exhausted and messages lost if bursts of messages are sent to the same output port. This situation can occur more often than desired, even in the field of distributed control systems. In this kind of systems the producer/consumers model is frequently used. According to this co-operation model the producer of a given datum (e.g. a sensor reading) sends it to several consumers of that datum. This model is efficiently supported in Ethernet by means of special addresses, called multicast addresses. Each network interface card can define a local table with the multicast addresses related to the data that it should receive. However, the switch has no knowledge of these local tables, therefore treats all the multicast traffic as broadcasts, i.e., messages with multicast destination addresses are transmitted to all ports. Therefore, depending on the predominant type of traffic exchanged in a given application (unicast vs. multicast/broadcast), one of the main benefits of using Switched Ethernet, multiple simultaneous transmission paths, can be seriously compromised.

Other problems concerning the use of switched Ethernet are [Dec01]:

- In the absence of collisions the switch introduces an additional latency;
- The number of available priority levels is too small to support the implementation of efficient priority based scheduling;
- The switch only makes Ethernet deterministic under controlled loads.

3.3.4 Ethernet-based protocols properties summary

Table **3.3.1** summarizes some of the properties of the several Ethernet-based protocols above discussed.

Protocol	Traffic classes			Dynam. Comm. Req.	Time. Guar-anties	Temp. Isolat.	Effici-ency	COTS Hard-ware
	Real-time		Non Real Time					
	Time Trig	Event Trig						
CSMA/DCR	No	Yes	Yes	Yes	Hard[1]	No	Low[2]	No[5]
TDMA	Yes	No	No	No	Hard	N.A.	High	Yes
Virtual time	No	Yes	Yes	Yes	Hard[1]	No	Low[2]	Yes
Windows	No	Yes	Yes	Yes	Hard[1]	No	Low[2]	Yes
Time-token	Yes	No	Yes	Yes	Hard	Yes	Low[3]	Yes
Switch	No	Yes	Yes	Yes	No[4]	No	High	Yes
Traffic Smoothing	No	Yes	Yes	Yes	Soft	No	Low[2]	Yes

Legend:

1 *Worst-case response time much higher than the average value*

2 *Collisions are part of the protocol*

3 *Each real-time message is preceded by a control message*

4 *Can be achieved by the use of admission control (not part of the protocol)*

5 *Requires modifications to the NIC's firmware*

N.A. *Not applicable*

Table 3.3.1: Ethernet-based protocols properties summary

3.4 Conclusion

This chapter starts by a brief presentation of distributed real-time systems, with particular emphasis in issues like co-operation models, message scheduling and message triggering paradigms. Further on it presents a survey on some of the most representative protocols that have been developed to support such kind of distributed systems.

Many real-world systems are complex and dynamic, evolving during time and require, or at least benefit, from the presence of a flexibility real-time communication network. For this reason, the protocols analyzed in this chapter have been assessed in what concerns issues like:

- Support for different traffic classes with distinct timeliness requirements;
- Support for dynamic changes on the message properties;
- Support for different message triggering models (time and event-triggered);
- Temporal isolation between the different types of traffic;
- Efficiency in bus bandwidth utilization;

The results are summarize in Table 3.2.1 concerning fieldbus protocols and in Table 3.3.1 concerning Ethernet-based protocols. From the observation of these tables, it can be concluded that none of the analyzed protocols fulfills all the properties referred above. Therefore, applications demanding flexible communication systems do not find adequate support in these protocols. In the following section it will be presented a new communication paradigm, the Flexible Time-Triggered paradigm, that aims at filling this gap.

Chapter 4

The FTT paradigm

The requirement for flexibility is becoming increasingly important in distributed computer-controlled systems motivated by the need to reduce the costs of set-up, configuration changes and maintenance [S^+96, Tho98]. This requirement extends to all system levels including the field level in process industries and the cell and machine control levels in manufacturing industries, where fieldbus-based distributed computer control systems can be found.

Moreover, recent applications such as agile manufacturing, real-time databases, automotive, mobile robotics and machine vision must deal with environments that are inherently dynamic. This type of applications are not easily or efficiently supported by "open loop" scheduling algorithms [SLST99], i.e., algorithms in which once the schedules are created they are not "adjusted" based on continuous feedback about the system evolution. While open-loop scheduling algorithms can perform well in static or dynamic systems in which the workloads can be accurately modeled, they can perform poorly in dynamic systems, where such degree of knowledge is hard to find or even non-existent. A possible methodology to support this type of requirements consists in regarding the computer system as a control system with the scheduler as the controller, and integrate practical feedback control techniques into scheduling algorithms [SLST99]. To support such framework efficiently, the real-time communication system should support on-line changes to the communication requirements, to reflect the evolving requirements, but nevertheless keeping timeliness and predictability guarantees.

This chapter presents a reasoning about the requirements posed to the communication system in the framework of flexible real-time distributed

computer control systems, culminating with a concise set of properties that must be fulfilled. Then, a new communication paradigm, the Flexible Time-Triggered (FTT) architecture, which supports these requirements, is presented.

4.1 Why a new protocol

Concerning specifically the fieldbus system, flexibility implies dynamic communication requirements, meaning that on-line addition, removal and adaptation of message streams must be supported. On the other hand, most of the data exchanges handled by the fieldbus are also subject to stringent timing constraints arising from control and monitoring requirements. Unfortunately, flexibility and timeliness have typically been considered separately and most of the fieldbuses available today favor either one aspect or the other [Tho98], i.e., either time-constrained services are guaranteed sacrificing flexibility or such guarantees are sacrificed in exchange for higher flexibility.

Another requirement typically found in fieldbus systems is the capacity to deliver both time and event-triggered communication services under timing constraints. The former ones are well suited to convey periodic updates of state data whilst the latter ones are more adapted to convey alarms and management data. Again, existing fieldbus systems privilege either one or the other type of services. In systems eminently time-triggered, event-triggered services are either non-existing or handled inefficiently in terms of either response time or network utilization. On the other hand, in systems eminently event-triggered, interesting properties of time-triggered services such as composability with respect to the temporal behavior are normally lost [Kop93].

The requirement for flexibility is sometimes considered as conflicting with the time-triggered approach [Kop97, KG94], since, according to this model, communication activities occur at pre-defined instants in time. However, time-triggered systems also may profit from flexibility [BA00, Mar02]. To achieve such behavior, the time-triggered traffic should be scheduled on-line, with the scheduler basing its decisions on the actual communication requirements. However, such flexibility should not compromise the system timeliness and predictability, and thus such flexible real-time systems should incorporate admission control, as discussed in Section 2.3.

Different scheduling policies provide schedules that exhibit different properties and imply different computational costs (Section 2.4). Therefore, a communication system with the capacity to support distinct scheduling policies can be adapted to different platforms and applications. For instance, in platforms with low processing power available, it can be used a fixed priority (e.g. RM) instead of dynamic priority (e.g. EDF) based scheduling policy, lowering the scheduling overhead at expenses of a potentially lower utilization of the communicational channel. Furthermore, in some circumstances it can be important to have the possibility to change on-line the scheduling policy of a given system. For example, during normal operation a system could be scheduled by EDF to maximize the communication channel utilization efficiency. However, upon a degradation in the communicational channel performance (e.g. due to electromagnetic interference), the transmission of the most important messages should be privileged. This behavior can be achieved by switching to a fixed-priority value-based scheduling policy.

Frequently real-time entities have a limited lifetime. For example, in distributed control systems one or more nodes execute control algorithms based on sensor data generated elsewhere. The communication between sensor and controller nodes is performed exclusively through the communication network. If, due to some problem, a sensor node fails in transmitting the value of an environment variable, the control algorithms may not be fed with correct inputs. In this case the controller nodes should be informed of the failure, to enable them to take some corrective actions, whenever possible and desired. Hence, the network protocol itself should provide services to know if the data values are still in accordance with the corresponding environment variables, which is a property designated by temporal consistency [Dec01] or accuracy [Kop97].

Another issue is related with the cooperation model. In many applications the same data is required in different network nodes. This requirement is efficiently supported by the producer-consumer co-operation model (Section 3.1.4). However, to provide this co-operation model efficiently, the communication system should have intrinsic support of multicast services, i.e., a single data message transmission should reach all consumer nodes.

To comply with all of these requirements, adequate choices of communication paradigms and protocols are needed. More specifically a protocol able to handle such flexibility requirements must support:

- Time-triggered communication with operational flexibility;
- Support for on-the-fly changes both on the message set and on the scheduling policy used;
- On-line admission control to guarantee timeliness to the real-time traffic;
- Indication of temporal accuracy of real-time messages;
- Support of different types of traffic: event-triggered, time-triggered, hard real-time, soft real-time and non-real-time;
- Temporal isolation: the distinct types of traffic must not disturb each other;
- Efficient use of network bandwidth;
- Efficient support of multicast messages;

As presented in Section 3.2, none of the existing fieldbus protocols fulfills all of these requirements. For instance, concerning the support of event and time-triggered traffic, existing protocols either do not support both types of traffic (e.g. TTP/C), or both types are supported but without temporal isolation (e.g. Profibus, P-Net, DeviceNet). In the cases where temporal isolation is enforced, the event-triggered traffic is handled inefficiently (e.g. WorldFIP, Foundation Fieldbus-H1), and/or the time-triggered traffic is specified statically, thus not supporting operational flexibility concerning the time-triggered traffic (e.g. TT-CAN, FlexRay). The same situation happens with the Ethernet-based protocols analyzed in Section 3.3. The Flexible Time-Triggered paradigm herein presented addresses these issues and fulfills the requirements for flexibility, timeliness and efficient combination of time and event-triggered traffic.

4.2 The Flexible Time-Triggered paradigm

The Flexible Time-Triggered (FTT) paradigm has its roots in the FTT-CAN protocol [AFF98, APF99], originally developed within the Electronic Systems Laboratory in the University of Aveiro. The FTT-CAN protocol is based on Controller Area Network, and aims to provide support for the

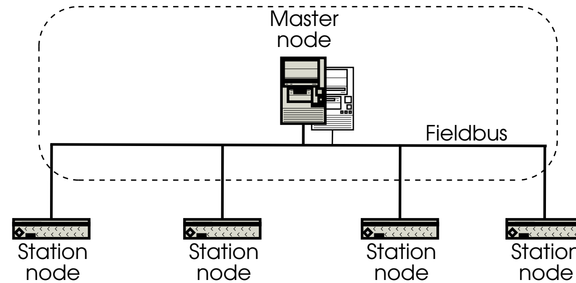


Figure 4.1: The FTT paradigm system architecture

combination of both time and event-triggered traffic with temporal isolation and operational flexibility concerning the time-triggered traffic. Its main target is low processing-power micro-controllers, used in embedded distributed real-time applications. During the development of the FTT-CAN protocol it was realized that the main concepts could be abstracted to form the Flexible Time-Triggered paradigm, a general communication paradigm, which, in its turn, could have implementations using other communication infrastructures.

The FTT paradigm defines the system architecture and application programming interface (API) as seen from the application software. Each of the FTT implementations has its peculiarities, such as bit-rate, admissible message lengths, addressing schemes, etc. However, these characteristics are abstracted, and the paradigm exhibits a common set of properties, which are independent of the particular implementation. The envisaged target systems range from low processing-power micro-controllers, like the 8051, used typically in embedded industrial control systems, to high performance systems, able to handle complex activities, such as computer-vision and autonomous mobile robot control.

4.2.1 System architecture

The FTT paradigm presents an asymmetric architecture, comprising one master node, possibly replicated for fault-tolerance reasons, and one or more station nodes (Figure 4.1). The master node is responsible for the management and coordination of the communication activities, and the station nodes execute the application software as well as the network protocol.

The master node implements the **centralized scheduling** concept, in

which both the communication requirements, message scheduling policy and on-line admission control are localized in one single node. Such concentration of functions allows to have at any instant complete knowledge of current system requirements and also the possibility to make atomic changes to any of them. Moreover, such architecture also facilitates the implementation of on-line admission control with fast response.

The distribution of the scheduling decisions to the network stations is periodically performed by the master through a special control message, the trigger message (TM). Thus, concerning the coordination of the communication activities, a master-slave relation is established between the master and the stations. To reduce the efficiency penalty usually associated to master-slave communication, the FTT paradigm uses a relaxed master-slave approach, designated **master/multi-slave transmission control**, in which a single trigger message causes the transmission of several slave messages, eventually originated in distinct station nodes. This method reduces the number of control messages, consequently improving the bandwidth utilization, and, at the same time, benefits from the timeliness properties associated to master-slave communication.

By using centralized scheduling and consistent interfaces between the scheduler, dispatcher, admission control manager and requirements manager, together with the distribution of the schedule decisions by means of the trigger message, the system gets a high degree of flexibility since:

- The station nodes on the network are not aware of the particular scheduling policy in use, since they strictly follow the traffic schedule conveyed in the trigger message. Therefore any scheduling policy can be implemented, irrespectively of its complexity and nature (e.g, fixed priorities, dynamic priorities), provided the master has enough processing power to timely compute and distribute the schedule.

- Several scheduler modules can be implemented, and the system can change between them "on-the-fly", autonomously or on demand. For example, the system can be configured to use Earliest Deadline First (EDF) scheduling in order to maximize the utilization factor under normal system operation, and switch to some kind of value-based fixed priorities scheduling on overloads, in order to guarantee that most important messages are scheduled within their deadlines.

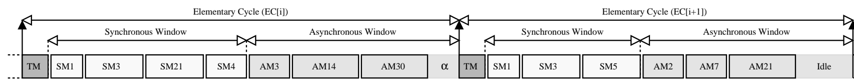


Figure 4.2: The Elementary Cycle structure

- All the required scheduling information is carried on the trigger message, therefore, when changing message properties (e.g. its periodicity), the synchronization of the update throughout the network is intrinsically guaranteed.

- The master holds enough information to know the demands of real-time traffic and how much leeway the system has, therefore it can safely allocate bus bandwidth to other kinds of traffic without jeopardizing the timeliness of real-time traffic.

4.2.2 The Elementary Cycle

In the FTT paradigm the bus time is slotted in consecutive fixed duration time-slots, called Elementary Cycles (ECs). The EC starts with the reception of the TM, and all nodes are synchronized by its reception. Each EC is composed by two consecutive windows, synchronous and asynchronous, that correspond to two separate phases (Figure 4.2).

The synchronous window conveys the time-triggered traffic, according to the contents of the trigger message. The length of the synchronous window ($lsw(i)$) can vary from EC to EC, according to the number and length of messages scheduled for that particular EC. It is however possible to impose a limit to the maximum size of the synchronous window (LSW), and thus grant to the asynchronous window a minimum guaranteed bandwidth share. The time-triggered traffic is subject to admission control and thus all messages accepted by the system have their timeliness guaranteed (dynamic planning-based scheduling).

The asynchronous window has a duration ($law(i)$) equal to the time gap between the EC trigger message and the synchronous window. It is used to convey event-triggered traffic, herein called asynchronous because the respective transmission requests can be issued at any instant. Unlike the synchronous traffic, the arbitration within the asynchronous window is not resolved by the master node. The only information supplied in the trigger

message related with the asynchronous window is their duration. A suitable protocol must then be used to perform the message arbitration within this window. The asynchronous traffic is handled in a best-effort policy. However, the use of deterministic medium-access policies combined with the possibility to define a minimum guaranteed bandwidth to the asynchronous traffic allows, when required by the application, to pre-analyze its requirements and compute if a given set of real-time asynchronous messages can meet their deadlines in worst-case conditions. This feature is usually required only by asynchronous messages related to alarms or other similar real-time events. In general, the asynchronous window is mainly devoted to non-real-time traffic, such as software upload/download, remote diagnostics and configuration, remote calibration, etc., with relaxed real-time requirements or even no real-time requirements at all.

In order to maintain the temporal properties of the time-triggered traffic, such as composability with respect to the temporal behavior, the synchronous window must be protected from the interference of asynchronous requests. A strict temporal isolation between both phases is enforced by preventing the start of transmissions that could not complete within the respective window. Since the message lengths are not correlated nor with the EC duration neither with the synchronous and asynchronous window durations, a short amount of idle-time (α) may appear at the end of the asynchronous window (exclusion window).

The FTT paradigm does not specify the relative order of the synchronous and asynchronous windows. This aspect is only defined by specific protocol implementation. The justification for this procedure is that particular implementations can profit form a particular window arrangement (e.g. [PA00]).

The communication services of the FTT paradigm are delivered to the application by means of two subsystems, the Synchronous Messaging System (SMS) and the Asynchronous Messaging System (AMS), that manage the respective type of traffic. The SMS offers services based on the producer-consumer model [TC99] whilst the AMS offers send and receive basic services, only. The components of each of these services are spread among the master and the station nodes, and presented in the following sections.

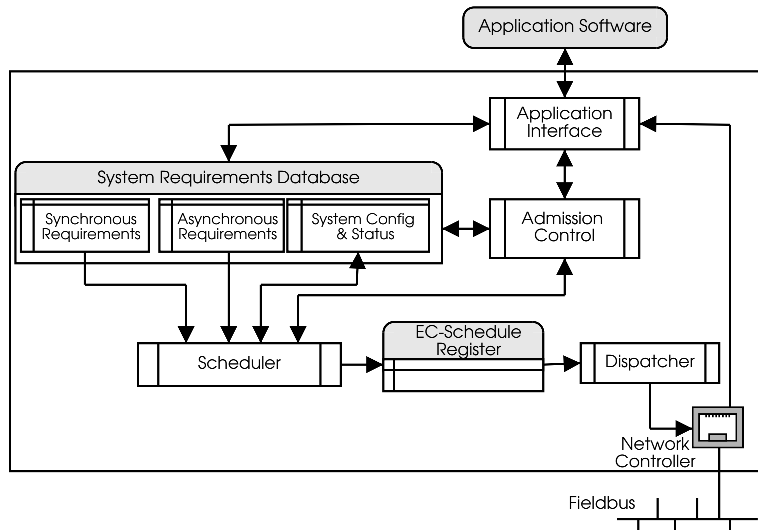


Figure 4.3: FTT master internal architecture

4.2.3 Master node architecture

The master node plays the role of system coordinator and it is responsible for providing an interface to allow system management, maintaining a local database holding the system communication requirements, building schedules generated according to the particular scheduling policy implemented and broadcasting these schedules at appropriate time instants. Figure 4.3 depicts the internal architecture of an FTT master.

The **Application Interface** provides a set of services that are used by the application software to perform the system configuration. All the interaction with the application software is made through this interface. These services are available both locally and remotely, via the network. The following classes of services are available:

- System configuration and management: set-up of the EC duration, bus speed, network topology and overheads (e.g. guard bands, message processing);
- Message management: addition and exclusion of messages, as well as modification of their properties;
- System Status Record access: retrieve information about system performance, like jitter figures, bandwidth use for each traffic class.

The **System Requirements Database** (SRDB) holds the properties of each of the message streams to be conveyed by the system, both real-time and non-real-time, as well as a set of operational parameters related to system configuration and status. This information is stored in a set of three tables.

The **Synchronous Requirements Table** (SRT) includes the properties of the synchronous messages conveyed by the system (Definition 4.1).

$$SRT \equiv \{SM_i(DLC_i, C_i, Ph_i, P_i, D_i, Pr_i, *Xf_i),\ i = 1..N_S\} \tag{4.1}$$

where for each message SM_i of a set of N_S synchronous messages, DLC_i is the data length in bytes, C_i is the respective transmission time (including all overheads), Ph_i allows to define an initial phase, P_i is the period or minimum inter-arrival time, respectively for periodic and sporadic messages, D_i is the deadline and, finally, Pr_i is a fixed priority. The basic time unit in the FTT paradigm is the elementary cycle duration, thus both Ph, P and D are expressed as integer multiples of the EC duration (E). Synchronous message exchange is based on the producer-consumer co-operation model, therefore it uses source addressing, i.e., the message identification is related to the message contents and not with the particular sender or consumer(s).

Besides the basic properties above defined, the SRT also supports an additional optional field (Xf) that can be used by particular scheduling algorithms that require other types of information. For instance, if it is required to support message streams with different levels of acceptable Quality of Service (QoS) concerning the respective bandwidth, the SRT can be extended with an admissible period range (Minimum, Nominal and Maximum). On the other hand, this mechanism also allows to restrict the operations allowed on the message stream attributes. For example, some flags can be used to indicate which messages can or cannot be removed or if the QoS manager can automatically update their properties.

The **Asynchronous Requirements** component is composed by the reunion of two tables, the Asynchronous Requirements Table (ART) and the Non-Real-Time Requirements Table (NRT).

The ART (Definition 4.2) is used to store the properties of the asynchronous messages conveyed by the system that, despite being asynchronous, may or may not have timeliness requirements. For example alarm messages

usually have hard timeliness requirements while messages used to perform remote diagnosis or configuration frequently do not have such timeliness constraints. The asynchronous messages are scheduled according to a best-effort policy, based on fixed priorities. Nevertheless, it is possible to pre-analyze the communication requirements in order to verify if a given subset of asynchronous message set, having timeliness requirements, can be scheduled by the system within their deadlines, in all anticipated load conditions.

$$ART \equiv \{AM_i(DLC_i, C_i, mit_i, D_i, Pr_i),\ i = 1..N_A\} \tag{4.2}$$

This table is similar to Definition 4.1 except for the use of mit_i, minimum inter-arrival time, instead of period, and the absence of initial phase Ph_i, since asynchronous messages are triggered by the application software at any instant, without phase control. As in the case of the synchronous messages, the asynchronous message exchange is based on the producer-consumer cooperation model, therefore it uses also source addressing.

The non-real-time traffic is handled strictly according to a best-effort policy. Since no timeliness guarantees are provided, the master node only needs to keep track of which stations produce this kind of traffic, and, for each of them, the size of the respective longest non-real-time message, as required to enforce the temporal isolation between synchronous and asynchronous traffic.

$$NRT \equiv \{NM_i(SID_i, MAX_DLC_i, MAX_C_i, Pr_i),\ i = 1..N_N\} \tag{4.3}$$

The NRT contents is defined by Definition 4.3, where SID_i is the node's identifier , MAX_DLC_i is data length in bytes of the longest non-real-time message produced by the node, MAX_C is the respective maximum transmission time, including all overheads, and Pr_i is the node's non-real-time priority, which can be used to implement an asymmetric distribution of the bus bandwidth among the different nodes. Finally, N_N is the number of stations producing non-real-time messages.

The last component of the System Requirements Database is the **System Configuration and Status Record** (SCSR). This record stores all system configuration data, such as the bus transmission speed, duration of the elementary cycle, minimum amount of bandwidth allocated to asynchronous

traffic, protocol overheads dependent on the network topology (e.g. network length and number of repeaters), etc. Moreover, the scheduler also stores in this record data concerning traffic figures, such as the bandwidth used by each traffic class. This information is made available to the application layer, therefore it can be used either in preliminary field tests for profiling purposes or at run-time to improve the system adaptability (e.g. changing the scheduling policy or message properties depending on some thresholds), raise alarms when some figures override specific thresholds, etc.

The **Scheduler** uses the information provided by the SRDB to build the EC-Schedules for the synchronous traffic. More specifically, the Scheduler reads the message properties of the both synchronous and asynchronous messages, as well as the system configuration information stored in the SCSR register, and, based on such data, decides which synchronous messages should be transmitted in the following EC, according to the particular scheduling algorithm implemented. The result of such computation is placed in the EC-Schedule register (ECSR).

The Scheduler also gathers information about the scheduled messages and update the SCSR status record accordingly. The data placed by the Scheduler in the EC-Schedule register explicitly defines the IDs of the messages that shall be transmitted, as well as the duration of the synchronous window. However, particular implementations can require additional information. For example, in implementations based on shared Ethernet or RS-485 the message transmission must be performed in exclusive time slots to avoid collisions, thus information about the specific message transmission time of each message must also be placed in the ECSR.

The **Admission Control** is based on the schedulability test of the synchronous traffic. The schedulability test must consider not only the message properties but also other relevant information like the maximum length of the synchronous window or which particular scheduling algorithm is being used. The admission control is invoked whenever there is a request for a change in the SRT. Changes are accepted only when the schedulability test result indicates that the system timeliness is not jeopardized. In any case the application interface is notified about the result of the change request.

Both the Scheduler and the Admission Control are encapsulated in modules with clearly defined interfaces. The system supports a seamless integration of several different modules that can be switched on-line, according to

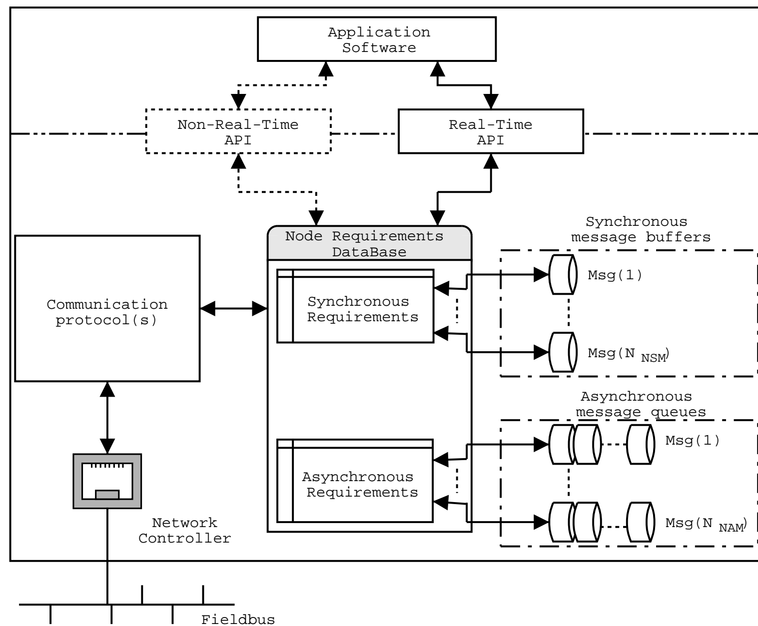


Figure 4.4: FTT station internal architecture

some triggering event, as referred above.

The **Dispatcher** reads the EC-Schedule Register, builds the next trigger message with such EC schedule and broadcasts it over the network. Since it is the reception of the trigger message in the remaining nodes that signals the beginning of an EC, it is important to schedule the Dispatcher task regularly, with sufficient precision.

4.2.4 Station node architecture

Station nodes, also known as ordinary or slave nodes, execute the application software required by the user, eventually requesting the services delivered by the communication system. The station node's internal architecture is depicted in Figure 4.4.

The application software interacts with the communication system trough a real-time API (RT_API) which enables the applications to:

- Define which messages are locally produced or consumed;
- Update the value of such real-time entities;
- Get the value of such real-time entities;

- Set-up callbacks associated to communication events such as message transmission and reception, as well as error conditions such as deadline misses;

Moreover, the FTT architecture also provides support for the integration of foreign communication protocols. This traffic is included in the NRT class, and thus it is intercepted by the FTT communication stack and transmitted within the asynchronous window, after explicit permission of the master node. This way the timeliness of the FTT real-time traffic is not jeopardized by the presence of traffic belonging to other protocols. The access to this communication stack is made trough its native application layer interface, which is denoted as Non-Real-Time API in Figure 4.4.

The Node Requirements Database

The Node Requirements Database (NRDB) holds the node's communication requirements, and is composed by two components, the Synchronous Requirements component and the Asynchronous Requirements component.

The exchange of synchronous messages is performed with autonomous control, i.e. the transmission and reception of messages is carried out exclusively by the network interface without any intervention from the application software. The message data is passed to and from the network by means of shared buffers. This means that the network interface, in what concerns the synchronous messages, behaves as a temporal firewall between the application and the network, since it isolates the temporal behavior of both parts, increasing the system robustness. There are two complementary API functions available to the application layer, *SMS_produce* and *SMS_consume*, which allow respectively producer nodes to update the local buffer with new data and consumer nodes to read the actual contents of the local buffer.

The information about each of the synchronous messages (N_{NS}) is stored in the NRDB's Synchronous Requirements Table (N_SRT), and consists of (Definition 4.4) the respective data length (DLC_i), the indication if it is a message locally produced or consumed (P_C_i), timer field to manage time validity information (Tmr_i), address of tasks associated with communication events, namely transmission (Tx_ev_i), reception (Rx_ev_i) and deadline miss (DM_ev_i).

$$N_SRT \equiv \{N_SM_i(DLC_i, P_C_i, Tmr_i, Tx_ev_i, Rx_ev_i,$$
$$DM_ev_i, Dbuff_i),\ i = 1..N_{NS}\} \qquad (4.4)$$

The N_SRT table also holds a pointer to the data buffer ($Dbuff_i$) used to store the data itself. It should be noted that there is concurrency in the access to the data buffer between the RT_API and the communication stack software. Moreover the use of basic mutual exclusion methods, such as semaphores, must be avoided because the communication software cannot be delayed when it is time to transmit a message. Therefore methodologies like double-buffering or Cyclic Asynchronous Buffers ([But97]), which allow multiple access, should be used. Alternatively it can also be used a single buffer, if there is an indication about the message validity in message frame, together with a suitable integrity verification function performed both in sender and receiver nodes.

An optional field can be appended to the table to store other relevant information, such as the number of messages received and transmitted, number of deadlines missed, jitter, lateness, message group definition, etc.

The transmission of the real-time asynchronous messages follows the external control paradigm, i.e. the transmission of messages takes place upon explicit requests from the application software. Such requests are issued by means of a basic API service called *AMS_send*, which is a non-blocking send function with queuing. The queue is ordered first by priority, according to the message identifiers, and second by request instant (FCFS). The length of each asynchronous message queue is set at configuration time and defines the maximum number of messages that can be queued at the same time. This is particularly relevant when the minimum inter-arrival time of transmission requests in a given stream is shorter that the worst-case time to process a single request of that stream.

The delivery of messages to the application software is accomplished by means of a complementary API basic service called *AMS_receive*, a receive function that allows waiting for a specified, or unspecified message. At the receiving node, the AMS also queues the messages arriving from the network until they are retrieved with the *AMS_receive* service. The length of the queue is also set-up at configuration time, similarly to the queue in the

sender side. In this case, the important aspect is the time the application takes to process each message.

More complex and reliable exchanges, e.g. requiring acknowledge or requesting data, must be implemented at the application level, using the two basic services referred above.

$$N_ART \equiv \{N_AM_i(DLC_i, P_C_i, Tmr_i, Tx_ev_i, Rx_ev_i, DM_ev_i, Dqueue_i),\ i = 1..N_{NA}\} \tag{4.5}$$

The information about each of the asynchronous real-time messages (N_{NA}) sent or received by the node is stored in the NRDB's Asynchronous Requirements Table (N_ART) (Definition 4.5), and consists of the respective data length (DLC_i), the indication if it is a message locally produced or consumed (P_C_i), timer field to manage time validity information (Tmr_i), address of tasks associated with communication events, namely transmission (Tx_ev_i), reception (Rx_ev_i) and deadline miss (DM_ev_i), and finally a pointer to the queue holding the messages waiting to be transmitted or already received but waiting to be read by the application, respectively if the node is a sender or a receiver of the particular message stream ($Dqueue_i$).

$$N_NRT \equiv \{N_NM_i(SID_i, MAX_DLC_i, P_C_i, Prot_i, Tx_ev_i, Rx_ev_i, DM_ev_i, Dqueue_i, DqueueFP_i),\ i = 1..N_{NN}\} \tag{4.6}$$

Non-real-time asynchronous message transmission is performed only after an explicit pol by the master node. The information about each of the non-real-time messages (N_{NN}) sent or received by the node is stored in the NRDB's Non-Real-Time Requirements Table (N_NRT) (Definition 4.6), and consists of the identification of the sender node (SID_i), the respective maximum data length (MAX_DLC_i), the indication if it is a message locally produced or consumed (P_C_i), the indication if it is an FTT message or a foreign protocol message ($Prot_i$), address of tasks associated with communication events, namely transmission (Tx_ev_i), reception (Rx_ev_i) and deadline miss (DM_ev_i), and finally a pointer to the queue holding the messages waiting to be transmitted or already received but waiting to be read

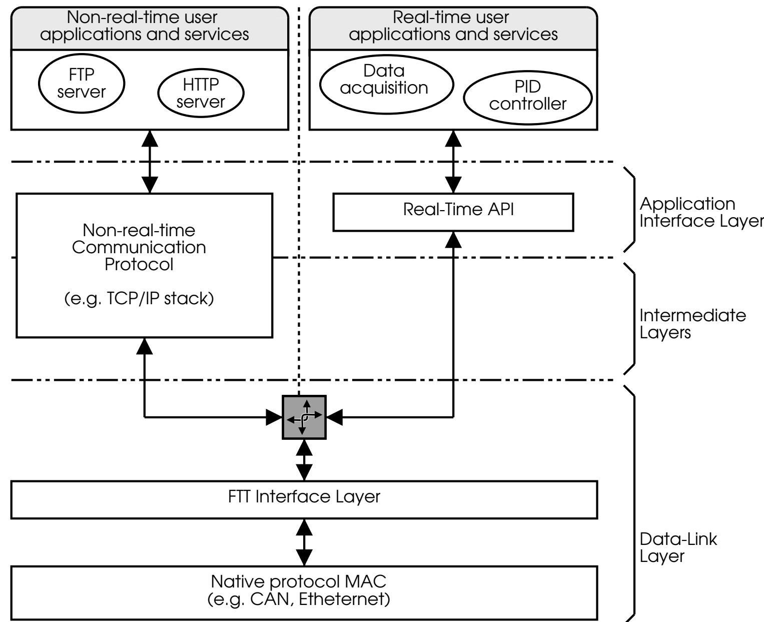


Figure 4.5: FTT station network software architecture

by the application, respectively if the node is a sender or a receiver of the particular message stream ($Dqueue_i$).

The *Prot* provides support to the possibility of exchange messages from other protocols within an FTT system. If the *Prot* field is set to non-FTT, the P_C field is ignored, since it is not performed any filtering concerning this kind of traffic. Moreover, in this case there are allocated two message queues, $Dqueue_i$ and $DqueueFP_i$, used respectively for message transmission and message reception.

Communication stacks

The access to the communication medium is performed trough an adequate communication protocol. Two parallel stacks can be used, one for real-time and the other for non-real-time communication, as depicted in Figure 4.5.

The non-real-time protocol stack provide the means to allow FTT to co-exist with other protocols. For instance, in the FTT-Ethernet protocol, this mechanism is used to allow the exchange of TCP/IP messages among system nodes, thus supporting standard applications and protocols such as FTP, HTTP and others to execute in system nodes. This aspect is par-

ticularly interesting, since there is a strong pressure for supporting the use of standard tools, namely web-based, to perform device management and monitoring as well as to facilitate the interoperability among the different plant levels [MAR+00, Wol00]. Different techniques can be used to implement this mechanism, such as encapsulation of foreign-protocol frames within FTT frames, procedure commonly known as tunneling. In other cases, such as with Ethernet, the native data frame already incorporates a control field that supports protocol multiplexing, thus in this case switching among the stacks can be performed just by handling the respective frame type tag.

The real-time protocol stack follows the collapsed 3 layers OSI reference model typically found in fieldbus systems. It provides a specific application interface, the Real-Time Application Programming Interface (RT_API),

The data-link layer (DLL) of the native communication protocol is modified, with the addition of a transmission control layer, both for real-time and non-real-time communication. This is referred to as the FTT Interface Layer (Figure 4.5) and it triggers and manages all communication activities in the system.

Concerning the synchronous traffic, the FTT Interface Layer receives and decodes the EC trigger message and transmits messages that carry entities produced locally and requested elsewhere, according to the information of the EC-Schedule. On reception of synchronous real-time frames the FTT Interface Layer matches the ID of the received messages with the list of the locally consumed entities, by checking the Node Requirements Database. If the received message is locally consumed, its local buffer is updated with the received data.

With respect to the asynchronous traffic, the FTT Interface Layer computes the temporal limits of the asynchronous window and when the asynchronous window begins it gets the asynchronous messages (if any) from the respective queues and transmits them according to the particular arbitration mechanism used. Moreover, the FTT interface layer must also detect the end of the asynchronous window and prevent the start of any message transmission that does not fit within this window, in order to enforce temporal isolation between traffic classes. On reception of asynchronous real-time frames the FTT Interface Layer matches the ID of the received messages with the list of the locally received entities, by checking the Node Requirements Database. If the received message is locally received, the received data is

placed in the respective reception queue.

Moreover, the FTT-Interface layer also receives the polling requests issued by the master node concerning the non-real-time traffic and transmits the required data right after the reception of the pol command. On reception, the non-real-time traffic is intercepted and queued by the FTT Interface Layer. Whenever the received non-real-time data frames are from a foreign protocol, they are unwrapped and reassembled (if required by the particular implementation) and then sent to the non-real-time stack. This methodology makes the FTT protocol operation fully transparent from the point of view of the non-real-time applications.

Additionally, the FTT interface layer is also responsible for the management of the temporal accuracy information of real-time entities. Associated with each real-time entity there is a timer, which is set to the validity interval, as specified by the application layer for the particular real-time entity, when the local buffer is updated. The timer is then decremented while the message waits to be transmitted, and its actual value at transmission time is inserted in the message just before its transmission. On the consumer side, the timer continues being decremented. Whenever the application software consumes the real-time entity, the associated timer value is also delivered together, allowing it to assess whether their value is still within the defined temporal validity window. Since message deadlines are expressed in EC duration multiples (E), the resolution of the temporal accuracy timer is also E, which reduces the overhead associated to their maintenance.

4.3 Synchronous Traffic Analysis

As discussed in Section 2.3, hard real-time systems demand a high degree of predictability, thus the feasibility of the schedule should be guaranteed in advance. Moreover, in on-line scheduled systems like FTT, messages can be created, changed and removed dynamically during runtime. In this case a suitable admission control mechanism is required to assess during system run-time if such operations can be accepted, that is, if the resulting message set is schedulable.

The remaining of this section is devoted to the discussion of schedulability tests that can be used for on-line admission control.

4.3.1 Synchronous Message Model

As discussed in Section 4.2.2, the scheduling model used for the synchronous traffic does not allow the transmission of messages to cross the boundary of the synchronous window. This is achieved by using inserted idle-time, i.e., whenever a message does not fit completely within the synchronous window of a given EC it is delayed to the next. Moreover, this same behavior is also enforced in the asynchronous window, despite its implementation being somehow different. Consequently, the EC trigger message is always transmitted regularly, without any blocking. The only limitation on the regularity of the EC results from the imprecision of the internal master clock and from the jitter that the supporting Operating System can induce in the activation of the Dispatcher task. Nevertheless, by proper selection of hardware and operating system, such imprecisions can be bounded to a value that can be safely neglected, typically a small fraction of the duration of the smallest message that can be transmitted over the bus. However, the use of inserted idle-time has also a negative impact on the traffic schedulability, since within the synchronous window it corresponds to a reduction on its length, and on the asynchronous window it corresponds to bus time that is wasted, since no messages are transmitted at all in it.

Besides the issue of the inserted idle-time, the synchronous message model of FTT can be characterized as follows:

- synchronous message periods P_i and relative deadlines D_i are integer multiples of the elementary cycle duration (E);

$$\forall_i \, P_i = m * E \, ; \, D_i = n * E, \, m, n \in \mathbb{N} \tag{4.7}$$

- all instances of a synchronous message SM_i are regularly activated ($a_{i,k}$), according to its period P_i;

$$\forall_i \, , a_{i,k} = k * P_i, \, k \in \mathbb{N} \tag{4.8}$$

- all instances of a synchronous message SM_i have the same relative deadline D_i, which is less than or equal to the respective period P_i;

$$\forall_{i,k} \, , d_{i,k} = a_{i,k} + D_i \tag{4.9}$$

- all instances of a synchronous message SM_i have the same worst-case transmission time C_i;

$$\forall_i \, , c_{i,k} = C_i \tag{4.10}$$

- worst-case message transmission times are necessarily shorter than the maximum synchronous window length (LSW);

$$\forall_i \, , C_i < LSW \tag{4.11}$$

- message activations are always synchronous with the start of the EC;

$$\forall_{i,k} \, , a_{i,k} = m * E, \, m \in \mathbb{N} \tag{4.12}$$

Moreover, it is assumed that all synchronous messages are independent.

In [AF01], *Almeida et al* present several techniques for the schedulability analysis of task sets scheduled with inserted idle-time, in similar conditions to those referred above. The model used to schedule the synchronous traffic in FTT is very similar to the one presented in [AF01], named *blocking-free non-preemptive scheduling*. In this model, tasks periods and deadlines are integer multiples of a basic cycle duration (E), the execution times are always shorter than E and task activations are always synchronous with the start of a cycle. The only difference is that in [AF01] the whole cycle is available to execute tasks, while in the FTT model the synchronous traffic is restricted to the synchronous window within each EC, with maximum length LSW.

One of those techniques is based on the adaptation of the existing analysis for preemptive scheduling of tasks with fixed priorities. Basically, it consists in inflating the message transmission times by a factor that allows accounting for the inserted idle-time. This adaptation is pessimistic by considering that the inserted idle-time always has its maximum value in every cycle, thus leading to an analysis that is sufficient, only. Another technique is based on the construction of the timeline during the longest busy interval. In this case, it is possible to calculate the exact amount of idle-time inserted in each EC during the busy interval, and thus a necessary and sufficient analysis is supported.

In both cases the analysis in [AF01] requires a simple modification to account for the impact of the EC trigger message and asynchronous phase,

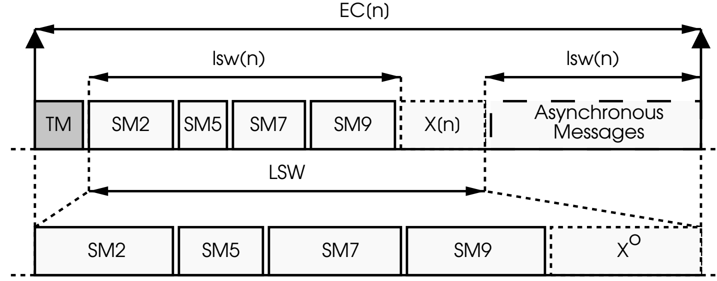


Figure 4.6: Expanding the synchronous window to allow using the blocking-free non-preemptive model

therein not considered.

4.3.2 Utilization-based schedulability analysis

In order to transform the FTT message model into the task model used in [AF01], so that the analysis therein presented can be used, it is necessary to model the effect of both the trigger message and the limitation on the length of the synchronous window, which can be restricted only to a fraction of the EC length.

A simple technique to model these effects is to inflate all execution times by a factor equal to $\frac{E}{LSW}$. This is equivalent to expanding the synchronous window up to the whole EC (Figure 4.6) and carries no consequence in terms of schedulability since messages scheduled for a given synchronous window will remain within the same cycle. Applying this transformation to the original set of messages SRT (Definition 4.1) results in a new virtual set that can be expressed as SRT^o (Definition 4.13) in which all the remaining parameters but the execution times are kept unchanged.

$$SRT^o \equiv \{SM_i^o(DLC_i, C_i^o, Ph_i, P_i, D_i, Pr_i),\, C_i^o = \frac{E}{LSW} * C_i,\, i = 1..N_S\} \quad (4.13)$$

The results in [AF01] are now directly applicable over SRT^o, particularly the theorem stating that any existing analysis for fixed priorities preemptive scheduling can be used in this model if the execution times C_i^0 are replaced by C_i' as in Equation 4.14, where E is the cycle duration and X^o the maximum inserted idle-time ($X^o = \max_n(X_n^o)$).

$$C_i' = \frac{C_i^o * E}{E - X^o} \tag{4.14}$$

Expanding 4.14 with the transformation in 4.13 and noting that $X^o = \frac{E}{LSW} * X$, yields the final transformation (Equation 4.15) that has to be carried out over the original message transmission times, i.e. those in the SRT, so that any existing analysis for fixed priorities preemptive scheduling can be used.

$$C_i' = C_i * \frac{E}{LSW - X} \tag{4.15}$$

However, any schedulability assessment obtained via that theorem is just sufficient, only. The reason is the pessimism introduced when using an upper bound for X. Except for a few particular situations, the exact value $X = \max_n(X_n)$ cannot be determined. Nevertheless, an upper bound is easy to obtain, e.g. the transmission time of the longest message among those that can cause inserted idle-time [AF01].

An important corollary of the theorem referred above is that Liu and Layland's utilization bound for Rate Monotonic [LL73] can be used with just a small adaptation as part of a simple on-line admission control for changes in the SRT incurring in very low run-time overhead. This is expressed in Condition 4.16.

$$\sum_{i=1}^{N_s} \left(\frac{C_i}{P_i}\right) < N_s(2^{\frac{1}{N_s}} - 1) * \left(\frac{LSW - X}{E}\right) \Rightarrow \begin{array}{c} SRT\ schedulable \\ with\ RM\ under \\ any\ phasing \end{array} \tag{4.16}$$

A similar line of reasoning can be followed to adapt the Liu and Layland's utilization bound for EDF [LL73]. In this case, the maximum inserted idle-time (X) plus the remaining amount of time in the EC outside the synchronous window ($E - LSW$) can be considered as the worst-case transmission time of a virtual message v, with worst-case transmission time $C_v = E - LSW + X$, that is added to the original set and transmitted every EC ($P_v = 1EC$), as depicted in Figure 4.7.

This virtual message v has the highest possible priority, since $P_v = D_v = 1\,EC$, and fills in the part of the EC that cannot be used by the synchronous messages. Assume, now, that the resulting extended set, i.e. the original

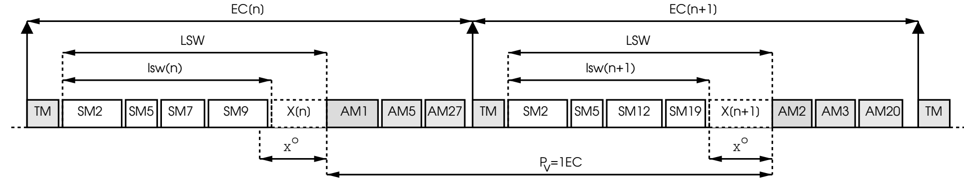


Figure 4.7: Modeling the effect of the inserted idle-time, asynchronous window and trigger message

SRT plus the virtual message, can be scheduled preemptively. Due to the absence of preemption instants, motivated by the synchronous activation model, and due to the absence of blocking, due to the inserted idle-time, the Liu and Layland's bound can be used (Equation 4.17).

$$U_v = \frac{E - LSW + X}{E} + \sum_{i=1}^{N_s}(\frac{C_i}{P_i}) \leq 1 \tag{4.17}$$

However, due to the extra load imposed by the virtual message, all other messages will finish transmission either in the same EC or later in this schedule than in the original one with the traffic confined to the synchronous window and with inserted idle-time. Thus, if the extended set is schedulable the SRT will also be. This results in the sufficient schedulability condition 4.18.

$$\sum_{i=1}^{N_s}\left(\frac{C_i}{P_i}\right) \leq \frac{LSW - X}{E} \Rightarrow \begin{array}{c} SRT\,schedulable \\ with\,EDF\,under \\ any\,phasing \end{array} \tag{4.18}$$

The analysis above presented is pessimistic, because it considers that the inserted idle-time always has its maximum value, thus leading to an analysis that is sufficient, only. However, in the FTT context these schedulability tests are executed on-line. In highly dynamic applications, with frequent changes to the message set or in which the system's response to change requests must be prompt, schedulability tests should have the lower computational complexity possible. Both schedulability tests presented above have a computational complexity of $O(n)$, similar to the one of the original Liu and Layland's analysis [LL73], and can be computed in $O(1)$, by keeping track of the current message set utilization, when used on-line.

4.3.3 A necessary and sufficient schedulability test

As discussed in Section 2.4.3, response-time based schedulability tests are usually less pessimistic than their utilization-based counterparts, and also provide estimations of the actual worst-case response time of each message. However, the trade-off is a higher computational complexity. In applications that do not have strict restrictions in the response time of change requests to the message set properties, or, in other hand, in systems where the critical resource is not the computational power but the transmission medium bandwidth utilization, it may be desirable to have more accurate schedulability tests.

In [AF01] *Almeida et al* also present a new analysis based on a traffic timeline, which allows obtaining an accurate schedulability assessment for fixed priorities scheduling such as RM and DM. Moreover, the analysis therein presented becomes necessary and sufficient if both of the following assumptions are verified:

A1. All messages must be considered in-phase, i.e., ready for transmission at a hypothetical instant t=0 called critical instant (worst-case phasing);

A2. No lower priority message can be scheduled before a higher priority one. Otherwise, one could not guarantee that the first message instance after the critical instant suffers the worst-case response time.

This analysis requires the execution of a simple algorithm (Algorithm 4.1) to obtain the worst-case response times to transmission requests ($Rwc_i, i = 1..N_s$), considered as the maximum time lapse from message exact periodic activation to complete transmission.

```
1.   for (k = 1 ; k ≤ Ns ; k++) { Rwc_k = 0 ; r_k(1) = 1; }
2.   for (n = 1 ; (n ≤ D_Ns and Rwc_Ns = 0) ; n++) {
3.       lsw(n) = 0;
4.       for (k = 1 ; k ≤ Ns ; k++) {
5.           r_k(n+1) = r_k(n);
6.           if (lsw(n) + r_k(n)*C_k ≤ LSW) {
7.               lsw(n) = lsw(n) + r_k(n)*C_k;
8.               r_k(n+1)=0;
9.               if (Rwc_k = 0) Rwc_k = n;
10.              }
11.          if (n mod P_k = 0) r_k(n+1) = 1;
12.          }
13.      }
```

Algorithm 4.1: Timeline analysis

The algorithm consists in determining, for all messages, the EC where they are first transmitted after the critical instant (line 9). This is carried out EC by EC (line 2), taking into account the effective message sequence in the schedule imposed by the respective priorities (line 4). This way, the inserted idle-time in each EC is accounted for with exactitude (lines 6 and 7), consequently resulting in exact worst-case response times.

The algorithm herein presented differs from the one in [AF01] in that it accumulates the load of each EC ($lsw(n)$) up to the maximum length of the synchronous window (LSW) only, and calculates the worst-case response time with a resolution of one EC. At the end of each complete run of the inner for loop in line 4, $lsw(n)$ contains the effective duration of the synchronous window in the nth EC. The vector $r_k = 1..Ns(n)$ indicates the messages with transmission requests pending in the nth EC. After having determined the worst-case response times for all messages, a trivial schedulability test can be carried out by comparing this time with the respective deadline. As long as both conditions referred above hold, the test supports a necessary and sufficient condition (4.19).

$$Rwc_i \leq D_i, \forall i = 1...N_s \Leftrightarrow \begin{array}{c} SRT\, is\, schedulable \\ with\, worst-case \\ phasing \end{array} \tag{4.19}$$

In case assumptions A1 or A2 do not hold, the values of Rwc_i obtained from the Algorithm 4.1 may not be exact but upper bounds to the effective worst-case values, and thus the schedulability test results in a sufficient but not necessary condition.

This method has a computational complexity $O(m * n)$, where m is the deadline range, in ECs, and n the number of synchronous messages that fit on the EC. Moreover, the computational demand of each of the elementary steps in the algorithm (line 5-10) is also considerably more costly than in the case of utilization-based tests, which consists in just a sum for each message. Since the decision on accepting or rejecting change requests to the message set only can be taken after the completion of the schedulability analysis, it must be assessed if the increased computational complexity and accuracy of this method when compared with the utilization based method (Section 4.3.2) pays off, specially in targets having constrained computational power, as frequently found in embedded applications.

4.4 Asynchronous traffic analysis

The asynchronous traffic carried on a fieldbus may have different properties and requirements. For instance, messages related with critical alarms must be schedulable even in worst-case scenario, and transmitted within bounded and known delay. However, messages related to data logging or system management usually can be delayed without compromising the system. Also, messages due to the Human-Machine Interface (HMI) can suffer a delay in the order of one second, without noticeable impact in the overall system performance.

Asynchronous messages are scheduled strictly according to fixed-priority policies. Whenever this feature is not natively supported by the underline communication network, the FTT AMS must override the respective MAC and enforce this behavior.

The Asynchronous Messaging System of FTT is deemed to guarantee the schedulability of all the hard real-time critical messages, even in worst-case conditions, and provide good average response time for soft and non real-time messages. For messages with deadline greater than the respective minimum inter-arrival time, the FTT AMS provides local queuing.

Three classes of messages are supported by the FTT AMS:

AT1. hard real-time sporadic messages with deadlines less or equal to the respective minimum inter-arrival time;

AT2. hard-real time sporadic messages with deadlines greater than the period, or that despite not having strict deadlines require guaranteed delivery (queuing required);

AT3. soft and non-real-time sporadic messages.

Hard real-time messages (classes AT1 and AT2) must be timely handled in any workload conditions, therefore pre-runtime analysis must be provided. Messages belonging to class AT3 are handled under a best-effort policy, and therefore no timeliness guarantees are provided.

4.4.1 Worst-case response time for AT1 asynchronous message class

The FTT asynchronous messaging system provides schedulability guarantees for hard sporadic messages, i.e., messages with a defined minimum inter-arrival time and hard deadlines. As referred in Section 4.2.2, asynchronous messages are transmitted in a period of time called asynchronous window. Only asynchronous messages that fit completely within that window are transmitted, therefore the temporal isolation of both synchronous and asynchronous phases of the EC is guaranteed.

The set of real-time asynchronous communication requirements is kept in the Asynchronous Requirements Table, characterized by Definition 4.2. Let the subset of the ART composed by the asynchronous messages having hard real-time requirements be denoted by ART^{RT} (Definition 4.20).

$$ART \supset ART^{RT} \equiv \{AM_i^{RT}(DLC_i, C_i, mit_i, D_i, Pr_i),\ i = 1..N_A^{RT}\} \quad (4.20)$$

Each entry in this table describes one asynchronous message stream, which must always be of a sporadic nature, i.e. there is a minimum inter-arrival time (mit) that must elapse between consecutive messages of the same stream. Notice that in the ART there may exist soft or non-real-time asynchronous messages which, for the sake of flexibility, are not constrained except by the assignment of a lower priority than hard real-time asynchronous messages.

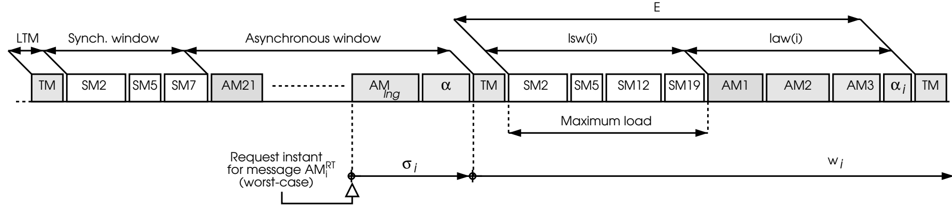


Figure 4.8: Maximum dead-interval (σ_i) and level-i busy window (w_i)

The maximum time that may elapse from a transmission request for real-time asynchronous message i (AM_i^{RT}) to complete message transmission is called the worst-case response time (Rwc_i) and is given by Equation 4.21.

$$Rwc_i = \sigma_i + w_i + C_i \tag{4.21}$$

The parameter σ_i corresponds to the time lapse between the request and the instant in which the message can enter in arbitration. It is a blocking term, denoted as **dead interval**. The parameter w_i allows to account for the interference caused by higher priority messages in the arbitration process until message AM_i^{RT} starts its transmission. This is known as **level-i busy window**. The critical instant for each message is defined as the instant that maximizes both σ_i and w_i.

Figure 4.8 shows the conditions that maximize the dead interval σ_i. This happens when, cumulatively:

- The transmission request occurs within the asynchronous window but there is already on the bus the longest lower priority message (AM_{lng});
- When the transmission of the lower priority message completes there is not enough time left in the asynchronous window for the transmission of message AM_i^{RT}, leading to insertion of idle-time (α);

The transmission time of message AM_{lng} can be upper bounded by considering the maximum transmission time among all lower priority asynchronous and non-real-time messages ($Ca = max(C_i, C_j) : C_i \in ART; C_j \in NRT$). On the other hand, the inserted idle-time (α) can be upper bounded by the transmission time of the message whose response time is being computed (C_i). However, if Ca is used instead of C_i, the value of σ_i will be slightly more pessimistic but it will become a constant, thus considerably

easier to use within calculations. Hence, an upper bound to the dead interval (σ^{ub}) can be derived through Equation 4.22.

$$\sigma^{ub} = 2 * Ca \tag{4.22}$$

The level-i busy window (w_i) starts just after the dead interval. Its maximum duration occurs when, cumulatively:

C I. All higher priority asynchronous messages were synchronously requested as soon as possible after the beginning of the dead-interval σ_i, i.e., synchronously with the request for AM_i^{RT}. This maximizes the number of multiple instances of each higher priority message that may occur during the busy window;

C II. The EC that follows the start of the busy window is also the critical instant for the synchronous traffic. This means that the sequence of ECs starting in the busy window contains the highest cumulative load demanded by the synchronous traffic.

To compute w_i it is important to determine the duration of the asynchronous windows within the ECs that follow the critical instant up to the one where message AM_i^{RT} can be effectively transmitted. This is achieved indirectly by determining the duration of the synchronous windows, which, in turn, can be obtained by inspection of the Synchronous Requirements Table. A vector (lsw) can then be built containing those values for the respective ECs. The number of ECs contained in the vector must cover w_i. Since this is unknown in the beginning, the vector is calculated iteratively, EC by EC, simultaneously with w_i. A method that can be used to generate the vector lsw based on the SRT is presented in [Alm99]. Equation 4.23 shows the conversion of the lsw into the law vector that contains, in the kth position, the length of the asynchronous window of the kth EC after the critical instant.

When a given synchronous message does not fit within the synchronous phase of an EC, it is successively postponed until one with enough room is found. Since neither the length of the EC nor the length of the synchronous phase are correlated with the length of the synchronous messages, idle-time can be inserted in the synchronous phase. This effect can lead to a situation

where the initial ECs after the critical instant do not have the highest synchronous load, because they may be affected by an higher inserted idle-time, thus lower load, than other ECs. To account for this effect on the analysis, the lower branch of Equation 4.23 maximizes the length of the synchronous window whenever inserted idle time may have been included.

$$law(k) = \begin{cases} E - LTM - lsw(k) & , lsw(k) + C_s < LSW \\ E - LTM - LSW & , lsw(k) + C_s \geq LSW \end{cases}$$
$$k = 1 \ldots \left\lceil \frac{w_i}{E} \right\rceil ; C_s = \max_{i=1 \ldots N_s}(C_i) \, : \, C_i \in SRT \tag{4.23}$$

The analysis that follows cannot directly use the results available for fixed priority task scheduling (e.g [THW94]), because of the variable length synchronous window and inserted idle-time. However, such results can be easily adapted as shown below. Generically speaking, the main difference is that the cumulative demand for bus time by the asynchronous messages with priority higher than Pr_i (i.e. $H_i(t)$) cannot be compared against linear time t. Instead, it must be compared against a function of t ($A(t)$) that returns the cumulative bus time available for asynchronous messages. This function must account for both effects referred above, i.e. variable synchronous windows and inserted idle-time. The value of w_i corresponds to the value of t that makes $H_i(t) = A(t)$, i.e. demand equal to availability (Figure 4.9).

The demand function $H_i(t)$ can be obtained by the usual way as in processor scheduling theory using Equation 4.24. It accounts for the maximum bus time demanded by the set of asynchronous messages with higher priority than that of message $AM_i^{RT}(hp_i)$. The addition of σ^{ub} to t is required by condition (C I) above. Since C_j represents the worst-case message transmission time, including all possible protocol overheads, and σ^{ub} is used instead of σ_i, the result will also be an upper bound to the effective maximum demand.

$$H_i(t) = \sum_{j \in hp_i} \left\lceil \frac{t + \sigma^{ub}}{mit_j} \right\rceil * C_j \tag{4.24}$$

Function $A(t)$ can be obtained by using the vector law as in Equation 4.25. Figure 4.9 shows how it is built. Notice that α_j stands for the inserted idle-time in the jth EC. However, since the exact values for α_j are unknown unless the exact order by which messages are transmitted is taken into account (which is not the case with Equation 4.24), the upper bound Ca can

be used instead, resulting in a lower bound for $A(t)$.

$$A(t) = \begin{cases} \sum_{j=1}^{k-1} \left(law(j) - \alpha_j\right), \\ \quad t: \ (k-1) * E \leq t < k * E - (law(k) + \alpha_k) \\ \\ \sum_{j=1}^{k-1} \left(law(j) - \alpha_j\right) + t - (k-1) * E, \\ \quad t: \ k * E - (law(k) + \alpha_k) \leq t < k * E - \alpha_k \\ \\ \sum_{j=1}^{k} \left(law(j) - \alpha_j\right), \\ \quad t: \ k * E - \alpha_k \leq t < k * E \\ \qquad with\, k - 1 = \lfloor \frac{t}{E} \rfloor \end{cases} \tag{4.25}$$

By using an upper bound for $H_i(t)$ and a lower bound for$A(t)$, the resulting value ofw_i will also be an upper bound. Its calculation is reduced to solving Equation (4.26).

$$w_i^{ub} = t : H_i(t) = A(t) \tag{4.26}$$

This equation can be solved iteratively by using $t^1 = H_i(0^+)$ and $t^{n+1} = t : A(t) = H_i(t^n)$. The process stops when $t^{n+1} = t^n$ (and $w_i^{ub} = t^{n+1}$) or $t^{n+1} > D_i - C_i - \sigma^{ub}$, and thus the deadline cannot be guaranteed. One or the other situation will occur in a bounded number of iterations, since the increment in each iteration is lower bounded by the transmission time of the smallest real-time asynchronous message. An upper bound to the worst-case response time for message AM_i^{RT} (R_i^{ub}) can be obtained through expression 4.21, replacing w_i by w_i^{ub} obtained from Equation 4.26, and σ_i by σ^{ub} obtained from Equation 4.22.

4.4.2 Worst-case response time for AT2 asynchronous message class

Some systems convey messages with deadlines greater than the minimum inter-arrival time or even not having strict deadlines at all, but for which the delivery should be guaranteed. For example, consider an assembly line in which whenever an item passes a given processing step an event message is sent to the inventory database. Usually there are no strict deadlines concerning the database update, therefore the transmission of these messages

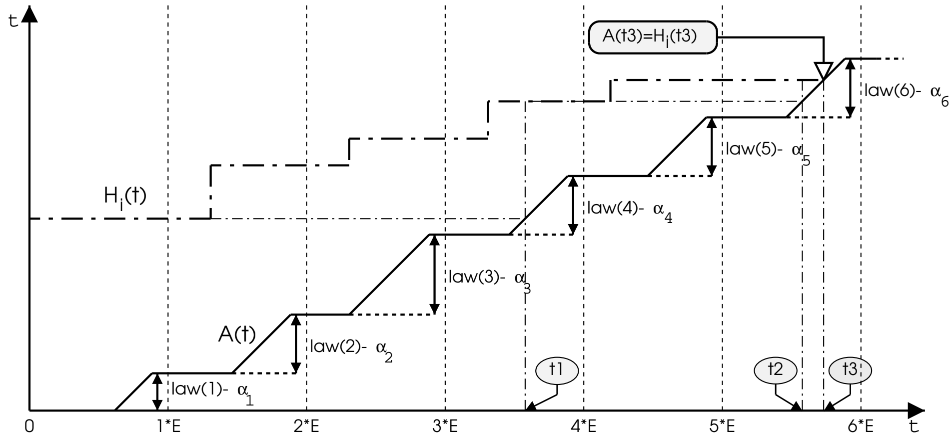


Figure 4.9: Calculating the level-i busy window

can be delayed if more urgent ones, for instance related with alarms, are ready. Nevertheless, it is important to guarantee that all the messages will be eventually transmitted. In this situation each station must queue the events until they can be transmitted. The message queuing could be performed by the user application. However it is safer and more efficient if this service is delivered by the communication system itself, because it has complete knowledge about the communication requirements, therefore can assess in advance whether it is possible to guarantee the message delivery, and also compute the queue length required.

Results from queuing theory allow obtaining statistic guarantees, knowing some key properties on the demand side. However, the methodology here proposed is based on worst-case analysis, thus, in any anticipated workload conditions the message delivery is guaranteed.

The analysis presented in Section 4.4.1 can be extended to accommodate the situation where messages have deadlines greater than the period. For this situation, the demand function (Equation 4.24) must include the maximum load due previous requests of the asynchronous message stream that are queued for transmission. In this scenario, the demand function ($H_i^q(t)$) becomes:

$$H_i^q(t) = \sum_{j \in hp_i} \left\lceil \frac{t + \sigma^{ub}}{mit_j} \right\rceil * C_j + \left\lceil \frac{t + \sigma^{ub}}{mit_i} \right\rceil * C_i \tag{4.27}$$

Note that Equation 4.27 includes also the demand of hard real-time asyn-

chronous messages belonging to class AT1, since these ones have the higher priority among all asynchronous messages.

The value of the upper bound for the level-i busy window (w_i^{ub}) is given by Equation 4.28.

$$w_i^{ub} = t : H_i^q(t) = A(t) \tag{4.28}$$

This equation can be solved used the same methodology used for solving Equation 4.26 in the previous section. However, recall that both of these equations only converge if the availability function ($A(t)$) grows at a faster rate that the demand function ($H_i(t)$). When solving Equation 4.26 iteratively, the stop condition concerning the message deadline ensures that the iteration always stops in a finite amount of time. However, since here we are considering also the possibility of messages without deadlines, it is necessary to use some other stop condition, ensuring that the computation stops in a finite amount of time even if the demand and availability functions do not converge. For practical reasons, one such criteria can be placing a limit on the maximum length of the queue, since in real implementations the amount of memory is always limited, and so must be the amount of memory reserved for queues.

Equation 4.29 can be used to provide at any time an upper bound on the maximum number of buffers required to queue the pending requests concerning a particular message i, simply substituting w_i^{ub} by the time instant in which this evaluation is performed.

The demand function H_i^q that appears in Equation 4.28 returns the worst-case amount of time required to dispatch all instances of message i. Therefore an upper bound on the number of transmission buffers (NB) that must be reserved for message i can be computed simply by calculating the maximum number of instances that can occur during that time interval (Equation 4.29). This method is simple since it requires only a short additional calculation performed after the computation of the dead interval and level-i busy window, but it is also pessimistic, since it does not consider that during this time interval some instances of the message can be transmitted, thus releasing buffers in the queue. A less pessimistic upper bound could be obtained by determining the time instants of all events, both transmission requests and transmissions, during the time interval starting from the critical

instant until the transmission of the last queued instance of the message, and compute the balance between the requests and transmissions. However this method is considerably more costly concerning the amount of computations required, when compared to the results given by Equation 4.29.

$$NB_i = \left\lceil \frac{w_i^{ub} + \sigma^{ub}}{mit_i} \right\rceil \tag{4.29}$$

Experimental results using this analysis are presented further on, concerning the FTT-CAN protocol (Section 6.3). It should be also referred that these analysis are not easily implemented on-line, not only due to the computation cost but also because of the interference with the synchronous requirements. Nevertheless, this analysis can be performed off-line. For systems with fixed synchronous requirements, its use is straightforward. For systems with dynamic synchronous communication requirements it is still possible to perform the analysis off-line, but in this case based in worst-case synchronous load scenarios.

4.5 Conclusion

This chapter starts by a discussion about the requirement for flexibility that is becoming increasingly important in distributed computer-controlled applications, either motivated by the need to reduce the costs of set-up, configuration changes and maintenance or by the appearance of applications such as agile manufacturing, real-time database, automotive, mobile robotics and machine vision, that must deal with environments that are inherently dynamic.

Since current protocols do not cope efficiently with these requirements (Sections 3.2 and 3.3), this discussion fosters the proposal of a new communication paradigm, the Flexible Time-Triggered paradigm (FTT), which has been developed specifically to support such type of flexible applications. The FTT paradigm supports on-the-fly changes to the message set, arbitrary scheduling policies, on-line admission control of real-time traffic, and support for different types of traffic with temporal isolation.

Schedulability analysis plays a fundamental role in real-time systems, since it is this tool that enables to assess if the time-critical activities carried by the system can meet its deadlines. Therefore, after the presentation of the

FTT architecture, it follows a section addressing the schedulability analysis issue concerning the synchronous traffic. In particular, are included utilization, response times and timeline schedulability tests. All of these methods are useful, since they provide results with distinct degrees of pessimism but at the same time have also distinct computational complexities. Therefore, it becomes possible to trade bus utilization efficiency by computation complexity, and thus to select the solution that better fits the particular application being developed.

Many real-time activities are asynchronously triggered by unforeseen events, for instance, messages related with alarms. Despite its common asynchronous nature, these events are heterogeneous concerning its timeliness requirements. Some, like the case of the alarms referred above, must be transmitted within bounded and pre-defined time intervals; others exhibit soft real-time requirements, and thus failing their delivery does not seriously compromise the system behavior; finally, some other events have no timeliness requirements at all. The FTT paradigm supports three different classes of asynchronous traffic: hard real-time asynchronous messages, with deadlines less than or equal to their minimum inter-arrival times (AT1); hard real-time asynchronous messages with deadlines greater than their minimum inter-arrival times or without strict deadlines but that require guaranteed delivery (AT2); soft and non real-time asynchronous messages. This chapter includes schedulability tests for the hard real-time types (AT1 and AT2), which allows to know in advance if the system is able to handle timely all those activities in all anticipated circumstances. Moreover, for AT2 messages the schedulability test herein presented also provides an upper bound for the number of buffers required to handle the message instances that may be queued, waiting for transmission.

Chapter 5

QoS management based on FTT

Due to continued developments along the last decades in the integration of processing and communications technology, distributed architectures have progressively become pervasive in many real-time application domains, ranging from avionics to automotive, adaptive control, robotics, computer vision and multimedia. In these systems, there has also been a trend towards higher flexibility in order to support dynamic configuration changes such as those arising from evolving requirements and on-line Quality-of-Service (QoS) management [S$^+$96]. These features are generally useful to increase the efficiency in the utilization of system resources [BLCA02] since typically there is a direct relationship between resource utilization and delivered QoS. In several applications, assigning higher CPU and network bandwidth to tasks and messages, respectively, increases the QoS delivered to the application. This is true, for example, in control applications [BA00], at least within certain ranges [Mar02], and in multimedia applications [LRM96]. Therefore, managing the resources assigned to tasks and messages, e.g. by controlling their execution or transmission rates, allows a dynamic control of the delivered QoS. Efficiency gains can be achieved in two situations: either maximizing the utilization of system resources to achieve a best possible QoS for different load scenarios or adjusting the resource utilization according to the application instantaneous QoS requirements, using only the resources required at each instant and maximizing the bus availability to asynchronous traffic.

Both situations referred above require an adequate support from the computational and communications infrastructure so that relevant parameters of tasks and messages can be dynamically adjusted. In the scope of this thesis this problem is regarded from the communications perspective only, considering an autonomous communication system that manages streams of messages, very much like a processor executes tasks. This approach is more robust and particularly adapted to distributed real-time systems with fault-tolerance requirements [Kop97].

Dynamic QoS management implies on-line changes to the traffic characteristics, such as addition, removal and adaptation of message properties. Moreover, some of the message streams have real-time QoS constraints, arising for example from control and monitoring requirements, which must be always fulfilled. Unfortunately, as discussed in Section 4.1, most of the existing communication protocols are not well suited to support the flexibility requirements presented by distributed real-time systems that implement dynamic QoS management functionalities. On the other hand, general purpose protocols such as IBM Token Ring, FDDI and ATM have some level of support for such QoS requirements, but are not broadly used as fieldbuses because of outdated technology or high cost.

5.1 Adding a QoS manager

According to the FTT architecture (Chapter 4) the scheduling activity is performed on-line, based on the actual message properties stored in the SRDB (Figure 4.3). This mechanism is the source of the operational flexibility exhibited by the FTT paradigm concerning the synchronous traffic. When the message set is changed, in its next activation the Scheduler uses the updated values, and thus the following EC-Schedules include the new communication requirements.

In its most basic functionality level, the FTT paradigm requires change requests to be handled by an on-line admission control. The purpose of this mechanism is to assess, before commitment, if the requests can be accommodated by the system i.e., if the message set that would result of the incorporation of the requested changes would still be schedulable. In this case, the changes can be safely committed to the SRDB, and consequently the request is accepted. Conversely, if the change request would result in an

unfeasible message set, it is rejected and the SRDB is kept unchanged.

From this point of view, the master node can be seen as a QoS server in the sense that when a message is admitted or changed, the master node verifies if its associated requirements (memory, network bandwidth, message deadline and jitter, etc.) can be fulfilled, and in this case also reserves these resources in a way that they will be strictly available in the future, assuring that all the accepted messages will receive the requested QoS.

Particularly concerning QoS requirements, some applications benefit or even require the definition of ranges of acceptable QoS levels. This is the case when system activities vary their requirements during the system lifetime, in response to environment changes. To handle these requirements efficiently, the communication protocol should not only guarantee that the minimum requirements will be fulfilled in all anticipated conditions, but also grant in all instants the higher QoS possible to all the activities. Moreover, it can also be required to support different levels of importance for these activities, implying that some of them can be favored with respect to the others, according to some well defined policy. The FTT paradigm can provide support for such advanced QoS management methodologies by aggregating a QoS manager to the on-line admission control block. With this architecture, the on-line admission control still decides about the acceptance of change requests based on the minimum requirements of the existing message streams. This will eventually generate some spare resources, e.g. spare bandwidth, that will be distributed by the QoS manager according to a pre-defined policy.

As described in Section 4.2.3, the master node holds in the Synchronous Requirements Table the properties of the synchronous message set. The SRT, besides the basic message properties (e.g. Period, Deadline) also provides room for extended data via the Xf field (Definition 4.1). The QoS manager can use this field to store the relevant properties for each of the synchronous messages. Examples of such properties are the specification of the admissible QoS ranges, relative importance and criticalness.

The FTT paradigm is based on a modular design, with well defined interfaces between the system components. The Scheduler bases its decisions on the actual contents of the SRT, so the QoS manager must map the communication requirements into standard message properties, such as periods (for an RM scheduler) and deadlines (for an EDF or DM scheduler). Moreover, SRT updates cannot be performed while the Scheduler is reading its contents

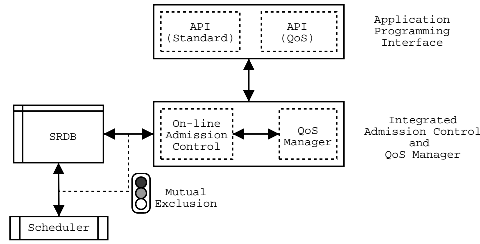


Figure 5.1: Adding QoS management to FTT

for building the following EC, therefore it is necessary to enforce atomic access to the SRT. If both of these properties are enforced, the operation of the Scheduler becomes completely independent not only of the existence of a QoS manager but also from the particular QoS management policy used. With respect to the Application Interface, the aggregated on-line admission control and QoS manager must implement the standard SRDB management functions (add, remove and change message properties), but can also extend the API to provide QoS management user-level functions specific to a particular QoS management policy, allowing for instance the application to request a given QoS for a specific message in response to environment changes.

5.2 Examples of QoS management policies

5.2.1 Priority-based QoS management

Many real-time systems are composed by sets of activities with distinct levels of importance concerning the behavior of the system. In these cases, QoS should be granted strictly according to the relative importance of these activities, with the more important ones receiving the highest QoS possible. A possible methodology to deal with this situation consists in assigning a QoS priority parameter to each of the activities. Then the QoS manager sorts the activities according to the QoS priority and distributes the required QoS to each one, when possible.

In the scope of real-time communications, a common QoS parameter consists in the bandwidth required my the message stream. In this case, the SRT (Definition 4.1) should be extended as follows:

$$Xf_i \equiv (V_i, T_{i_{min}}, T_{i_{max}}),\ i = 1..N_S \tag{5.1}$$

where V_i specifies the relative message importance and the minimum ($T_{i_{min}}$) and maximum ($T_{i_{max}}$) periods bound the bandwidth required by each message stream.

5.2.2 Elastic Task Model based QoS management

One of the characteristics of the priority-based QoS manager above presented is that the spare resources are distributed among the messages in a strict priority order. This might be restrictive when, for example, it is desirable to do a more equitable distribution of the spare resources. In this case, the Elastic Task Model QoS manager is more adequate since it allows a tighter control over the way the spare resources are distributed.

According to the elastic model proposed in [BLA98], the utilization of a task is treated as an elastic parameter, whose value can be modified by changing the period within a specified range. Each task is characterized by five parameters: a worst-case computation time C_i, a nominal period T_{i_0}, a minimum period $T_{i_{min}}$, a maximum period $T_{i_{max}}$, and an elastic coefficient E_i. Thus an elastic task can be denoted by:

$$\tau_i(C_i, T_{i_0}, T_{i_{max}}, T_{i_{max}}, E_i)$$

The elastic coefficient specifies the flexibility of the task to vary its utilization for adapting the system to a new feasible rate configuration: the greater E_i, the more elastic the task. Thus, from a design perspective, elastic coefficients can be set equal to values which are inversely proportional to task's importance.

Admission of new tasks or requests of variations in the properties of existing ones are always subject to an elastic guarantee and are accepted only if there exists a feasible schedule in which all the other periods are within their range. In [BLA98] it is proposed to scheduled tasks by the Earliest Deadline First algorithm [LL73], hence, if $\sum \frac{C_i}{T_{i_{max}}} \leq 1$ the task set is schedulable.

Whenever a feasible schedule exists, if $\sum \frac{C_i}{T_{i_{min}}} \leq 1$, all tasks can be created at the minimum period $T_{i_{min}}$, otherwise the elastic algorithm is used to adapt the task's periods to T_i such that $\sum \frac{C_i}{T_i} = U_d \leq 1$, where U_d is some desired utilization factor. The elastic algorithm consists first in computing by how much the task set must be compressed ($U_0 - U_d$) and then to determine

how much each task must contribute to this value, according to its elastic coefficient, as follows:

$$\forall i\, T_i = T_{i_{min}} - (U_0 - U_d)\frac{E_i}{E_v} \tag{5.2}$$

where U_0 is sum of nominal task utilizations and $E_v = \sum_{i=1}^{n} E_i$.

However, due to the period constraints ($T_{i_{min}} \leq T_i \leq T_{i_{max}}$) the problem of finding the values T_i can require an iterative solution, since during compression one or more tasks may reach their maximum period. In this case the additional compression has to affect only the remaining tasks. In [BLCA02] it is shown that, in the worst case, the compression algorithm converges to a solution (if there exists one) in $O(n^2)$ steps, where n is the number of tasks.

To cope with this framework the SRT (Definition 4.1) should be extended to incorporate the above referred parameters.

$$Xf_i \equiv (T_{i_{min}}, T_{i_0}, T_{i_{max}}, E_i),\ i = 1..N_S \tag{5.3}$$

5.2.3 Applying the Elastic Task Model to message scheduling

The Elastic Task Model was originally developed for task scheduling in single microprocessors. Under this framework, tasks are preemptive. However, in the context of message scheduling, message transmissions cannot be suspended and resumed later, therefore preemption is not allowed. Another difference refers to the resolution used to express periods, initial phasings and deadlines. The FTT paradigm uses a coarse resolution equal to the EC duration while in the original elastic task model the resolution can be arbitrarily small. Moreover, the transmission time of messages in FTT is always much smaller than the EC duration while in the elastic task model the task execution times are not constrained beyond a limited utilization factor.

Despite these differences, the elastic task model can be easily applied to the FTT framework. However, the periods resulting from Equation 5.2 are not necessarily multiples of the EC duration (E) and thus, they must be rounded up (Figure 5.2) to the next integer multiple of E (T_i'), as in (5.4). The rounding must be done in excess, in order to guarantee that the resulting message set does not have a greater utilization factor than desired (U_d). After rounding up the periods, each message utilization U_i' is given by (5.5) and the overall effective utilization U_{eff}' is obtained by summing U_i' for

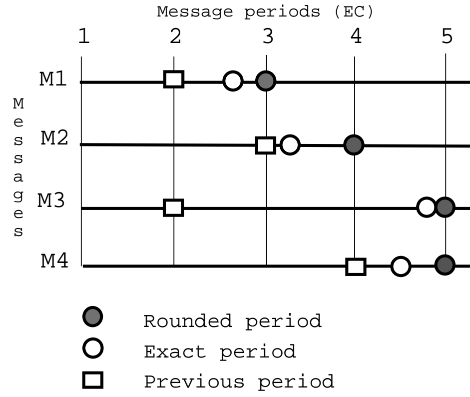


Figure 5.2: Rounding of periods in FTT-Ethernet.

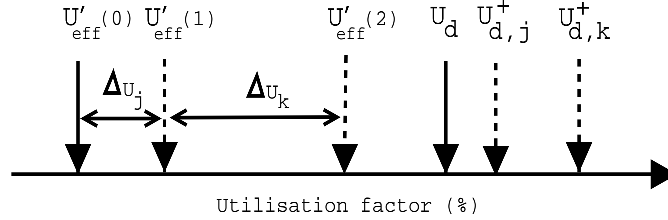


Figure 5.3: Increasing the effective utilization factor in FTT-Ethernet.

all i. Due to the rounding ups of the periods, $U'_{eff} \leq U_d$ (Figure 5.3).

To avoid this situation and improve the efficiency on the FTT implementation, the elastic task model was extended with an additional optimization step, performed after the initial compression algorithm, in which the spare utilization factor is better distributed among the messages. This redistribution is carried out coherently with the philosophy of the elastic model, i.e. guaranteeing that the resulting effective utilization factor does not exceed U_d (Figure 5.3).

The optimization step allows calculating a succession of effective utilization values $U'_{eff}(n)$ starting from U'_{eff} defined as above. Firstly, the process computes a vector with utilization values $U^+_{d,i}$ for every message i that can be decompressed (Γ_v) and has utilization lower than the one resulting from Equation 5.2, using Equation 5.8. Each of these values corresponds to the increased overall utilization that would result if the utilization of message i was enlarged as in Equation 5.6, due to reducing the respective period to the

nearest integer multiple of E. The vector $\{U^+_{d,i}\}$ is sorted in ascending order and for each i, if $U'_{eff}(n) + \Delta U_i \leq U_d$ then $U'_{eff}(n+1) = U'_{eff}(n) + \Delta U_i$ and the period of message i is reduced by E, the duration of one EC. After scanning the whole vector, the final message periods impose an overall bandwidth utilization factor that is potentially closer to the desired value U_d.

$$\forall \tau_i \in \Gamma_v \quad T'_i = \lceil T_i \rceil = \lceil \frac{C_i}{U_i * E} \rceil * E \geq T_i \tag{5.4}$$

$$U'_i = \frac{C_i}{T_i} \tag{5.5}$$

$$U^+_i = \frac{C_i}{T'_i - E} \tag{5.6}$$

$$\Delta U_i = U^+_i - U'_i \tag{5.7}$$

$$\forall \tau_i \in \Gamma_v \quad U^+_{d,i} = U_d + (U^+_i - U_i)\frac{E_v}{E_i} \tag{5.8}$$

5.3 QoS management case study: a mobile robot

5.3.1 Communication requirements

To illustrate the use of the FTT paradigm in providing dynamic QoS management, this section presents an hypothetical case study based on the requirements of a mobile robot that uses a distributed embedded control system. The robot should navigate autonomously within a delimited geographical area, and must exhibit the following behaviors: obstacle avoidance, path following and beacon tracking. The desired global robot behavior is determined by a subsumption architecture that arbitrates among the existing behaviors, deciding which is the active one. The behavior arbitration is carried out as follows:

1. whenever an obstacle is detected, avoid it;
2. in the absence of obstacle, follow a path indicated by a line on the floor;

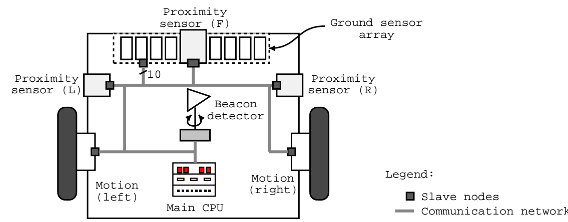


Figure 5.4: Robot components

3. in the absence of obstacle and line, track a beacon and move towards it;

4. otherwise move randomly.

To support the desired behaviors the robot is equipped with two independent motors, a set of three proximity sensors to detect nearby objects, a beacon detector, a line sensor made of an array of 10 individual sensors and a main CPU to execute the high level control and planning software (Figure 5.4). These elements are interconnected by a shared broadcast bus over which the FTT paradigm has been implemented. The FTT master is implemented in the main CPU, jointly with application tasks. The sensor readings are produced by the respective sensors and consumed by the main CPU. On the other hand, the main CPU produces the speed set-points that are consumed by the motor controllers, which execute closed-loop speed control. These controllers also produce displacement measures that are consumed by the main CPU to support trajectory control.

Table 5.1 characterizes the communication requirements, i.e. the message set and respective properties. Basically, each sensor will produce a 1-byte message with the respective reading except for the motor controllers that will produce a 2-byte message with the displacement information. The QoS requirements are expressed in terms of admissible ranges for the production rates of each message. Since specified periods are integer multiples of $10ms$, this value has been used to define the EC duration. Moreover, the synchronous window share was restricted to 80% of the EC duration. The remaining 20% were left for the trigger message as well as for possible asynchronous traffic, not defined here.

In order to derive tangible values, we assume an implementation over CAN [Rob91], operating of 100Kbps. Table 5.2 shows the resulting minimum

Source	Signal name	Data Bytes	# of Mesgs	Period(ms)	
				Min	Max
Obstacle sensors	OBST $_{1..3}$	1	3	10	50
Line sensors	LINE $_{1..10}$	1	10	10	1000
Beacon	BCN_INT	1	1	200	2000
sensor	BCN_ANG	1	1	50	200
Main CPU	SPEED $_{1..2}$	1	2	10	150
Motors	DISP $_{1..2}$	2	2	20	500

Table 5.1: Message set and properties

Signal name	Tx time (μs)	# of mesgs	Period(EC)		Utilization(%)	
			Min	Max	Min	Max
OBST $_{1..3}$	650	3	1	5	3.90	19.50
LINE $_{1..10}$	650	10	1	100	0.65	65.00
BCN_INT	650	1	20	200	0.03	0.33
BCN_ANG	650	1	5	20	0.33	1.30
SPEED $_{1..2}$	650	2	1	15	0.87	13.00
DISP $_{1..2}$	750	2	2	50	0.26	6.50
Total utilization (%)					6.07	106.63

Table 5.2: Message set network utilization

and maximum network utilizations when the minimum and maximum QoS requirements are used, respectively.

Considering that an EDF scheduler is used, and applying the analysis presented in Section 4.3, the upper bound for guaranteed traffic schedulability is 73.5%. Recall that only 80% of the network bandwidth is available for synchronous traffic. This upper bound is well above the minimum required utilization but also well below the respective maximum requirement. This means that it is not possible to transmit all the messages at the respective highest rates but, on the other hand, if the lowest rates are used, there is a significant spare bandwidth. This gives room for QoS management in order to assign the spare bandwidth to specific message streams, increasing the respective QoS delivered to the application.

To better understand the use of dynamic QoS management, notice that the robot needs permanently updated information from all sensors but it executes only one behavior at a time (subsumption architecture). Therefore, the

communication system should deliver the highest QoS to the active behavior, increasing the rate of the respective messages. Conversely, inhibited or latent behaviors, may be given lower QoS levels assigning lower transmission rates for the respective messages.

For instance, whenever the robot is following a line on the ground, line sensors should be sampled at the highest rate for accurate control. Obstacle detection must still be monitored in order to avoid possible obstacles near the line but, if no near obstacles are detected, lower sampling (transmission) rates can be used. Beacon detection is not relevant in this case. If a near obstacle is detected, the robot must switch the active behavior to obstacle avoidance, assigning highest QoS to this behavior and changing the transmission rates of the respective messages accordingly.

In the following sections we will show how the QoS management policies referred before can be applied to this case.

5.3.2 Using the priority-based QoS manager

In the case of priority-based QoS management, spare resources that remain after fulfilling the minimum resource requirements are distributed among the messages following an order of decreasing QoS priority. These priorities are message parameters that reflect the respective importance in the current robot state. In this dynamic situation, the QoS priorities must also be dynamic, deduced from the actual sensor readings and taking into consideration the referred hierarchy of behaviors as referred above.

In this particular case, a specific task running in the main CPU analyzes the received sensor readings, runs the behavior arbitration to define the active behavior and generates the QoS priorities. Whenever the relative priorities change, they are supplied to the QoS manager that calculates new effective message periods and applies them to the SRT in the FTT master structure. The rules to generate these QoS priorities are straight forward: the active behavior has highest one, the remaining behaviors are given priorities proportional to the excitation level of the respective sensors. Table5.3 shows the QoS priorities that were obtained in three different situations with three different active behaviors. The table also shows the results generated by the QoS manager, i.e. the granted transmission periods for each message, as well as the total bandwidth utilization. This utilization is always close to the maximum allowed (73.5% as referred before), meaning that the system

Signal Name	Active behavior					
	Obstacle avoidance		Path following		Beacon tracking	
	QoS Priority	T_i	QoS Priority	T_i	QoS Priority	T_i
OBST $_{1..3}$	1	1	3	5	5	1
LINE $_{1..10}$	4	3	1	1	6	3
BCN_INT	4	20	5	20	4	20
BCN_ANG	4	5	5	9	1	5
SPEED $_{1..2}$	2	1	2	4	2	1
DISP $_{1..2}$	3	2	4	50	3	2
Utilization	63.29%		73.50%		63.29%	

Table 5.3: Message set utilization: priority-based QoS manager

is efficiently exploring its resources, i.e. network bandwidth in this case. The fact that the maximum utilization is not attained is due to the coarse time granularity used in the FTT paradigm (EC length), which causes step variations in the total utilization.

5.3.3 Using the Elastic Task Model QoS manager

The Elastic Task Model uses two independent parameters per message [BLA98], the nominal period and the elastic coefficient. The former ones allow to define the optimum periods within the allowable range. The latter ones define the flexibility given to the QoS manager to change the effective periods in the vicinity of the nominal ones. Again, in our case study we would like to adjust these parameters according to the instantaneous application needs or, in other words, according to the current sensor readings.

Therefore, a task running on the main CPU is also used to analyze the sensor readings, determine the active behavior and generate the QoS parameters. In this case, the generation of the parameters is done in the following way: for the active behavior, the nominal period of the respective messages is set to the minimum values, or close, and the elastic coefficient to one, or slightly higher, forcing a high QoS; for the remaining behaviors, the respective messages get a nominal period equal to the maximum values and the elastic coefficient is set proportionally to the respective sensor readings. In this latter case, when the excitation level of the sensors increases, the coef-

Signal Name	Obstacle avoidance			Path following			Beacon tracking		
	T_{i_0}	E_i	T_i	T_{i_0}	E_i	T_i	T_{i_0}	E_i	T_i
$OBST_{1..3}$	1	1	1	5	10	1	5	5	1
$LINE_{1..10}$	100	8	3	1	1	2	50	20	2
BCN_INT	100	20	20	200	20	20	30	10	50
BCN_ANG	10	20	5	20	20	10	5	1	8
$SPEED_{1..2}$	1	1	1	2	1	1	1	1	1
$DISP_{1..2}$	4	5	2	10	10	2	2	5	2
Utilization	63.29%			73.48%			73.44%		

Table 5.4: Message set network utilization: ETM QoS manager

ficients become larger thus increasing the chance of the respective behavior receiving higher QoS.

The QoS manager is invoked whenever an elastic coefficient changes. However, to reduce the number of invocations and keep the run-time overhead under adequate levels, the mapping between sensor readings and elastic coefficients should be coarse, using large quantization steps. Moreover, it is important to use some level of hysteresis in order to prevent undesired oscillations in changing from step to step.

Table 5.4 also shows three situations in which the active behavior is different. The respective QoS parameters are shown together with the effective message periods generated by the QoS manager. The overall network utilization in all three situations is close but below the maximum possible (73.5% in this case). The reason is the same as explained in the case of the priority-based QoS manager, i.e. it is due to the coarse time resolution within the FTT paradigm.

5.4 Conclusion

This chapter discusses the benefits and implications of supporting dynamic QoS management in distributed real-time systems, particularly in what concerns the communication network. Supporting dynamic QoS management requires a degree of flexibility that is not efficiently supported by existing real-time communication protocols.

Resulting from its operational flexibility, the FTT paradigm found one

of its main applications in supporting systems that benefit from, or even require, dynamic QoS management. Another strong point of the the FTT paradigm in this domain it is their ability to support arbitrary QoS management policies, as long as the QoS attributes can be mapped onto standard properties (periods, priorities or deadlines).

To illustrate how the FTT paradigm supports dynamic QoS management, this chapter also presents a simplified case study using a mobile autonomous robot. Two possible QoS management policies are briefly presented, one that is priority-based and the other based on the elastic task model, and it is shown how they can be used in the scope of the FTT paradigm. The results obtained confirm that using the FTT paradigm in distributed real-time applications can lead to efficiency gains in network bandwidth that arise from the support to dynamic QoS management policies.

Chapter 6

Contributions to FTT-CAN

The FTT-CAN protocol aims mainly real-time applications based on low processing-power micro-controllers, typically found in distributed embedded systems [ZPS99]. Due to the constraints presented by this framework, namely concerning the limited resources available (network bandwidth, CPU processing power, memory), the implementation of the FTT-CAN protocol was biased towards simplicity and resource economy. Moreover, some techniques have been specifically developed to reduce the protocol overhead, like the use of a planning scheduler [AF98] in the master node. Nevertheless, both the system architecture, functionality and application interface of the FTT paradigm have been preserved.

6.1 The FTT-CAN Elementary Cycle

The FTT-CAN elementary cycle structure is similar to the generic EC structure described in Section 4.2.2, except that the asynchronous window precedes the synchronous one (Figure 6.1). The reason that has motivated this decision is related with the need to decode the EC-Schedule carried by the trigger message before nodes can start to transmit their respective synchronous messages. Decoding the EC-Schedule and scanning the local tables to identify what synchronous messages should be produced in the respective EC takes an amount of time that strongly depends on the node processor capacity, and can be as large as the transmission time of one or more messages when simple 8-bit micro-controllers are used [Alm99, PA00]. Thus, if the synchronous window was defined right after the TM, the gap between

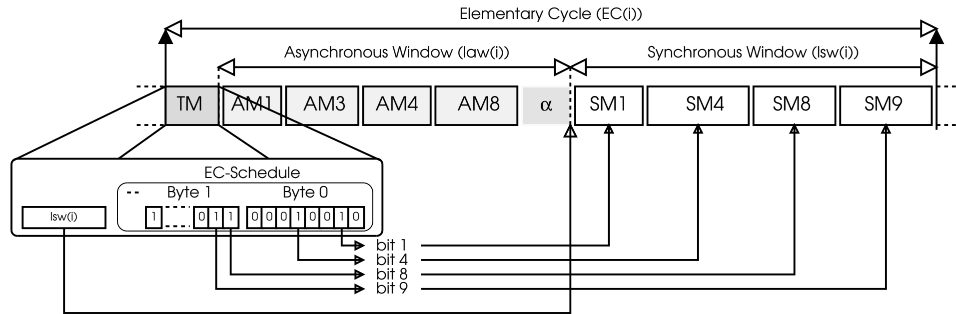


Figure 6.1: FTT-CAN Elementary Cycle

this message and the first synchronous message would be hardware dependent and the corresponding bus time would be wasted. On the other hand, asynchronous traffic transmission is considerably less demanding, since just consists in getting data from a queue. Moreover, this process can be started during the transmission of the TM, because the EC-Schedule is relevant only for the synchronous messages, resulting in a synchronized start of transmission of all the pending asynchronous messages. This aspect is particularly important, since in this case the arbitration of the pending asynchronous messages is performed in strict priority order, which is a fundamental requirement of the schedulability analysis presented in Section 4.4.

6.1.1 Message Arbitration

The FTT-CAN protocol relies heavily on the deterministic CAN arbitration mechanism (Section 3.2.1) to reduce the overhead required by its operation. Concerning the synchronous traffic, the trigger message only needs to convey the identification of the synchronous messages that should be produced in the EC and the duration of the synchronous window (Figure 6.1). Using this information, each node identifies which messages it should produce and starts their transmission at the beginning of the synchronous window. Several nodes can submit messages for transmission at the same time and the CAN MAC automatically serializes their transmission. The same situation occurs in the asynchronous window; nodes having asynchronous messages queued enable their transmission at the beginning of the asynchronous window (actually during the transmission of the TM), and the CAN MAC serializes them in strict priority order as specified by the message's identifiers.

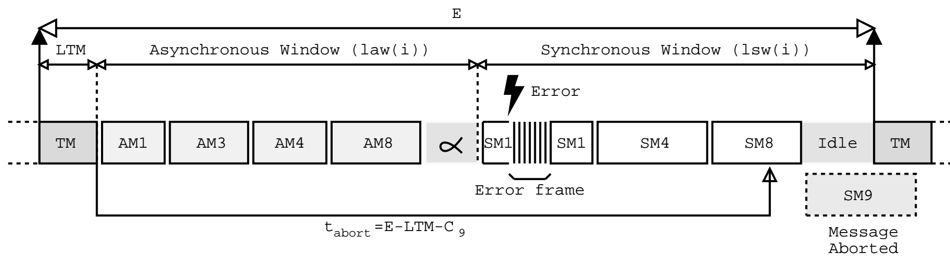


Figure 6.2: Preventing synchronous window overrun

6.1.2 Enforcing temporal isolation

In order to maintain the temporal properties of the traffic, both synchronous and asynchronous messages should be confined within their respective windows, enforcing a strict temporal isolation between both phases. This is achieved by preventing the start of message transmissions that could not complete within their respective window.

With respect to the synchronous traffic, under normal circumstances the synchronous messages scheduled for transmission should fit within their respective window. However, in case of errors CAN controllers automatically retransmit the affected messages, and thus if no further actions are taken transmissions may extend over the duration of the synchronous window. To avoid this phenomenon, upon reception the TM all nodes set a timer with the latest instant where a message can start to be transmitted and still finish within the synchronous window ($t_{abort} = E - LTM - C_i$), as depicted in Figure 6.2.

When this timer expires, nodes check the transmit status register of the CAN controller, and, if the message is still waiting for transmission issue an abort command, thus preventing the start of the transmission of the message that otherwise would extend over the following EC. With this mechanism synchronous messages are confined to the synchronous window, even in the presence of errors.

When nodes are producers of several messages, maintaining a timer per message can result in a considerable overhead. To overcome this situation, nodes can use a single timer, set with the time associated with the transmission time of the longest synchronous message produced by the node itself. The schedulability is reduced, but the overhead can become significantly

lower.

Concerning the asynchronous traffic, nodes having asynchronous messages queued try to transmit them without any knowledge about the state of the remaining nodes. Therefore there is no guarantees that the set of ready messages among all system nodes will fit within one asynchronous window. Under these circumstances it becomes mandatory to confine the asynchronous messages into the asynchronous window, suspending their transmission outside those periods of time. This is achieved by removing from the network controller transmission buffer any pending request that cannot be served up to completion within that interval, keeping it in the transmission queue. When nodes queue an asynchronous message for transmission they also set a timer with the latest allowed start instant. Since the asynchronous window length is deduced from the synchronous window, and the length of the synchronous window is carried in the trigger message, the abort instant for message AM_i can be computed as $t_{abort} = E - LTM - lsw - C_i$. As for the case of the synchronous traffic, to reduce the overhead associated with the timer management, nodes can use a single timer, set in this case with the size of the longest asynchronous message originated in the node.

6.1.3 FTT-CAN message types

The FTT-CAN protocol defines the following message types:

- EC Trigger Message [TM_MESG_ID];
- Synchronous Data Messages [DATA_MESG_ID];
- Asynchronous Data Messages [AM_DATA_MESG_ID];
- Control Messages [CONTROL_MESG_ID];

The four most significant bits of the CAN ID field [ID.b10...ID.b7] are used to define the particular message type, as depicted in Table 6.1.

The contents of the TM is depicted in Table 6.2.

The **Type** field contains the MST_MESG_ID, identifying the TM. **The Master ID** field allows the existence of up to 8 different masters in the network. In case of failure of the active master, an election mechanism protocol (Section 6.5) selects one of the backup masters to become the new active master. The **New Plan** field is used to signal the start of a new plan when

0 [Synch]	0 [Master]	00	TM_MESG_ID
	1 [Slave]	10	DATA_MESG_ID
1 [Asynch]	000		CONTROL_MESG_ID (HP)
	100		AM_DATA_MESG_ID (RT)
	110		CONTROL_MESG_ID (LP)
	111		AM_DATA_MESG_ID (NRT)

Table 6.1: Message type identification

Type	Master ID	New Plan	Sequence Number	Synchronous Window Len.	EC Schedule
CAN ID field				CAN Data field	
[b10..b7]	[b6..b4]	b3	[b2..b0]	MSB	1 to 7 bytes
TM_MESG_ID	0 to 7	{0,1}	0 to 7	0 to 255	Bitmap

Table 6.2: EC Trigger Message structure

a planning scheduler is used (Section 6.4). The **Sequence Number** field is incremented by the active master in each EC and allows the detection of up to 8 consecutive trigger message omissions. The **Synchronous Window Length** field contains the duration of the synchronous window in the current EC, with a resolution of $\frac{LSW}{255}$. Finally, the **EC-Schedule** field indicates which synchronous messages should be produced in the EC, encoded in a bitmap. Each synchronous data message is associated with a particular bit. The mapping of message ID in the bitmap field if performed in ascending order, right to left ($SM_0 \leftrightarrow bit_0; SM_1 \leftrightarrow bit_1 ... SM_{N_S} \leftrightarrow bit_{NS}$), for all N_S synchronous messages.

Recalling that CAN frames are subject to bit-stuffing, Equation 3.2 can be adapted to compute the maximum number of bits required by the trigger message, as follows (Equation 6.1):

$$LTM_{bits} = (2 + \left\lfloor \frac{N_S - 1}{8} \right\rfloor) * 8 + 47 +$$

TX rate *(Mbps)*	LTM *(byte / #mesgs)*	LTM μs	E *(ms)*	E *(%)*
0.125	5/32	854	10	8.54
0.125	8/56	1098	10	10.98
1.000	5/32	105	5	2.10
1.000	8/56	135	5	2.70

Table 6.3: Communication overhead imposed by the EC Trigger Message

Type	TX_ND	Message ID	Message Data
CAN ID field			CAN Data field
[b10..b7]	b6	[b5..b0]	0 to 8 bytes
DATA_MESG_ID	{0,1}	0 to 64	Application specific

Table 6.4: Synchronous Data Message structure

$$+ \left\lfloor \frac{34 + (2 + \left\lfloor \frac{N_S - 1}{8} \right\rfloor) * 8 - 1}{4} \right\rfloor \quad , \quad 1 \leq N_S \leq 56 \tag{6.1}$$

By knowing the maximum number of synchronous messages allowed in a particular system (N_S) and the transmission speed (TX_{RATE}), the the worst-case time required to transmit the TM is given by:

$$LTM = \frac{LTM_{bits}}{TX_{RATE}} \tag{6.2}$$

As stated in Section 4.2.1, the use of the master/multi-slave transmission control, in which one single TM triggers the transmission of several data messages in distinct nodes, allows to considerably reduce the protocol overhead when compared with a pure master-slave transmission control. Table 6.3 presents the overhead due to the transmission of the TM in FTT-CAN in four typical scenarios. Note that this overhead can be further reduced by using a higher value for the EC length or by reducing the data length of the TM whenever the applications require fewer synchronous messages.

Synchronous Data Messages are used to periodically distribute state data among the network nodes, and are always transmitted within the synchronous window, when indicated in the EC-Schedule carried by the TM. The synchronous data message structure is depicted in Table 6.4.

Type	Not used	Message ID	Message Data
CAN ID field			CAN Data field
[b10..b7]	b6	[b5..b0]	0 to 8 bytes
AM_DATA_MESG_ID ({RT,NRT})	—	0 to 64	Application specific

Table 6.5: Asynchronous Data Message structure

The **Type** field contains the DATA_MESG_ID constant indicating that the frame is a synchronous data frame. The transmit new data flag (TX_ND) allows to implement a lighter version of the temporal validity information described in Section 4.2.4. The TX_ND flag, if set, indicates that the source node has updated its local image of the respective real-time entity after the last transmission. Conversely, if this bit is not set, it means that the application had not updated the local image, and thus the contents of the message is the same as the one in its last instance. A full description of this mechanism can be found in [Alm99, APF02]. The **Message ID** field allows to identify each of the messages. Finally, the **Message Data** field contains up to 8 bytes of payload data.

Asynchronous Data Messages are used to convey event information, are sent after application explicit request, and are transmitted within the asynchronous window. The structure of a these frames is depicted in Table 6.5.

The structure of this frame is similar to the synchronous data message frame, except that in this case there is no transmit new data flag, due to the event nature of these messages.

There are two levels of priority associated with asynchronous data messages (Table 6.1) which map into two different traffic classes. Higher priority (**RT**) asynchronous messages are subject to real-time constraints, and thus appropriate analysis (Section 4.4) can be performed in order to compute in advance if its timeliness requirements can be met, thus they pertain to the asynchronous real-time traffic class. However such analysis does not involve the low priority (**NRT**) asynchronous messages, which are handled according to a best-effort policy (Section 4.4). Thus, low priority asynchronous messages fall into the non-real-time asynchronous traffic class.

Asynchronous Control messages are used to perform system management (e.g master synchronization data, software download, requests for SRT

Type	Not used	Message ID	Message Data
CAN ID field			CAN Data field
[b10..b7]	b6	[b5..b0]	0 to 8 bytes
CONTROL_MESG_ID ({HP,LP})	—	0 to 64	Application specific

Table 6.6: Control Message structure

changes, non-real-time message polling,etc.). The internal structure of this type of frame is similar to the structure of asynchronous data messages and is depicted in Table 6.6.

There are two priority levels assigned to control messages. The high-priority messages (**HP**) have the highest priority among all the asynchronous messages (Table 6.1) and are used for time-critical management operations, such as urgent SRT change requests. The lower priority (**LP**) control messages have the lower priority among all the asynchronous messages. These are used to carry operations that are not time constrained, such as remote diagnosis or software updates.

The maximum number of bits required by both synchronous, asynchronous and control messages is given directly by Equation 3.2 and their respective transmission time computed as in Equation 6.2.

6.2 Synchronous traffic

The generic schedulability analysis for the FTT message model has been introduced in Sections 4.3 and 4.4, concerning respectively synchronous and asynchronous traffic. This section addresses the adaptations concerning the synchronous traffic.

6.2.1 Schedulability analysis

The schedulability tests presented in Section 4.3 can be directly applied to the FTT-CAN protocol. It should be recall that the analysis requires the use of worst-case transmission times. Therefore, in the definition of the synchronous requirements table (Equation 4.1) the message transmission time (C_i) must be derived from the number of data bytes (DLC_i) using Equation 3.2 to compute the maximum number of bits and then Equation

6.2 to compute the corresponding worst-case transmission time.

6.2.2 Experimental results

The FTT-CAN protocol inherits from the FTT paradigm the possibility of using of arbitrary scheduling policies (Section 4.2.1). The scheduling is carried out based on the SRT independently of the message identifiers. Thus, any scheduling policy can be easily implemented, e.g. Rate-Monotonic (RM), Deadline-Monotonic (DM), Earliest-Deadline First (EDF), Least-Laxity First (LLF), overriding the identifier-based traffic scheduling embedded in the MAC of CAN.

The possibility of implementing more efficient scheduling policies can be particularly relevant for heavily loaded systems, because different scheduling paradigms allow obtaining different temporal behaviors and different bus utilization factors. For example, in the work of Liu & Layland [LL73] it is shown that EDF allows full CPU utilization with independent preemptive tasks, whilst for RM the upper bound for guaranteed timeliness can be as low as 69%. While the previous limit represents the worst-case for RM, a simulation study carried out by Lehoczky, Sha and Ding [LSD89] with random task sets showed that RM is able to achieve on average an utilization as high as 88%.

In the specific context of message scheduling certain constraints must be accounted for, resulting in lower utilization bounds. For example, in the particular case of fieldbuses, such as the CAN bus, messages are transmitted without interruption and consequently must be scheduled non-preemptively. Nevertheless, the relative difference between the schedulability levels of EDF and RM scheduling still holds. Particularly for the CAN bus, some comparative results between RM and EDF using realistic loads [ZS97] show a difference around 20% in network utilization in favor of EDF.

To assess the advantages of using EDF in the scope of FTT-CAN with respect to the level of schedulability and system overhead, a set of simulations and experiments were carried out. The target hardware test platform is a CANivete system [F$^+$98] based on the Philips 80C592 clocked at 11.059MHz with the CAN interface configured to run at 123Kb/s. The system architecture is depicted in Figure 6.3.

As discussed in Section 6.1 the decoding of the EC-Schedule and SRT scanning requires an amount of time that is strongly dependent on the pro-

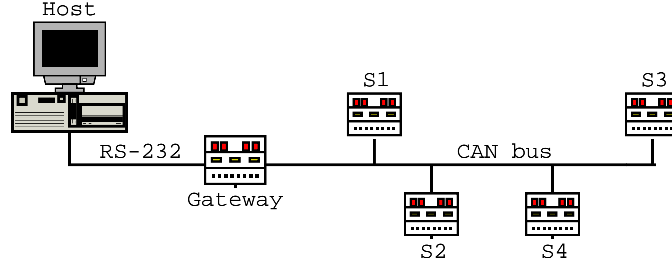


Figure 6.3: Experimental set-up

cessing power within the nodes. For the hardware platform described above this overhead ($POVRHEAD$) has been experimentally measured, and an upper bound of 1ms (roughly 120 bits at 123Kb/s) was found. This bus time cannot be used by synchronous traffic, thus the maximum duration of the synchronous window (LSW) can be computed by Equation 6.3.

$$LSW = LEC - (LTM + POVRHEAD + LAW) \tag{6.3}$$

In order to assess the actual difference in scheduling capability between RM and EDF in FTT-CAN, a simulation with 10.000 random messages sets was performed. Each set contains 32 messages respecting the following constraints:

- 5 messages with period 1 EC;
- 10 messages with period between 3 and 6 ECs uniformly distributed;
- 17 messages with period between 10 and 16 ECs uniformly distributed;
- Data length: 1..8 bytes uniformly distributed;
- IDs are ordered by increasing period.

The purpose of using this pattern is to obtain sets with high network utilization and with messages of three different categories concerning the respective transmission periods: short, medium and long.

Considering the maximum number of 32 synchronous messages ($N_S = 32$) used in the simulations, the maximum number of bits required by the TM and its corresponding worst-case transmission-time (Equations 6.1 and 6.2) become respectively:

$$LTM_{bits} = 105\,bit \tag{6.4}$$

$$LTM = 0.854ms \tag{6.5}$$

Considering that no further bandwidth is reserved for asynchronous traffic except the one due to the processing overheads (i.e. *LAW=0, POVRHEAD=1ms*), an EC duration of $8.9ms$ and a transmission rate of $123Kb/s$, the maximum length of the synchronous window is:

$$LSW = 8.9 - (0.854 + 1 + 0) = 7.046ms \tag{6.6}$$

For the message set herein considered, an absolute upper bound for the inserted idle-time ($X = \max_n(X_n)$) results from a message with eight data bytes, resulting in:

$$X_{bits} = 135\,bit \tag{6.7}$$

$$X = 1.098ms \tag{6.8}$$

The least upper bound of bus utilization for RM (U_{lub_RM}) and EDF (U_{lub_EDF}) scheduling policies can now be computed using Conditions 4.16 and 4.18.

$$U_{lub_RM} = 46.8\% \tag{6.9}$$

$$U_{lub_EDF} = 66.8\% \tag{6.10}$$

These values are lower than the typical values for preemptive task scheduling as presented in [LL73]. This is expected since such values do not consider the impact of inserted idle-time neither any kind of protocol or processing overhead. For the values above it can be observed a difference in scheduling capability under guaranteed timeliness of near 20% in favor of EDF.

However, as it can be observed in Figure 6.4 the percentage of schedulable sets obtained in the simulation is substantially higher than the least-upper bounds derived above, both for RM and EDF. In fact, all sets in the simulation with utilization factor up to 71% are schedulable both by RM and EDF,

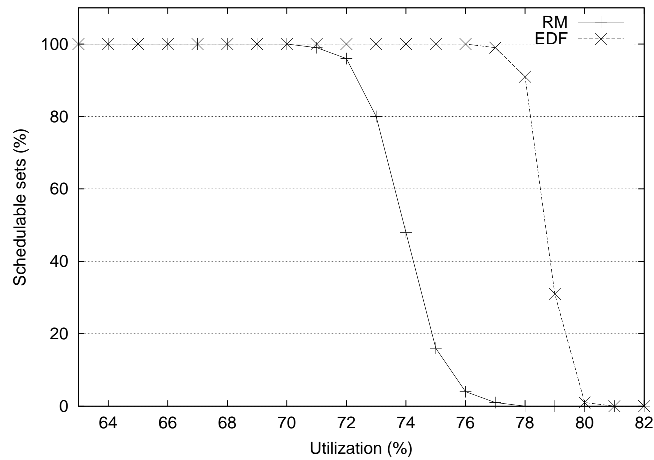


Figure 6.4: Schedulability versus bus utilization under RM and EDF

and those with utilization up to 77% are schedulable by EDF, only. These results also show that the least upper bound for RM stated in Condition 4.16 is more pessimistic than the one for EDF presented in Condition 4.18. This situation is also expected since the original bound for RM preemptive scheduling is also more pessimistic than the one for EDF. It is also important to recall that, due to the transmission of the EC trigger message and to the processing overhead specific of the infrastructure used, only 80% of the bus bandwidth is available for the synchronous messages. Notice that, as expected, EDF practically allows fully utilization of this bandwidth.

To have a measure of the relative performance of FTT-CAN in the support of EDF scheduling, it was carried a brief review of the related work. Other methodologies for implementing EDF scheduling on CAN [ZS95, Nat00, LK98] relied exclusively in the native MAC of the protocol. Since the priority of the messages depends on the identifier bits and priorities in EDF are dynamic, this approach implies dividing the identifier in at least two fields, one to encode the priority (variable) and another to identify the message itself (fixed). In [ZS95, Nat00, LK98] several techniques for managing the priority field are discussed, which consider the restriction of using a limited number of identifier bits as well as the need to keep the processor overhead in acceptable levels.

In [ZS95] it is proposed a solution based on the encoding of absolute deadlines relative to a periodically increasing time reference designated epoch.

However, this solution has difficulties in dealing with message sets containing periods orders of magnitude apart. In this case either it is used a coarse time granularity, leading to a large number of priority inversions, or the number of bits used to encode the deadline is increased, reducing the number of distinct messages that can be scheduled. A particular technique is presented, named Mixed Traffic Scheduling, according to which the traffic is first scheduled by EDF, using the priority field, and then by fixed priorities using the message identifier field. Nevertheless, this leads to a reduction in the benefits of using EDF.

In [Nat00] the author proposes to encode the time to the absolute deadline (therein referred to as slack) in a logarithmic time scale, increasing the temporal resolution as deadlines are approached and thus, reducing the number of possible priority inversions for early deadlines. A consequence of this technique is that the identifier bits, used to encode the priority of the messages waiting for transmission, must be updated each time messages compete for the bus access after it becomes idle (referred to as arbitration round).

In [LK98] the authors encode the time to the absolute deadline in a linear time scale, but using extended frames (ID field with 29 bits, CAN 2.0 B). In this approach, the IDs of the messages waiting for transmission must also be updated before each arbitration round. Although this technique allows for larger ranges of periods and deadlines, the additional number of bits required by the ID field (20 bits, including stuffing) spoils a significant part of the additional bandwidth that is made available by using EDF, since the increased ID field length in CAN 2.0B [Rob91] requires between 13% to 40% more bandwidth than version A.

Major drawbacks shared by all these approaches can be summarized as follows:

- Reduction on the number of supported messages due to the use of some identification bits to encode the priority;
- All nodes must periodically update the priority field, resulting in a non-negligible processing overhead;
- Priority inversions induced by the limited resolution available to express deadlines;
- Global clock synchronization required, further consuming CPU and

network bandwidth.

As opposed to these approaches to EDF message scheduling on CAN, in the FTT-CAN protocol all the scheduling decisions are performed in the Master node. Consequently, most of the drawbacks presented above do not hold. Firstly, in FTT-CAN the priority, i.e. time to the deadline in the case of EDF, is held in a variable within a data structure and no identifier bits are used to encode it. Thus, no reduction is imposed on the number of messages, besides the field reserved for message type definition (Section 6.1.3). Secondly, the scheduling activity is confined to the Master. The EC trigger message identifies the synchronous messages that must be produced in each EC. All other nodes follow a slave-like operation that is completely independent from the scheduling technique used by the Master. Thus, the use of EDF does not impose any extra computational activity in any node beyond the Master. Thirdly, the SRT is maintained in an adequate structure in the Master memory. Message parameters, such as periods and deadlines, are held within variables which type can be adequately chosen to support the required range of values. Thus, the range of periods that can be handled within FTT-CAN is virtually unlimited, beyond the constraint of being integer multiples of the EC duration, although there is a clear impact in memory requirements and processing overhead. Finally, all nodes are synchronized by the EC trigger message and there is no need for global clock synchronization.

To assess the performance of the FTT-CAN approach compared with the other methodologies above referred, it was carried a simulation study in similar conditions. The simulation results presented in [ZS95] are not very interesting because they are based on 10Mbps CAN network, which is not realistic. On the other hand, the methodology presented in [LK98] uses CAN 2.0 B and thus a direct comparison would not be possible. Therefore the comparison was carried only with respect to the methodology presented in [Nat00]. The workload consists in:

- Random message sets with 30 messages grouped in 3 distinct categories according to their periods (ms), [3,12], [30,120] and [250,1000];
- Deadline to period ratio is in the range [0.8,1.0] uniformly distributed;
- CAN bus at 250Kbps.

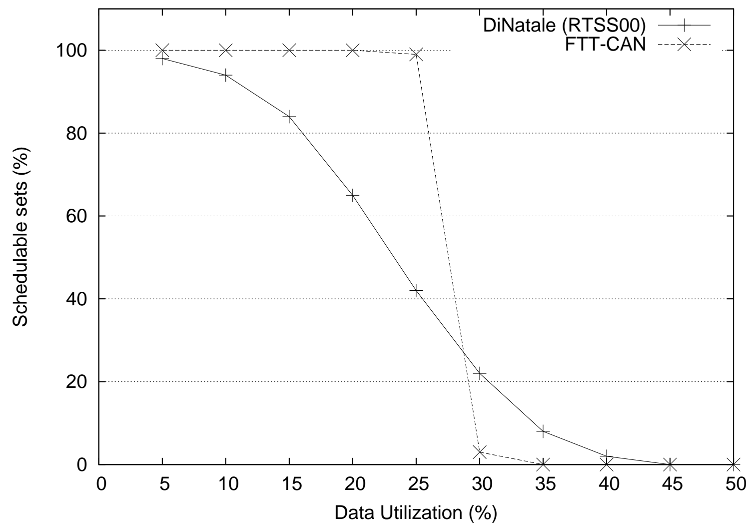


Figure 6.5: Percentage of schedulable message set using EDF scheduling on CAN

The results obtained are plotted in Figure 6.5. For each point in the plot 5000 random sets were generated, giving a total of 60000 message sets. To allow an easier comparison with [Nat00], the x-axis shows the effective data utilization, i.e. equivalent transmission time of data bits only, over message period.

The results in Figure 6.5 are roughly similar to those presented in [Nat00], but the curve is more abrupt with FTT-CAN, presenting a larger level of schedulability for a wide range of data utilization values. Hence, the FTT-CAN based EDF implementation is able to achieve a comparable or even better data throughput despite the use of a centralized approach and simple micro-controllers in the nodes beyond the Master.

The advantages of using FTT-CAN to support EDF scheduling on CAN are summarized below:

1. Simplicity of scheduler implementation in the Master node. Furthermore, the scheduling policy can easily be changed on-line, e.g. during transient overloads.

2. Message scheduling separated from the MAC arbitration, avoiding the undesirable compromise between dynamic priorities and message identifiers.

3. CPU load required by EDF scheduling confined to the Master. Remaining nodes require a constant CPU load to decode the EC trigger message, whichever is the scheduling policy being used.

4. Support for virtually unlimited range of message's periods and deadlines simply by using appropriate types for the respective variables.

5. Explicit global clock synchronization is not required, thus further saving network and CPU load in all nodes.

On other hand, in FTT-CAN there is also a limitation imposed on the temporal resolution. In fact, in FTT-CAN all periods and deadlines are expressed as integer multiples of the EC duration and a sub-EC resolution is not supported. Within the EC, messages are scheduled according to the fixed priority that corresponds to the respective CAN identifiers. This limitation, nevertheless, does not seem to be particularly relevant since for typical applications (e.g. automotive, machine tool control) the shortest deadlines and periods lie in the range from 1ms to 10ms, which is the same magnitude of the envisaged EC duration in FTT-CAN systems. On other hand, FTT-CAN is able to schedule with EDF only the synchronous traffic, while the other approaches above referred can handle asynchronous (event) traffic.

6.3 Asynchronous traffic

6.3.1 Schedulability analysis

The asynchronous traffic schedulability analysis presented in Section 4.4 for the generic FTT paradigm is applicable to the FTT-CAN implementation. The only modification that must be performed concerns the swap in the relative positions between the synchronous and asynchronous windows, which implies and adaptation of the time intervals in Equation 4.25, resulting in Equation 6.11. Moreover, the analysis also requires the message scheduling to be performed in strict priority order. This is automatically provided by the CAN MAC, since the message IDs are set according to the desired message priority.

$$A(t) = \begin{cases} \sum_{j=1}^{k-1} (law(j) - \alpha_j)\,, \\ \quad t:\, (k-1)*E \le t < (k-1)*E + LTM \\ \\ \sum_{j=1}^{k-1} (law(j) - \alpha_j) + t - (k-1)*E, \\ \quad t:\, (k-1)*E + LTM \le t < k*E - (law(k) + \alpha_k) \\ \\ \sum_{j=1}^{k} (law(j) - \alpha_j)\,, \\ \quad t:\, k*E - (law(k) + \alpha_k) \le t < k*E \\ \qquad with\, k-1 = \left\lfloor \frac{t}{E} \right\rfloor \end{cases} \tag{6.11}$$

6.3.2 Experimental results

This section presents the results of two experiments conducted with the purpose of testing and assessing the behavior of the FTT-CAN Asynchronous Messaging System, concerning both AT1 and AT2 classes of asynchronous real-time messages presented in Section 4.4.

The experiences were performed on the CANivete system [F+98] described in Section 6.2.2. The CAN bus transmission rate used in the experiments is approximately 123Kbps, and the EC duration is set to 8.9ms. The time measurements were carried using one of the processor's internal timers, which supplies a resolution about $1\mu s$.

The sets of messages used are derived from "PSA Peugeot Citroen" CAN message set, with some customization in the message properties (length and period/minimum inter-arrival time) to generate an adequate bus utilization. The synchronous load is the same in both experiments and is described in Table 6.7. The asynchronous message set for each of the experiments is described in Table 6.8.

In the experimental set-up, all the asynchronous messages are produced at their maximum rate, and their transmission is requested just after the end of the asynchronous window of the EC, in an effort to achieve a scenario close to the worst-case one used in the analysis. One thousand transmission/reception events have been recorded for each message.

The first set of messages produced the results presented in Table 6.9. Concerning the analysis data (two rightmost columns on Table 6.9), it can be observed that messages with ID 7 and 8 are guaranteed to be schedulable

Message ID	Number of Data Bytes	Period (ECs)
1	1	1
2	3	1
3	3	2
4	2	1
5	5	2
6	5	4

Table 6.7: Synchronous communication requirements

Message ID	Number of Data Bytes	mit (ECs) [Experiment 1]	mit (ECs) [Experiment 2]
7	4	1	1
8	5	1	1
9	4	1	1
10	7	1	2
11	5	1	2
12	1	1	2

Table 6.8: Asynchronous communication requirements

within their minimum inter-arrival time. Message 9 starts to be transmitted before the arrival of its next instance, but finishes its transmission after, therefore, only one transmission buffer is required to handle it. All instances of message 10 can be transmitted if at least three transmit buffers are provided. Messages 11 and 12 are not guaranteed to be schedulable.

Since the analysis is based in worst-case assumptions, it can be expected that experimental results are in some extent better than analytic ones. Comparing the response time (columns 4 and 6 of Table 6.9) it can be observed that the maximum measured response time is always lower than the one computed. Also, in practice only one buffer for message 10 is used, and all instances of message 11 are schedulable if two transmission buffers are provided. The differences between analytical and experimental results are due to difficulties in reproducing worst-case conditions in the experimental set-up. Two factors are particularly relevant to explain the differences observed:

- variable amount of stuff bits, which can lead to messages being about 20% shorter than the worst-case length considered in the analysis;

Mesg ID	Experimental Data				Analytic results	
	Response time (μs)			# buffers	Resp. time (***) (μs)	# buffers
	Min	Avg	Max			
7	3714	5073	6997	1	7884	1
8	3976	5668	7490	1	8684	1
9	5367	6388	8063	1	9444	1
10	5962	6971	8641	1	27684	3
11	6720	10381	15843	2	**	**
12	*	*	*	*	**	**

(*) Cannot be computed due to lost messages

(**) Cannot be computed since the analysis does not guarantee schedulability

(***) Time to transmit all queued instances of the message

Table 6.9: Results from experiment 1

- inserted idle-time shorter than the worst-case value considered in the analysis, which was used to simplify it. The impact of this factor would be reduced by using a longer EC with a longer asynchronous window.

Table 6.10 shows the results obtained with the second set of asynchronous messages.

It can be observed in Table 6.10 that the analysis guarantees the schedulability within the minimum inter-arrival time of messages seven, eight and nine. All instances of messages 10,11 and 12 are also guaranteed to be

Message ID	Experimental Data				Analytic results	
	Response time (μs)			# buffers	Resp. time (***) (μs)	# buffers
	Min	Avg	Max			
7	4142	5199	7465	1	7844	1
8	4139	5752	7256	1	8684	1
9	5263	6504	8058	1	9444	1
10	6135	7081	8422	1	26648	2
11	7727	8718	9611	1	38180	4
12	8709	9228	10800	1	71348	4

(***) Time to transmit all queued instances of the message

Table 6.10: Results from experiment 2

schedulable if enough transmit buffers are provided (2, 4 and 4 respectively). In the experiment it was verified that all the messages were scheduled within the respective minimum inter-arrival time, therefore there was no lost messages, and only one transmission buffer was used. As stated before, this fact can be explained by the worst-case assumptions made in the analysis.

From the comparison between the experimental and analytical results it can be concluded that, on one hand, the measured values were always within the range predicted by the analysis, and, on the other hand, analytic response time bounds derived for the real-time asynchronous messages are, as expected, pessimistic. A more exact bound for the inserted idle-time could reduce the degree of pessimism of the analysis, but would require an higher computational overhead (Section 4.4.1). However, the major source of pessimism in the analysis is due to the CAN bit-stuffing and cannot be avoided, because the message length depends on the data to be transmitted, which of course cannot be foreseen.

6.4 Using a Planning Scheduler

As described in Section 4.2.1, during run-time an on-line scheduler builds the EC-Schedules for each EC, based on the actual requirements of the synchronous traffic, specified in the SRT. These schedules are then inserted in the data area of the respective EC trigger message and broadcast with it. Due to the on-line nature of the scheduling function, changes performed in the SRT at run-time will be reflected in the bus traffic within a bounded delay.

However, scheduling is on one hand a costly activity in terms of processing requirements and on the other hand a critical activity, since failing to build an EC-Schedule in time (i.e., before the beginning of the following EC) results in an interruption on all the communication activities. For systems based on low computational capacity nodes (e.g., based on simple 8 bit micro-controllers) the processing demand required by the scheduler can be beyond the capacity of the master's CPU.

To overcome this situation, two different solutions have been developed to implement the scheduler. One is the planning scheduler [APF99, Alm99], a software-based implementation that allows reducing the processing overhead of on-line scheduling. This technique consists on building a static schedule

table for a given period of time into the future, called plan, and rebuilding that table on-line at the end of each plan. The plan duration is not correlated with the number of synchronous messages or its periods, therefore the memory resources used by this structure are bounded and can be set-up a priori. Previous work [Alm99] on this subject has shown that for the case of Rate Monotonic, the scheduler overhead is inversely proportional to the plan length. Therefore, managing the plan length allows to, up to a certain extent, trading memory by CPU usage.

The second solution that has been developed to implement the scheduling function in FTT-CAN makes use of FPGA-based scheduling co-processors. This solution provides, at a higher hardware cost, the extra computational capacity required to execute both the scheduling and schedulability analysis on-line. For example, the co-processor described in [MF01] scans the SRT and creates a new EC-Schedule every EC. Moreover, it is also capable of executing several schedulability tests in that interval. The result of this solution is a high degree of flexibility and responsiveness, plus a residual computational overhead, only, in the master processor, which allows the use of less powerful, and thus more economic, micro-controllers.

Although the use of a scheduling co-processor seems more interesting, it implies a cost penalty, particularly when dependability issues call for the use of master replication (Section 6.5). Therefore, from the economic point of view, the use of a software-based solution seems more adequate. However, the use of a planning scheduler limits in some extent the system flexibility, due to the static nature of the plans. Changes on the synchronous communication requirements are considered by the scheduler in a per plan basis instead of a per EC basis. Thus, the time required by a change request to take effect on the communication network takes more time, situation that raises a conflict between the need to use longer plans, to reduce the scheduling overhead, and the need to use shorter plans, to have shorter response times to changes to the communication requirements.

6.4.1 Responsiveness limits

Once a change request is made concerning the current synchronous message set, a certain period of time elapses until that request takes effect at the bus level. This time interval is referred to as the synchronous transient response time ($STRT$).

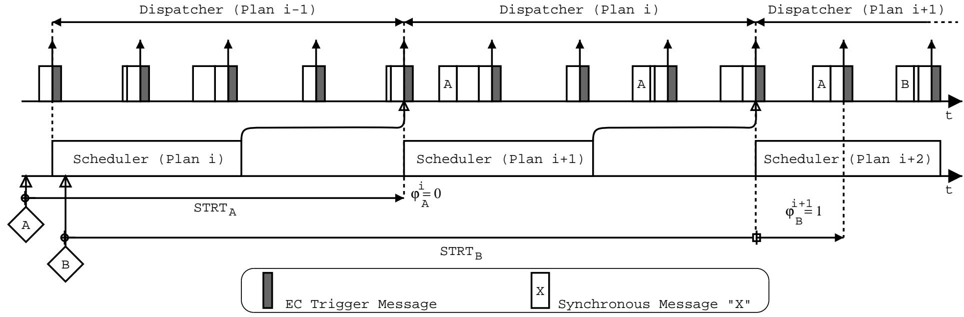


Figure 6.6: SMS Responsiveness bounds

The STRT can be decomposed in three parts (Figure 6.6):

- the time from the request to the end of the current plan;
- the plan in which the scheduler takes into account the new requirements;
- the initial phase (φ) of the message stream relative to the beginning of the plan where changes are already reflected. Note that $\forall i$, $\varphi_i \leq P_i$.

The minimum value (marker A in Figure 6.6) occurs when cumulatively the request is made just before the end of one plan, and φ is zero. The maximum value occurs if the request is issued just after the beginning of one plan (marker B in Figure 6.6), and the initial phase has its maximum value. Therefore, the absolute bound for the synchronous transient response time, when using the SMS alone ($STRT_{SMS}$), varies between one and two plans plus the initial phase (as defined above).

$$LPlan \leq STRT_{SMS} \leq 2 * LPlan + \varphi \tag{6.12}$$

Since the $STRT_{SMS}$ is a direct function of the plan duration, the responsiveness can be improved by shortening the plan. However, the reduction of the plan duration increases the CPU load [AFF99, Alm99]. Below a given value, the scheduler might not have enough time to build next plan in time, that is, before the dispatcher processes the current one. Moreover, some interesting properties of the planning scheduler, like the look-ahead feature [Alm99], are negatively affected by the reduction of the plan length. As

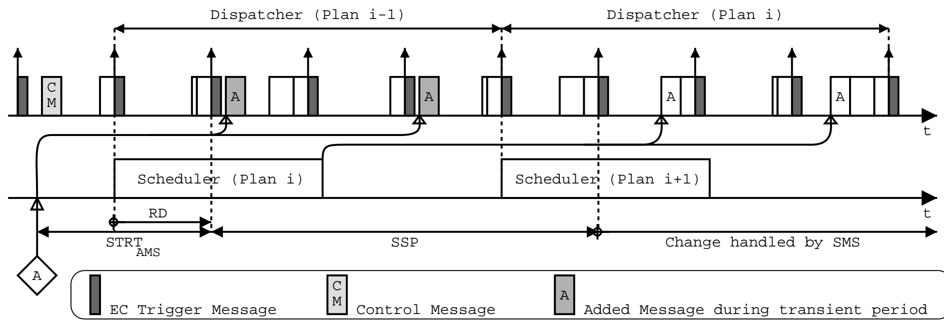


Figure 6.7: Using the AMS to temporarily convey a new synchronous message

a consequence, there is a lower bound to the plan duration, limiting the responsiveness that can be achieved this way.

Another way to improve the responsiveness is to start the scheduler as late as possible. Since the worst case execution time of the scheduler (*wcetSch*) can be estimated on-line [Alm99], using this approach the synchronous transient response time can be bounded to the interval indicated in Equation 6.13.

$$wcetSch \leq STRT_{SMS} \leq LPlan + wcetSch + \varphi \quad (6.13)$$
$$LPlan : \; Plan\, duration$$

6.4.2 Improving the responsiveness

As seen above, the responsiveness of the SMS, when a Planning Scheduler is used, is upper bounded by the plan duration plus the scheduler execution time. Since these cannot be made arbitrarily short, further improvement to the responsiveness of SMS in FTT-CAN requires that change requests should be handled even during the current plan, bypassing the planning scheduler for a short period of time, but without disturbing the other synchronous messages already scheduled.

To achieve this purpose the asynchronous messaging system (AMS) can be used to produce the required message(s) until the requested changes are handled by the SMS, as described in the previous section and depicted in Figure 6.7. After the dispatcher starts processing the plan in which the

new message parameters are reflected (Plan i in Figure 6.7), the system resumes normal operation, that is, as the message is included in the SMS it is removed from the AMS. The period of time during which the AMS is used to support the transmission of synchronous messages is referred to as synchronous support period (SSP). The Master station, by means of a specific control message (CM in Figure 6.7), establishes the beginning and duration of the SSP for each change request.

The following relationship can be established between the $STRT$ with and without the AMS support:

$$STRT_{AMS} = STRT_{SMS} - SSP \tag{6.14}$$

If the change to the message set consists only in the addition of a new message, the process above described is adequate. However, if the change request is performed over a message stream already present in the SRT (e.g., to change the stream's period), the existing instances of the message in the SMS during the synchronous support period (SSP) must be suppressed. Those instances still use to the older parameters (before the change) while the updated instances are transmitted by the asynchronous system. The suppression is achieved by applying a filter to the TM which resets the bit that corresponds to that message. Therefore, removing one stream present in one plan already built only requires a change in one bit of that filter.

Depending on the type of the change request that is made, one or several of the following actions may be necessary:

1. A change of one bit in the filter;
2. The production of a control message to signal the start and duration of the SSP (synchronous support period);
3. A set of data messages produced in the AMS, during the SSP.

If the change request consists in the elimination of one message stream, only action 1 is required. However, if the change request consists in adding a new message, control and data messages will be produced in the AMS during the SSP (actions 2 and 3). If the change request concerns a modification in the parameters of an existing message (e.g. period), actions 1,2 and 3 are required.

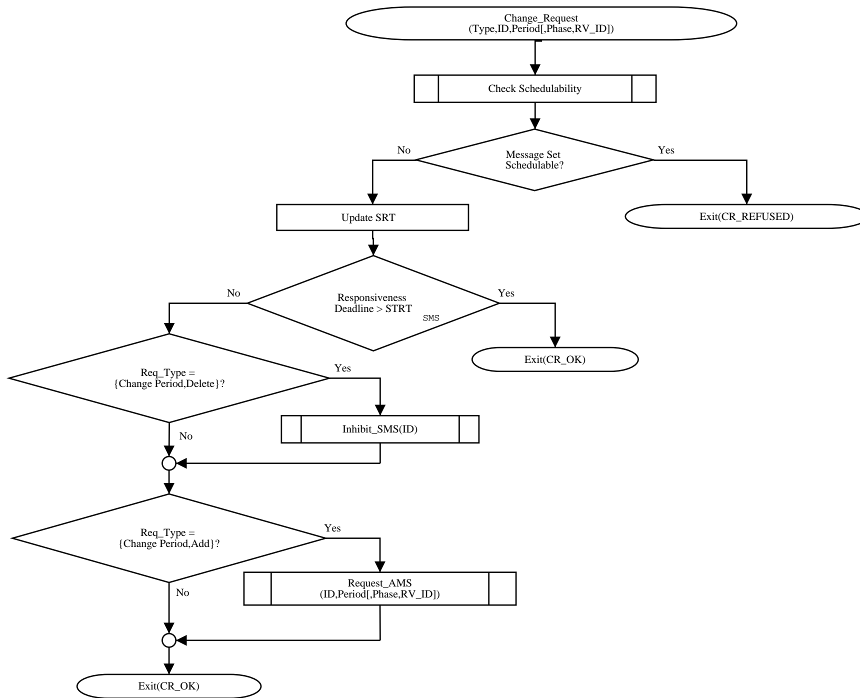


Figure 6.8: Operational flowchart

6.4.3 Implementation issues

From the operational point of view, several steps must be performed in order to process the request for a change to the synchronous message set. Figure 6.8 presents a flowchart describing the proposed methodology for improving the responsiveness of the planning scheduler for change requests.

After a request to a change on a synchronous message, a schedulability analysis is executed, which rejects changes that would result in a non-schedulable message set. However, in the remainder of this section we will consider that any requested change has already been analyzed and it does not compromise the message set schedulability. In case the on-line analysis is performed, its execution time must be included in the $STRT$.

If the change request is accepted, the change is made to the SRT, and then it is evaluated if their admissible delay to take effect on the bus allows the use of the SMS alone ($Response_deadline > STRT_{SMS}$). If so, no further handling is necessary. Otherwise, two more steps must be performed. In first place it is verified if the request is made over a message already present in the

SMS (change of period or elimination), and, if so, a request is made to the dispatcher to remove the message from the synchronous message area during the $STRT$. Next, it is evaluated if the request implies to add a message; if so, a request is made to the AMS to start its production in asynchronous mode.

The start and duration of the temporary production of synchronous messages using the AMS, if required, is commanded by the master node via a control message. During this period of time (SSP as defined before) the producers transmit the required messages autonomously. The communication overhead of this control protocol is thus one control message per change request. The start of production message (SP_SSP) must convey the ID of the message to be produced, its period (expressed in ECs), a release delay (also in ECs) that must be applied between the reception of this message and the effective start of stream production, and the number of instances that must be produced using the AMS. Seven data bytes are used, one for variable ID, and two for message period, release delay and number of instances.

6.4.4 Performance analysis

During the synchronous support period (SSP), the control and synchronous messages corresponding to a change request are handled by the AMS, and will compete for the bus jointly with other asynchronous messages. For time-critical message streams it is necessary to guarantee in advance that the AMS has enough capacity to timely support the transmission of the control and data messages respectively during the $STRT_{AMS}$ and SSP. For this reason, it was derived a set of sufficient conditions, which allow to guarantee that a set of change requests is handled within specific time bounds.

Bus demand and responsiveness

As explained in Section 6.4.2, during the SSP any new and modified messages are produced using the AMS. However, if the request is accepted by the schedulability test it means that the SMS has enough leeway to hold the message. As the AMS holds the remaining bandwidth, it can be concluded that the production of data messages during the SSP will use space borrowed by the AMS from the SMS. However, this argument requires that the start of synchronous support period (SSP) takes into account the phase of the

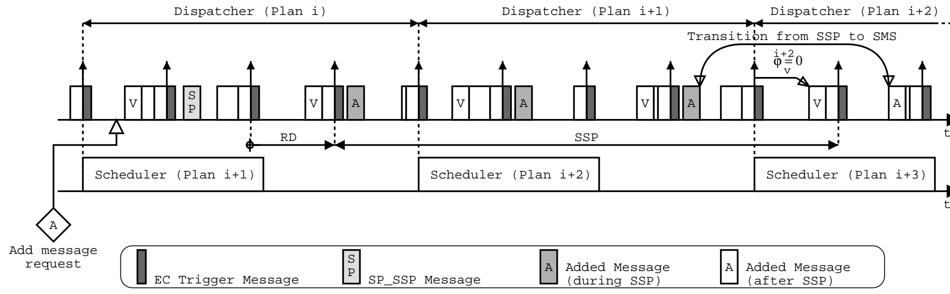


Figure 6.9: Transition from SSP to SMS

variable. This is necessary to maintain the same relative phasing in both production periods, SSP and SMS, resulting in a smooth transition from one to the other.

Consider for instance the example illustrated in Figure 6.9, where a message is added with period of 2 ECs and phase of 1EC relative to a reference message v. The SP_SSP message is sent by the Master Station, informing the respective producer node that it should start producing the new stream using the AMS with period of 2 ECs and starting in the 2nd EC after the reception of the control message. This way, the release of the first message in the stream is appropriately delayed (RD in Figure 6.9) so that the relative phasing is the same in SSP as in SMS.

In order to evaluate where the SSP should start, the Master node must calculate which will be the initial phase relative to the start of the plan of the first instance of the message produced in the SMS. Notice that this plan (i+2 in Figure 6.9) is not yet built at the request instant. However, knowing the initial phase of a variable v on plan i, its initial phase in plan (i+1) is given by Equation 6.15, where W is the length of the plan (in ECs) and P_v is the period of variable v (also in ECs).

$$\varphi_v^{i+1} = \left\lceil \frac{W - \varphi_v^i}{P_v} \right\rceil * P_v - (W - \varphi_v^i) \tag{6.15}$$

When the request for a change is performed, the current scheduler instance (i+1 in Figure 6.9) can be either terminated or still in execution. In the former case, the next plan (i+1) is already built and φ_v^{i+1} is known. Thus, Equation 6.15 is applied once, only, to determine φ_v^{i+2}. In the latter case, plan i+1 is not built yet and thus, Equation 6.15 must be applied twice

to evaluate ${\varphi_v}^i + 2$ based on ${\varphi_v}^i$. Knowing the relative phase of a message u with respect to a reference message v (Ph_u^v), and the initial phase of this one (${\varphi_v}^i + 1$) the number of ECs between the SP_SSP and the first instance of the message stream produced in the SMS (${\varphi_v}^i + 1$) is given by Equation 6.16, where W is the length of the plan, $curEC^i$ is the EC where the request is handled within plan i ($1 \leq curEC \leq W$) and Ph_u^v is the phase of the message being added (u) relatively to message v.

$$L_{RD+SSP_u} = W - curEC^i + W + \varphi_v^{i+2} + Ph_u^v \tag{6.16}$$

Finally, the number of instances that must be produced during the SSP (NI_{SSP_u}) is given by Equation 6.17.

$$NI_{SSP_u} = \left\lfloor \frac{L_{RD+SSP_u}}{P_u} \right\rfloor \tag{6.17}$$

The release delay of the first instance relative to the reception of the control message (RD) is given by Equation 6.18.

$$RD_{SSP_u} = L_{RD+SSP_u} - NI_{SSP_u} * P_u \tag{6.18}$$

When using the AMS support to increase the responsiveness to changes in the synchronous message set, the synchronous transient response time ($STRT_{AMS}$) is substantially reduced (Figure 6.7). In fact, its worst-case value occurs when the request is done before the beginning of the synchronous window of one EC and the respective control message (SP_SSP) can only be transmitted in the asynchronous window of the following EC. Unless the accumulated number of control messages, due to the queuing of several requests, is greater than the available space in the asynchronous window, the $STRT_{AMS}$ will be less than 2 ECs, plus the release delay RD. Since $0 \leq RD \leq P_u - 1$, the worst-case value of the responsiveness achieved by this method, expressed in ECs, is given by Equation 6.19, where P_u is the period of variable u, measured in ECs.

$$STRT_{AMS_u} < P_u + 1 \tag{6.19}$$

Pre-run-time analysis

The SP_SSP control messages are transmitted in the asynchronous windows, competing for the bus together with other asynchronous messages. Thus, to guarantee that the bound in Equation 6.19 is respected, it is necessary to perform a pre-run-time evaluation. As discussed above, during the SSP the production of the synchronous messages is made in space borrowed from the SMS by the AMS. However, the same assumption cannot be made concerning the control messages. For these, it must be evaluated if the minimum bandwidth reserved to the AMS at configuration time ($LAW = E - LTM - LSW$) is enough to handle them in a timely way. As discussed in Sections 4.4 and 6.3.1, due to a possible idle-time insertion (α), the minimum guaranteed effective bus time available in each EC for asynchronous transactions is less than LAW and it can be computed using Equation 6.20.

$$LAW_{UT} = LAW - \alpha \tag{6.20}$$

The inserted idle-time term (α) is bounded by the transmission time of the longest asynchronous message (Ca), which is given by Equation 6.21.

$$Ca = max\{C_i\},\ i = 1..N_A \tag{6.21}$$

In a worst-case situation, when using either higher transmission rates or low processing power micro-controllers, the Master may take more time to handle a change request (i.e. perform the previous calculations) than to send the respective SP_SSP message. In this situation, the Master must release the bus between any consecutive SP_SSP messages. Consequently, in the meanwhile, the bus can be taken by another asynchronous message which will cause a blocking to the following SP_SSP message. The maximum duration of such blocking is also given by Ca. This same blocking can happen every time the Master tries to send an SP_SSP message. Therefore, if there are N_{CR} change requests pending, in order to guarantee that the respective SP_SSP messages can be sent in one EC so that the bound in Equation 6.19 is respected, the following condition must be verified:

$$N_{CR} * (Len(SP_{SSP}) + Ca) \leq LAW_{UT} \tag{6.22}$$

This expression establishes a relationship between LAW and the maximum number of simultaneous change requests that the system is expected to handle so that the STRT of each request is still bounded by Equation 6.19.

6.5 Dependability issues

As stated is Section 3, distributed real-time systems carry real-time activities that, to be correctly accomplished, require both timely execution of tasks within processing units and timely data exchanges between network nodes. Failures on any of these aspects can lead to disruption of the services provided to the application. When dealing with safety-critical applications, in which system failures can lead to catastrophic results (concerning either equipment, materials or human lives), specific fault-tolerance techniques must be used to limit the impact of such failures or even avoid their occurrence, at least within specific fault models.

Since the FTT paradigm aims also at safety-critical applications, within our work group there is an active line of research in fault-tolerance and dependability issues. This section presents a contribution to such research, a master replica synchronization mechanism, which was jointly specified and developed in the scope of this thesis.

6.5.1 FTT-CAN Master replication

The whole FTT-CAN distributed system is synchronized by the reception of the EC trigger message. If the master stops working, the TM is omitted leading to a complete communication disruption. To overcome such situation backup masters can be used. During normal operation these masters monitor the network looking for EC trigger messages. Whenever a TM is delayed more than a given tolerance an election mechanism is triggered and one of the backup masters takes the control and starts transmitting the missing EC trigger messages, becoming from that instant on the primary master. In a FTT-CAN network there can be up to 8 masters, each one having a unique identifier (Table 6.2).

At node level, master nodes use internal replication of the scheduler and the SRT to achieve fail-silence in the value domain. Whenever the EC schedule built by the replica does not match the one built by the primary one, the generation of trigger messages is autonomously stopped. At the system

level, fault tolerance is implemented by the replication of the master node itself (spatial redundancy).

6.5.2 Master replica synchronization protocol

A fundamental aspect is the synchronization between primary and backup masters. It must be guaranteed that in each EC all the masters generate similar schedules at the same time. In every EC all backup masters compare their own schedules with the schedule conveyed in the trigger message and also compare a short cyclic sequence number (3-bit) that is also encoded in the trigger message. Whenever an inconsistency is detected the backup master issues a synchronization request, causing the current primary master to download the SRT as well as the relative phasing information necessary to resume scheduling synchronously. The synchronization process below described was developed for systems implementing a planning scheduler (Section 6.4). Ongoing work is being performed concerning systems scheduled on a per-EC basis.

The synchronization process (Figure 6.10) may take a few ECs, depending on the size of the SRT and on the current network utilization. It is a time critical task since during its execution the backup master cannot replace the active master in case of failure. Furthermore the overhead introduced by the synchronization protocol also affects the performance of the asynchronous messaging system, since it relies in asynchronous control messages to transmit the information required.

The quantity and nature of the data that has to be received by a backup master to enable its synchronization with the active master depends on the adopted scheduling algorithm. However, this data can usually be divided in two groups, one containing message properties that are independent of the scheduling activity and other containing scheduling dependent properties. Considering, as an example, either Rate Monotonic (RM) or Earliest Deadline First (EDF) scheduling policies, the scheduling independent properties consist in the data size, period and relative deadline. On the other hand, scheduling dependent data consists in the messages phases at the beginning of each plan or EC for RM and the absolute deadlines of pending message instances for EDF.

The message identifier is always sent with the pertinent data. The timeline of the synchronization process is depicted in Figure 6.10. Once the active

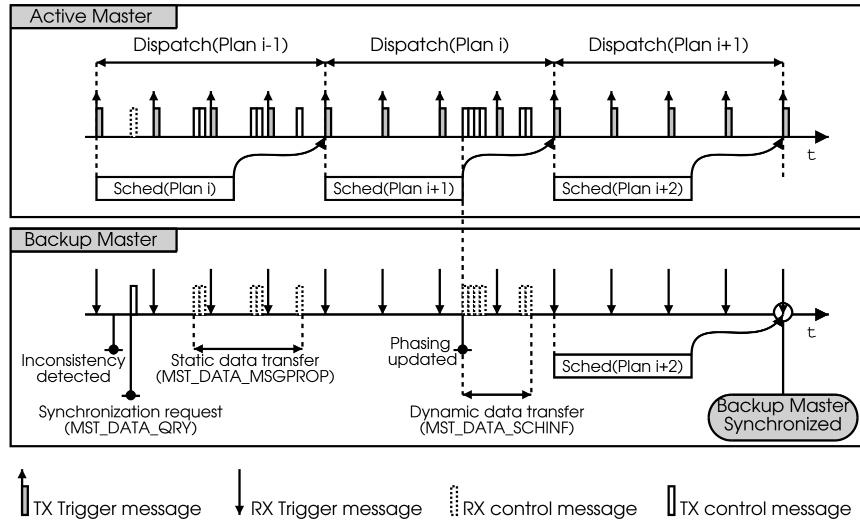


Figure 6.10: Timeline of the scheduling synchronization process

master receives the synchronization request (*MST_DATA_QRY*), it starts to download the SRT table and the relative phasing data in two rounds. In the first round, the SRT is split and conveyed into several messages (*MST_DATA_MSGPROP*). These messages carry only the scheduling independent data. Once the first state transfer round is complete, the scheduling dependent data is also split into several messages (*MST_DATA_SCHINF*). The transmission of this last state transfer round must be enclosed within a single plan and only after the scheduling of the next plan is completed in order to assure the consistence of the time dependent scheduling data. Once the scheduling dependent data is fully received by the backup master, this one waits for the beginning of the next plan to start the scheduler.

After completing the scheduling of the next plan, the backup master is ready to monitor the trigger messages produced by the active master and replace it in case of failure, as soon as a new plan begins. The start of a new plan is encoded in control part of the trigger message (Table 6.2).

6.5.3 Computing the worst-case synchronization time

The resynchronization of an FTT master requires the proper reception of a set of data from the currently active Master. During this process the backup master is unable to replace the current active master, since it does

not have enough information either in the time or value domain, to build schedules in parallel. Therefore, to calculate the system failure probability it is important to compute an upper bound for the time required by the synchronization process.

The transmission of the scheduling independent data can spawn along more than one plan, since these values do not change due to the scheduling activity. However, scheduling dependent data must be completely transmitted between the end of the activity of the scheduler and the end of the plan, since in each instance the scheduler updates it. If for some reason this could not be accomplished in a particular plan the whole set of scheduling dependent data must be then sent again after the next instance of the scheduler.

The number of CAN frames required to download the data from the active master depends both on the quantity of messages (N_{RT}) and on the amount of data required to represent the respective set of properties for each one. Knowing that the maximum number of data bytes that can be carried in a single CAN frame is 8, Equation 6.23 gives the number and size of the CAN data frames needed to transmit both static and scheduling dependent data of the whole set of synchronous messages. The MP_{LEN} parameter defines the number of bytes required to carry the properties of each message.

$$\begin{cases} \lfloor (N_{RT} * MP_{Len})/8 \rfloor_{DLC_1=8} + 1_{DLC_2=(N_{RT}*MP_{Len})-\lfloor (N_{RT}*MP_{Len})/8 \rfloor *8} \\ \qquad\qquad , if\, DLC_2 \neq 0 \\ \lfloor (N_{RT} * MP_{Len})/8 \rfloor_{DLC_1=8} \\ \qquad\qquad , if\, DLC_2 = 0 \end{cases} \tag{6.23}$$

Besides the data frames, the synchronization process also requires two more control frames:

- MST_DATA_QRY : sent at the beginning of the synchronization process, requesting data from the active master;
- MST_DATA_OK / $MST_DATA_SCHINF_REFRESH$: to signal the successful end of the transaction or the need to update the state-dependent data frames, respectively.

None of these messages carry any data bytes.

The FTT-CAN protocol supports real-time asynchronous messages, with guaranteed response time, as described in Section 6.3.1. Providing the en-

semble of asynchronous messages exchanged on the system, the minimum bandwidth reserved for the asynchronous window and the relative priority given to the asynchronous messages used to convey synchronization data, it is possible to obtain an upper bound for the transmission time required to send the complete set of messages, using Equation 4.21 (Section 4.4.1).

The worst-case situation occurs when a new plan starts just before the transmission of the last message containing scheduling dependent data. In this case the whole set of messages carrying this type of data must be transmitted again, starting after the end of the current scheduler instance (Figure 6.10). After receiving the updated data, the out-of-sync backup master needs to wait for the beginning of the next plan to start the scheduler with same data as the active master. After having built the schedule, the beginning of a new plan sets the instant from which the backup master becomes fully synchronized and able of acting as a master if necessary (Figure 6.10).

Therefore, an upper-bound to the time required (ST_{WC}) for a master to become fully synchronized can be computed by calculating the set of messages required by the process (M_{SP}) and applying Equation 6.24:

$$ST_{WC} = R_S + 2 * PLAN_W \tag{6.24}$$

where R_{SP} is the response-time of the last message in M_{SP} counting from the synchronization request instant and $PLAN_W$ is the plan duration is ms.

6.5.4 Active master replacement

The replacement of the active master by a backup master, in case of failure, is based on a timer and on the normal CAN transmission and receive interrupts. The takeover process is depicted in Figure 6.11. At the backup master, upon the reception of a trigger message, a timer is programmed to generate an interrupt during the reception of the next trigger message. During the interrupt service routine (ISR) associated with the reception of a trigger message the backup master writes on the transmission buffer its own trigger message, orders its transmission and immediately after issues a transmission abort command. If the active master is already transmitting a trigger message in the bus, then the abort operation is successful, otherwise the abort operation fails and the trigger message produced by the backup

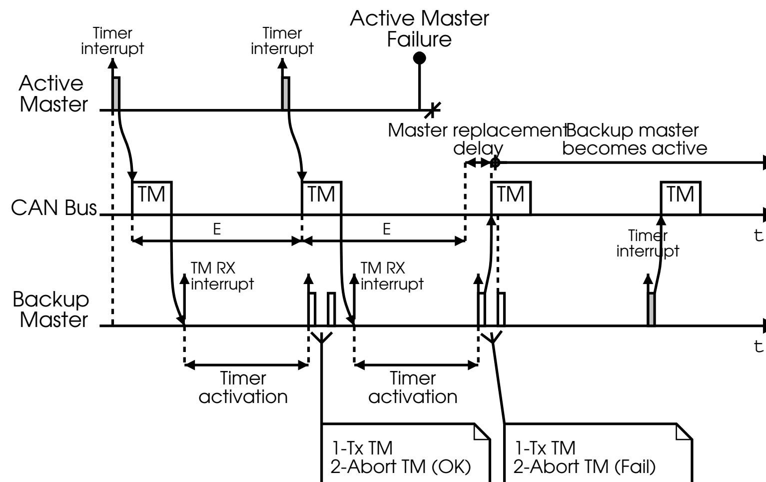


Figure 6.11: Master replacement process

master is effectively transmitted. In the latter situation the backups master becomes the system active master. This situation can be detected because a transmit interrupt will be raised in this latter case.

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

6.5.5 Experimental results

To assess the feasibility and correctness of the proposed synchronization process, some experiments were carried out using a 5-node network based on CANivete [FSMF98] boards. The EC duration was set to 8.9ms, the trigger message used 2 data bytes, supporting a maximum of 8 synchronous messages, and the maximum duration of the synchronous window was set to 4.5ms. The plan duration was 30 ECs. The network workload also included asynchronous data messages, with up to 8 data bytes. The synchronous message set used in this experimental set up is represented in Table 6.11. The synchronous messages were scheduled according to the Rate Monotonic policy. In this case the scheduling independent data consists of the message identifier, data size, period and absolute deadline, while the scheduling de-

ID	*Period*	*Deadline*	*Init phase*	*Size*
1	1	1	0	1
2	1	1	0	3
3	2	2	0	3
4	3	3	0	2
5	4	4	0	5
6	4	4	0	5

Table 6.11: Synchronous message properties.

(Period, Deadline and Init phase in ECs; Size in bytes)

pendent data consists only in the relative phasing of the messages at the beginning of the next plan. All these properties are encoded in one byte each.

Using Equation 6.23, the total number of messages needed by master synchronization protocol is three 8 byte messages for the scheduling independent data and one 8 byte plus one 4 byte messages to send the scheduling dependent data. The response time calculated from Equation 4.21 (Section 4.4.1) is $23.062ms$, resulting in an upper bound for the synchronization time (Equation 6.24) $ST_{wc} = 557.062ms$.

The experiment was repeated several times in different conditions and, on average, the time to fully synchronize was around $385ms$, which is less than one and a half plans. However, in a small fraction of the experiments this value was considerably higher ($550ms$), although below the computed worst-case value above referred. The low average synchronization time, when compared to the worst-case bound, can be explained by the use of a large plan, leading to a high probability of the synchronization requests being completely served before the end of the plan. Notice that due to low processing power of the micro-controllers used in the test platform, the use of such a large plan is a requirement.

6.6 Conclusion

This chapter presents the contributions to the FTT-CAN protocol developed during the scope of this thesis.

Concerning the synchronous traffic, it is explored the possibility of using distinct scheduling policies, namely RM and EDF. Simulation and experimental results show that the use of EDF instead of RM allows to increase the network utilization efficiency, with the increased scheduling overhead being reflected on the master node only. Moreover, it is performed a comparison with other techniques to perform EDF message scheduling on CAN. The results show that the FTT-CAN protocol achieves similar levels of schedulability, but without incurring in some important drawbacks of those approaches, like high overhead in all network nodes, constrained addressing scheme and difficulties in handling wide ranges of deadlines.

Previous implementations of the FTT-CAN protocol relied on a planning scheduler to reduce the scheduling overhead in the master node. However, such methodology also leads to a reduction in the responsiveness to changes to the synchronous message properties. In the scope of this thesis it was developed a method to overcome such effect, by using the asynchronous window to convey temporarily the synchronous messages during the period that the SMS is unable to reflect those changes in the bus traffic. The method proposed allows to have response times upper bounded by 2 ECs plus the message period. Moreover, this response time becomes completely independent of the plan length, which can thus be freely managed to suit the processing power of the platform.

Other relevant contribution to the FTT-CAN protocol consists in the development of the asynchronous messaging system. This chapter includes the adaptation of the generic analysis (Section 4.4) to the FTT-CAN implementation. Moreover, it is also presented a set of experimental results that show the validity of the implementation. These results show that the FTT-CAN protocol is able to carry real-time event-triggered traffic under guaranteed timeliness.

The final contribution to the FTT-CAN protocol is the development of synchronization and election protocols for fault-tolerant FTT-CAN systems. The synchronization protocol allows first the backup masters to acquire the current message set properties, and then to synchronize the internal activities (scheduler and dispatcher) with the active master. The election protocol defines the process of master replacement upon failure of the active master. Although this is on-going work, the first approach herein presented shows a possible way to deal with the existence of a single point of failure, which is

one of the main problems pointed out to centralized architectures, such as the FTT-CAN.

Chapter 7

The FTT-Ethernet protocol

Intelligent nodes, integrating microprocessors with communication capabilities, are extensively used in the lower layers of process and manufacturing industries, as well as in the control of complex machinery [Tho99]. In these environments, applications range from embedded command and control systems to computer vision, robotics and process supervision. The amount of information exchanged in these system has increased dramatically over the last years and it is now reaching the limits that are achievable using traditional fieldbuses, such as CAN, WorldFIP and ProfiBus [Son01, Dec01].

On other hand, modern process and manufacturing plants have layered network architectures allowing a separation between the different functional levels [BM01, JN01]. A typical taxonomy of such architectures consists in 3 levels, as depicted in Figure 7.1. Backbone level networks span the entire production facility and interconnect a broad range of computer systems, supporting office, engineering, production and management applications. Cell level networks typically interconnect a small number of control devices within a limited area (e.g. robots, conveyors, machine tools), which usually are responsible by some specific process or manufacturing tasks within the plant. Finally, the Fieldbus layer interconnects the set of sensors, actuators and controllers employed to perform specific tasks within specific machines or processes.

Concerning the traffic characteristics, at the backbone level usually there are large amounts of traffic exchanged, with no real-time constraints. This traffic results frequently from the access to remote resources, like databases, and thus is bursty, with data packets carrying several hundreds of bytes.

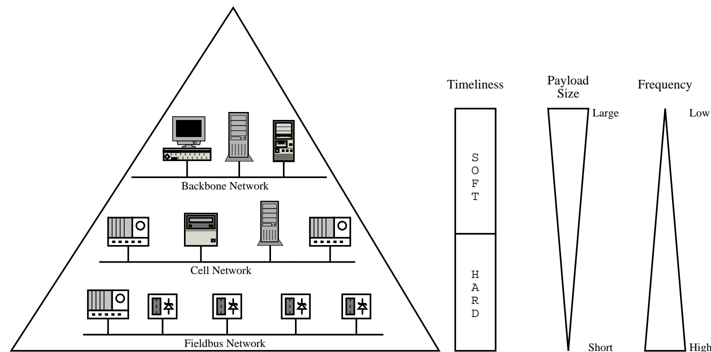


Figure 7.1: Layer model of factory communications

Response times in the range of seconds are usually acceptable [JN01]. At the Cell layer there are both real-time and non real-time data exchanges, and typically the data packets carry less data but occur more often when compared to the Backbone level. Finally, at the Fieldbus level it is typically found real-time traffic, usually generated by sensors and control devices, consisting of short data packets associated either with particular environment variables or actuation signals . These messages usually carry a few bytes at most, and occur regularly and frequently, demanding response times that can as low as a few milliseconds.

To fulfill both timeliness and throughput requirements, several protocols have been extensively analyzed for both hard and soft real-time communication systems, but Ethernet is emerging as one of the technologies of choice. Besides being a cheap, mature and well specified technology, with wide availability of both hardware equipment and technicians familiar with the protocol, two major factors are behind this interest in Ethernet: bandwidth and compatibility. In fact, steady increases on the transmission speed have happened in the past and are expected to continue occurring in the near future, and thus it can be expected that Ethernet should be able to support current and future demands in this type of applications. With respect to the compatibility issue, TCP/IP stacks over Ethernet are widely available, allowing the use of application layer protocols such as FTP, HTTP, SOAP, etc. The support of such protocols leads to an inherent compatibility with the communication protocols used at higher plant levels, easing the information exchange between plant levels, which in this case can be accomplished

without the need for communication gateways [JN01]. This framework facilitates ubiquitous access to devices in the plant, allowing for instance equipment controllers to communicate directly with each other, with information system servers and with field devices.

As discussed in Section 3.3, the destructive and non-deterministic arbitration mechanism employed by the Ethernet protocol prevents its direct use to convey real-time traffic. This situation led to the development of several protocols meant to bring such capabilities to Ethernet, the most representatives of which have been briefly described in Section 3.3. However, none of these proposals completely fulfills the requirements described in Section 4.1, which are summarized bellow.

- Time-triggered communication with operational flexibility;
- Support for on-the-fly changes both on the message set and the scheduling policy used;
- On-line admission control to guarantee timeliness to the real-time traffic;
- Indication of temporal accuracy of real-time messages;
- Support of different types of traffic: event-triggered, time-triggered, hard real-time, soft real-time and non-real-time;
- Temporal isolation: the distinct types of traffic must not disturb each other;
- Efficient use of network bandwidth;
- Efficient support of multicast messages;

This observation fostered the interest in applying the FTT paradigm to Ethernet, leading to the FTT-Ethernet protocol, which will be presented in the reminder of this chapter.

7.1 The FTT-Ethernet Elementary Cycle

The FTT-Ethernet elementary cycle structure follows closely the FTT paradigm EC structure described in Section 4.2.2 and it is depicted in Figure 7.2. The

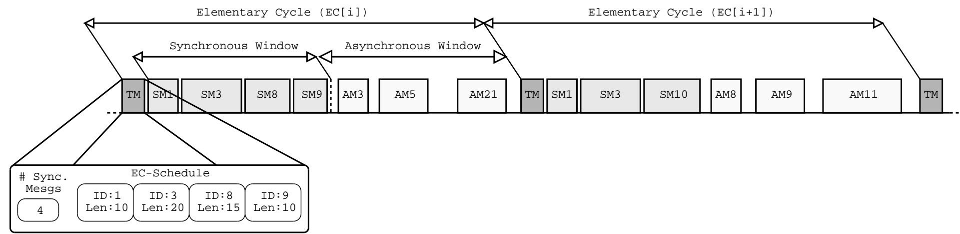


Figure 7.2: FTT-Ethernet Elementary Cycle

EC starts with the trigger message, which in this case conveys the quantity, identification and length of the synchronous messages that should be produced in the respective synchronous window. With this information nodes can compute the transmission instants of each of the synchronous messages as well as the length of the synchronous window.

7.1.1 Message Arbitration

As discussed in Section 3.3.1, the CSMA/CD arbitration technique employed by Ethernet turns it inadequate to carry real-time traffic, since the message transmission times are non-deterministic. To overcome this situation, the FTT-Ethernet protocol adds a transmission control layer above the Ethernet MAC, to achieve predictable transmission times.

Concerning the synchronous traffic, the TM conveys not only the identification of the messages but also their transmission time (Figure 7.2). Moreover, the messages must be transmitted in the same order indicated in the TM. This way, nodes having synchronous messages scheduled for transmission can compute the time required by other synchronous messages that must be transmitted before and start the transmission at that instant. If all the nodes follow this strategy the transmission instants become disjoint in the time domain and thus no collisions occur, resulting in predictable transmission times.

With respect to the asynchronous traffic, a different arbitration scheme must be used. Contrarily to the synchronous traffic, in this case there is no global knowledge about which nodes in the system have messages to transmit. The only way that nodes have to gather information about the system status is by monitoring the communication medium state. To achieve collision-free transmissions, the FTT-Ethernet protocol adopts a distributed

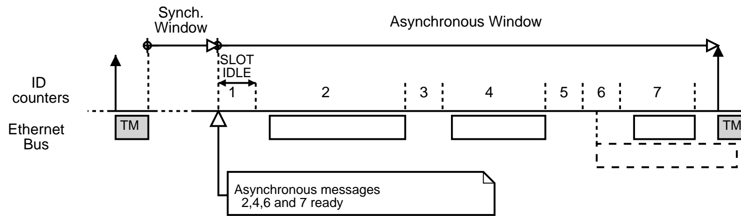


Figure 7.3: Asynchronous message arbitration scheme

arbitration scheme based on mini-slotting, which uses the communication medium status to assign the right to transmit to the highest priority ready message. Asynchronous messages have a unique identifier (Section 4.2.3), and to each identifier it is associated an also unique priority and a corresponding transmission slot.

The asynchronous window is divided in time slots, each one assigned to a specific message ID (Figure 7.3). After the start of the asynchronous window, all the nodes in the network that are senders of asynchronous messages set an internal ID counter to a predefined value (e.g. 1), which corresponds to the highest possible priority. If the asynchronous message with priority 1 is ready, its sender node starts its transmission. If not, the bus will remain idle. After a pre-defined amount of time ($SLOT_IDLE$), all the nodes check the bus state. If there is an ongoing transmission, the nodes wait for the end of the transmission and then increment the internal ID counter. If there is no ongoing transmission, the nodes infer that the message was not ready for transmission and increment the internal ID counter immediately. This process is repeated until the end of the asynchronous window and provides a collision free arbitration mechanism for event messages.

7.1.2 Enforcing temporal isolation

To maintain the temporal properties of the traffic, both synchronous and asynchronous messages should be confined within their respective windows, enforcing a strict temporal isolation between both phases. As in the case of FTT-CAN, this is achieved by preventing the start of message transmissions that could not complete within their respective window.

Concerning the synchronous traffic, messages scheduled to be transmitted should fit within their respective window, unless some abnormal event or

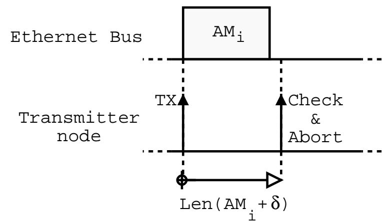


Figure 7.4: Preventing window overrun

perturbation, such as an error, has prevented them to be transmitted at the scheduled instants. To avoid that in this situation the messages could extend beyond the synchronous windows, each node that transmits a message is also responsible for verifying if the message has been completely transmitted within the specified time interval (Figure 7.4). To perform this operation, whenever a node is allowed to transmit a message it also sets a timer that expires at the expected end of transmission instant plus a small tolerance factor (δ in Figure 7.4). When this timer expires the status of the Ethernet controller is verified and, if due to some abnormal condition the message had not yet be transmitted, its transmission is aborted.

Concerning the asynchronous traffic, nodes having ready asynchronous messages have no knowledge about the state of the remaining nodes. Therefore there are no guarantees that the set of ready messages among all system nodes will fit within the asynchronous window. Thus, when a node having asynchronous messages to transmit wins the arbitration process (as described in Section 7.1.1) it must verify if the time remaining until the end of the asynchronous window in enough to transmit the message. If so, it transmits the message (Messages 2,4 and 7 in Figure 7.3). If not, the transmission is not started and the message is kept in the transmission queue until the following EC (Message 6 in Figure 7.3). As for the case of the synchronous traffic, sender nodes must verify if at the expected end of transmission instant the message was in fact completely transmitted, and issue an abort transmission command if due to some perturbation the transmission was delayed.

7.1.3 FTT-Ethernet message types

The FTT-Ethernet protocol defines the following message types:

- EC Trigger Message [TM_MESG_ID];

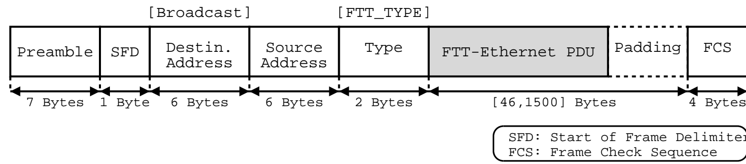


Figure 7.5: FTT-Ethernet frame

Type		TM Flags		Num. Synch. Mesgs	ID	Tx Time	...
TM Type	Master ID	Reserv.	Seq. Num.				
2 Bytes		2 Bytes		2 Bytes	2 Bytes	1 Byte	...
[b15..b12]	[b11..b0]	Undef.	[b7..b0]	[b15..b0]	[b15..b0]	[b7..b0]	...
TM_MESG_ID	0 to 4096	Undef.	0 to 256	0 to 65535	0 to 65535	0 to 256	...

Table 7.1: EC Trigger Message structure

- Synchronous Data Messages [SM_DATA_MESG_ID];
- Asynchronous Data Messages [AM_DATA_MESG_ID];
- Control Messages [CONTROL_MESG_ID];
- Foreign protocol messages;

The structure of native FTT-Ethernet messages (Trigger Message, Synchronous and Asynchronous data messages and Control Messages) is depicted in Figure 7.5. These messages use the Ethernet broadcast address (destination address of the Ethernet frame set to all 1's), required by the producer-consumer co-operation model, and use the Ethernet frame Type field set to a constant value (FTT_TYPE), in order to allow the identification of the protocol specific frames. Foreign protocol messages are not modified by the FTT-Ethernet protocol and thus its contents and address scheme is not changed.

Trigger message

The contents of the TM is depicted in Table 7.1.

The **Type** field contains two sub-fields, the **TM Type** which conveys a constant value (MST_MESG_ID) identifying the frame as a TM, and the **Master ID** sub-field that contains a unique identifier for each one of

Type		SDM Flags		Time to Deadline	Message Data
SDM Type	SDM ID	Reserved	Seq. Num.		
2 Bytes		2 Bytes		2 Bytes	up to 1494 Bytes
[b15..b12]	[b11..b0]	Undef.	[b7..b0]	[b15..b0]	
DATA_MESG_ID	0 to 4096	Undef.	0 to 256	0 to 65535	

Table 7.2: Synchronous Data Message structure

the masters in the network. This field is expected to be used in the implementation of a master redundancy protocol, similar to the one presented in Section 6.5 for the FTT-CAN protocol. The **TM Flags** field also contains two sub-fields: a **Reserved** sub-field that is not used in the current version, and a **Sequence Number** sub-field that is incremented by the active master in each EC, facilitating the detection of missing trigger messages. The **Number of Synchronous Messages** field indicates how many synchronous messages are scheduled for the current EC. Finally, it follows a set of **(ID + Tx Time)** that identify each of the synchronous messages that should be produced in the EC as well as their respective transmission time, in μs.

Synchronous data messages

Synchronous Data Messages are used to periodically distribute state data among the network nodes, and are always transmitted within the synchronous window, when indicated in the EC-Schedule conveyed in the TM. The synchronous data message structure is depicted in Table 7.2.

The **Type** and **SDM Flags** fields are equivalent to their counterparts in the TM above described. The $SM_DATA_MESG_ID$ constant it is used in the **SDM Type** sub-field, tagging the message synchronous. The **Time to Deadline** is used to convey information about the "age" of the data, as described in Section 4.2.4. Finally, if follows the Message Data field, which conveys the data itself. Since Ethernet's data field is constrained to a maximum of 1500 Bytes and the overhead due to the FTT-Ethernet protocol (Type, SDM Flags and Time to Deadline fields) is 6 bytes, each FTT-Ethernet synchronous data message can carry up to 1494 data bytes.

Type		SDM Flags		Time to Deadline	Message Data
ADM Type	ADM ID	Reserv.	Seq. Num.		
2 Bytes		2 Bytes		2 Bytes	up to 1494 Bytes
[b15..b12]	[b11..b0]	Undef.	[b7..b0]	[b15..b0]	
AM_DATA_MESG_ID	0 to 4096	Undef.	0 to 256	0 to 65535	

Table 7.3: Asynchronous Data Message structure

Asynchronous data messages

Asynchronous Data Messages are used to convey event information, and are sent after explicit application request. Asynchronous data messages are always transmitted within the asynchronous window. The structure of a these frames is depicted in Table 7.3.

The structure of this frame is similar to the synchronous data message frame, except that in this case the $AM_DATA_MESG_ID$ constant it is used in the **ADM Type** sub-field, tagging the message as asynchronous.

As in the case of FTT-CAN, there are two levels of priorities associated with asynchronous data messages which map into two different traffic classes. Higher priority (**RT**) asynchronous messages are subject to real-time guarantees, and thus appropriate analysis (Section 4.4) can be performed in order to know in advance if its timeliness requirements can be met. However, such analysis does not involve the low priority (**NRT**) asynchronous messages, which are thus handled according to a best-effort policy. Low priority asynchronous messages fall into the non-real-time asynchronous traffic class. Asynchronous RT messages are assigned to higher priorities than NRT ones, thus are always transmitted first during the asynchronous window (Section 7.1.1). By this reason it is safe to ignore the presence of the NRT asynchronous messages in the schedulability analysis.

Asynchronous control messages

Asynchronous Control messages are used to perform system management (e.g master synchronization data, software download, requests for SRT changes, etc.). The internal structure of this type of frame is similar to the structure of both synchronous and asynchronous data messages, as can be observed in Table 7.4, with the only difference in the **Type** field, where it is indicated in this case that the message is an asynchronous control message (**CM Type**

Type		SDM Flags		Time to Deadline	Message Data
CM Type	CM ID	Reserv.	Seq. Num.		
2 Bytes		2 Bytes		2 Bytes	up to 1494 Bytes
[b15..b12]	[b11..b0]	Undef.	[b7..b0]	[b15..b0]	
CONTROL_MESG_ID	0 to 4096	Undef.	0 to 256	0 to 65535	

Table 7.4: Control Message structure

field set to $CONTROL_MESG_ID$).

As for asynchronous data messages, there are also two priority levels assigned to control messages. The high-priority messages (**HP**) have the highest priority among all the asynchronous messages and are used for time-critical management operations, such as alarms. The lower priority (**LP**) control messages have the lower priority among all the asynchronous messages and are used to carry operations that are not time constrained, such as remote diagnosis and data logging.

7.2 Schedulability analysis

7.2.1 Message's transmission time computation

Schedulability analysis requires the precise knowledge of the time necessary to perform the transmission of each message carried in the system, which is computed as follows.

Trigger Message

The FTT-Ethernet TM length can vary from EC to EC, depending on the number of synchronous messages scheduled for transmission on each EC. However the use of varying values for the length of the TM in simpler schedulability tests is not desired since it would require a significant computation overhead (in fact it would be necessary to build the schedules to know how many messages would be scheduled for each EC). Thus it is defined a maximum value for the number of messages that can be scheduled in each EC (EC_MAX_SMESG) that is used to compute a worst-case (maximum) transmission time for the TM (LTM). The TM requires an overhead of 6 Bytes (Type, TM Flags and Number of Synchronous Messages fields) plus 3 bytes (ID + TX Time fields) for each synchronous message scheduled for the

LTM (*Max mesgs by EC / Bytes*)	LTM μs		EC usage (%)			
		EC(ms)	5	10	50	100
10/72	57.6		1.15	0.58	0.12	0.06
20/92	73.6		1.47	0.74	0.15	0.07
50/182	145.6		2.91	1.46	0.29	0.15
100/332	265.6		5.31	2.66	0.53	0.27

Table 7.5: Communication overhead imposed by the EC Trigger Message

respective EC. Therefore, considering the length restrictions (Section 3.3.1), the worst-case length (in bytes) for the TM is given by Equation 7.1.

$$LTM_{byte} = \begin{cases} 72 & , EC_MAX_SMESG < 14 \\ 32 + 3 * EC_MAX_SMESG & , EC_MAX_SMESG \geq 14 \end{cases} \quad (7.1)$$

Ethernet devices must allow a minimum idle period between transmission of frames [IEEc], commonly known as inter-frame gap (IFG) or inter-packet gap (IPG). This time period is meant to provide a minimum recovery time between frames to allow devices to prepare for reception of the following frames. The minimum inter-frame gap is 96 bit times, which corresponds to $9.6\mu s$ for 10 Mbps Ethernet and $960ns$ for 100 Mbps Ethernet. Knowing the transmission speed (TX_{RATE}), the worst-case time required to transmit the trigger message can now be computed (Equation 7.2).

$$LTM = \frac{LTM_{byte} * 8 + 96}{TX_{RATE}} \quad (7.2)$$

As stated in Section 4.2.1, the use of the master/multi-slave transmission control, in which one single TM triggers the transmission of several data messages in distinct nodes, allows to considerably reduce the protocol overhead when compared with a pure master-slave transmission control. Table 7.5 presents the worst-case overhead due to the transmission of the TM in FTT-Ethernet in four exemplificative scenarios, referred to 10Mbps Ethernet ([IEEf]). Recall that this overhead depends on the EC length and the maximum number of synchronous messages allowed in each EC.

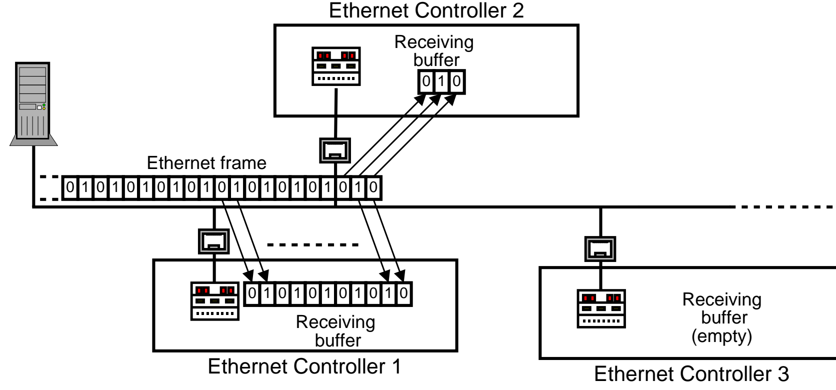


Figure 7.6: Ethernet propagation delay

Control and data messages

Noting that the FTT-Ethernet protocol overhead required by both synchronous, asynchronous and control messages is equal, its respective byte length and transmission times can be computed using Equations 7.3 and 7.4 respectively, where DLC represents the data payload of the message.

$$M_{Len} = \begin{cases} 72 & , DLC \leq 40 \\ 26 + 6 + DLC & , DLC > 40 \end{cases} \tag{7.3}$$

$$M_{TX_time} = \frac{M_{Len} * 8 + 96}{TX_{RATE}} \tag{7.4}$$

7.2.2 Synchronous traffic

The schedulability analysis presented in Section 4.3 can be directly applied to the FTT-Ethernet protocol with just a small adaptation.

Due to the relation between the transmission speed and the bus length, in Ethernet distinct receiver nodes can be receiving different bits in the same time instant, as depicted in Figure 7.6.

This transmission methodology results in some unpleasant effects. On one hand, unless the "copper distance" of the distinct network nodes is known in advance, there is no easy way to make the distinct nodes to agree in a common time value for the reception instant of the trigger message. On the other hand, it must be ensured that messages have enough time to propagate through all the network before other message can start to be transmitted.

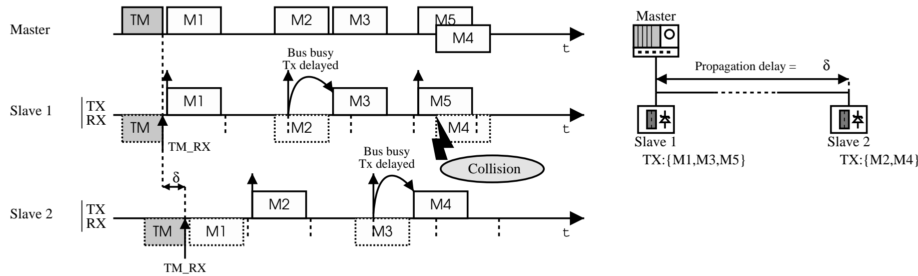


Figure 7.7: Unwanted collision between synchronous messages

An exact computation of this value would require a precise knowledge about the network length and the position of each node within the network. If both these effects are not properly considered, frame collisions can occur, compromising the fulfillment of the traffic timeliness requirements. Figure 7.7 depicts a scenario with a master node and two slaves, one near the master and the other in the farther end of the network. If the propagation delay (δ) is ignored in the scheduling, a collision between synchronous messages M4 and M5 happens.

Computing accurately the message propagation delays would require a complete characterization of the network, namely the propagation speed in the physical medium, delays due to the presence of hubs and the relative position of the nodes. Gathering all this information not only is complex but also would imply that any change on the network topology, such as adding or removing nodes or even connect a node to a different hub port, would impact on the scheduling parameters. Moreover, the inclusion of this information would strongly increase the scheduling complexity. Therefore, for the FTT-Ethernet implementation it was decided to use a single worst-case value, ETH_DELAY_UB, which depends only on the worst-case propagation delay that can occur between any two points of the network. This value is then added to the transmission time of all messages. Although this approach is less efficient, concerning network utilization, than the exact computation of the values for each message, it does not imply any increase in the scheduling overhead. Moreover in many applications the fieldbus networks span over limited geographical regions and thus the propagation delays are considerably shorter that the 464 bit times values allowed by the Ethernet protocol ([BMK88]). The ETH_DELAY_UB value can be easily com-

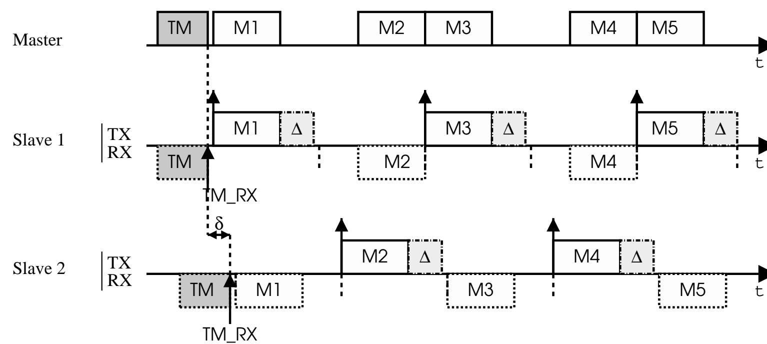


Figure 7.8: Including the propagation delays in the schedule

puted by knowing the maximum cable length of the Ethernet segment and, when present, by adding the delays due to hubs, which is a parameter that is usually available from theirs respective data-sheets. Figure 7.8 illustrates the same set-up depicted in Figure 7.7, but with the message transmission times inflated as described above.

7.2.3 Asynchronous traffic

The asynchronous traffic schedulability analysis presented in Section 4.4 was based on the following assumptions:

1. When two or more asynchronous messages contend for bus access, they are transmitted strictly according to their relative priorities;

2. The transmission time of all message instances of the same message stream are the same;

3. The arbitration process does not consume bandwidth.

With the mini-slotting arbitration mechanism used by the FTT-Ethernet protocol (Section 7.1.1) assumption 1 is met. Moreover, in Ethernet the packet size does not depend on the particular data value, thus assumption 2 is also met. However, the mini-slotting scheme uses waiting times to assess the bus state and thus assumption 3 is violated.

According to the mini-slotting scheme described in Section 7.1.1, there is a disjoint time interval assigned to each asynchronous message. When a node has a message to transmit it must wait for the right slot and then

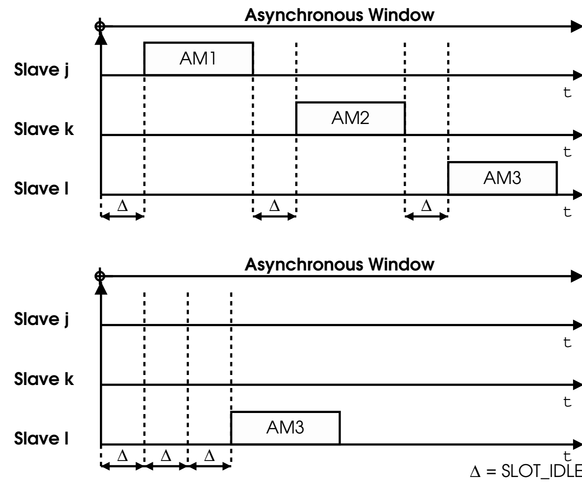


Figure 7.9: Asynchronous arbitration overhead

start the transmission. The transmission must start within a specific time interval since the other network nodes will assess the bus state after that same time interval to infer if the message was ready or not. Although the nodes should start the transmission right after the beginning of the respective slot, due to the processing overhead required to trigger a message transmission and also due to the propagation delay in the physical medium, the start of the message can be received at any time during the pre-defined time slot duration. Due to this uncertainty a conservative approach should be used, that is, each arbitration step is considered as requiring the maximum possible time ($SLOT_IDLE$). If this conservative approach is used the arbitration process can be easily modeled, since the total arbitration time felt by a particular message becomes independent of the higher priority messages being ready or not. This is illustrated by Figure 7.9, where asynchronous message $AM3$ observes 3 time slots used by the arbitration process, despite higher priority messages $AM1$ and $AM2$ being ready for transmission (on top) or not (on bottom).

Therefore Equation 4.24 requires only a small modification to account for the overhead due to the mini-slotting arbitration scheme. Noting that the arbitration process is started in the beginning of each asynchronous window, in each new EC the mini-slot ID counter is preset to 1 and the arbitration process is restarted. Thus, an asynchronous message i suffers two types of interferences from higher priority messages:

- An arbitration interference, occurring once by each higher priority message (ready or not), in every EC;
- The transmission time of the ready instances;

The arbitration overhead is independent of the properties of the higher priority messages. It is only important to know how many higher priority levels exist (N_{hp_i}) and the length of the arbitration slot. Equation 7.5 models both these factors.

$$H_i(t) = \sum_{j \in hp_i} \left\lceil \frac{t + \sigma^{ub}}{mit_j} \right\rceil * C_j + \left\lceil \frac{t}{E} \right\rceil * SLOT_IDLE * N_{hp_i} \quad (7.5)$$

7.3 FTT-Ethernet implementation

The implementation of the FTT-Ethernet protocol requires an adequate management of its components and of the interactions among these and the application software, in order to obtain a correct behavior of the communication system. The most sensitive protocol components, such as the Dispatcher and the Scheduler in the master and the FTT-Ethernet Interface Layer in the slaves, present tight temporal constraints that must be met. To fulfill these temporal constraints and support a higher abstraction level in the applications development, the FTT-Ethernet implementation was performed over a real-time kernel. The real-time kernel should support multitasking, real-time scheduling, expression of diverse task constraints (e.g. temporal, precedence and resource), inter-task communication and synchronization, and device drivers to isolate hardware dependent code. The real-time kernel used was S.Ha.R.K. (**S**oft and **Ha**rd **R**eal-time **K**ernel) [GGAB01], developed in the ReTiS Lab of *Scuola Superiore di Studi e Perfezionamento S. Anna*, in Pisa, Italy.

7.3.1 S.Ha.R.K. brief overview

S.Ha.R.K. is a dynamic configurable kernel designed for supporting hard, soft, and non real-time applications with interchangeable scheduling algorithms. The kernel is fully modular in terms of scheduling policies, aperiodic servers, and concurrency control protocols. Modularity is achieved by

partitioning the system activities between a generic kernel and a set of modules, which can be registered at initialization time to configure the kernel according to specific application requirements. The kernel supports device scheduling, thus allowing to extend scheduling algorithms used for the CPU to other hardware resources. Tasks are owned by Scheduling Modules; each scheduling module behaves like a multi-level scheduler, in the sense that tasks registered on high priority modules are scheduled in foreground with respect to tasks registered on lower priority modules. The system is compliant with almost all the POSIX 1003.13 PSE52 specifications to simplify porting of application code developed for other POSIX compliant kernels. In addition to the standard features of the previously referred specifications, S.Ha.R.K. provides various other services, such as:

- Temporal isolation and task execution time control;
- Cyclic Asynchronous Buffers and other mechanisms for non-blocking communications;
- Interrupt and hardware port handling.

7.3.2 Implementing FTT-Ethernet on top of Shark

As referred above, the FTT-Ethernet protocol includes components that are time-critical as well as other components with more relaxed time-constraints. Moreover, it is important to reduce to a minimum the potential interference of the application software in the timeliness of the protocol components. These different timeliness requirements are easily managed by S.Ha.R.K., through its explicit support to tasks with distinct QoS requirements. In particular, the implementation of the Master node and of the Slave nodes inserts the set of important tasks in a higher priority scheduling module than the other non-critical tasks.

Master node

The time critical tasks performed inside the master node are the Scheduler and Dispatcher tasks. The Master node also may carry other non-critical activities such as the keyboard and display handling. The order of execution of the time-critical tasks related to communication activities is shown in Figure 7.10.

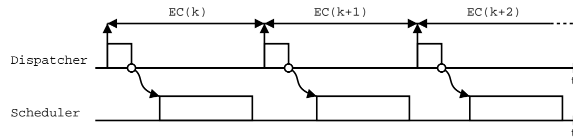


Figure 7.10: Master node: time-critical activities

The Dispatcher task is responsible for transmitting the EC trigger message, which carries the EC-Schedule for an elementary cycle. Since the correct behavior of the communication system is linked to the regularity of the EC duration, this task receives the highest priority and it is autonomously and periodically activated using the appropriate kernel services for hard tasks. The transmission of the EC trigger message is achieved by a call to the S.Ha.R.K. network API that directly sends a packet to the Ethernet layer.

The Scheduler also has strict time constraints because it must deliver a new EC schedule before the start of the next EC. For that reason its execution is enabled as soon as the Dispatcher reads the current EC schedule from the EC Schedule Register. It is thus precedence constrained with respect to the Dispatcher, and therefore it is registered as a hard aperiodic task. Unlike the Scheduler, which has only a deadline constraint, the Dispatcher is highly sensitive to jitter. Therefore, it is assigned to a scheduling module on a higher priority level than the Scheduler task.

Slave nodes

The internal critical tasks executed inside the slave nodes are related to the correct transmission and reception of the Ethernet messages. Other non-time-critical activities are carried out by the system, such as the local requirements database (NRDB) management, the update of the local buffers, the interface to higher protocol layers, and finally user tasks with keyboard and operator console handling. The message transmission and reception group includes two tasks, executed in the order depicted in Figure 7.11.

Notice that slave nodes must wait for an TM before initiating any communication activity. Then, every time an Ethernet packet arrives, an interrupt is raised. To limit the interference of that interrupt on the currently running task, the network interrupt handler queues the packet and activates a task

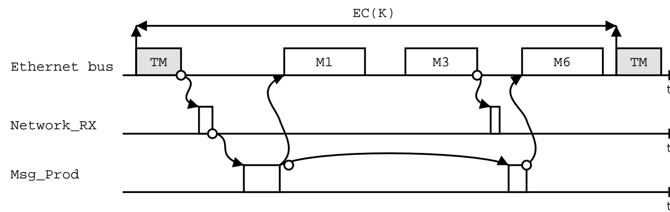


Figure 7.11: Slave node: time-critical activities

(Network_RX in Figure 7.11). This task is scheduled with all the other tasks, and it is responsible for parsing the packet header and separating the EC trigger messages from real-time and non-real-time ones. Since the activations of the Network_RX task follow an unknown pattern, the respective task model is soft. The nodes become aware of the reception of messages only after the execution of the Network_RX task. Therefore, this task must be inserted into the highest priority scheduling module.

The reception of an EC trigger message activates a task, Msg_Prod. This task identifies which local synchronous messages must be transmitted in the current EC and sets a number of timed-events, managed by the kernel, which will cause the transmission of the messages to occur at appropriate instants in time. Unbounded delays in the execution of this task lead to delays in the predetermined transmission instants and, consequently, to collisions on the bus. Therefore, this is the most time-critical and jitter-sensitive task on the slave node and for that reason it is also inserted into the highest priority scheduling module.

7.4 Experimental results

The FTT-Ethernet protocol inherits the properties of the FTT paradigm, namely on-line changes to the message set, distinct classes of messages (synchronous and asynchronous) with different timeliness requirements (hard, soft and non-real-time) and arbitrary scheduling policies. Some experiments concerning the implementation of RM and EDF scheduling policies have been performed [PAG02], yielding results similar to the ones obtained for its FTT-CAN counterparts (Section 6.2.2). However, due to its high bandwidth capacity, FTT-Ethernet is particularly well suited to support demanding real-time applications comprising activities such as multimedia and

computer-vision. Many of these applications have highly variable resource requirements, and thus high efficiency gains can result from the implementation of adequate QoS policies, which has motivated a special emphasis on the study and implementation of QoS management in the FTT-Ethernet protocol.

The issue of QoS management as been introduced in Section 5 concerning the FTT paradigm. This section presents the implementation of the Elastic Task Model [BLA98] over FTT-Ethernet.

7.4.1 Experiment characterization

The Elastic Task Model has been implemented on the top of the S.Ha.R.K. kernel [GGAB01] with the FTT-Ethernet as the real-time communication protocol. A set of experiments on a multimedia application were performed. The same set of experiments was carried out also with Hub and Switch based Ethernet to assess the benefits of the presence of a deterministic communication layer.

The developed application consisted in the simulation of a video surveillance security system, containing a set of physically distant video cameras and a central console. Each camera can be served by distinct QoS, according to the current bandwidth availability and the relevance of the data being sent. Change requests submitted to the Synchronous Messaging System are firstly submitted to the elastic guarantee mechanism. If the requests result in an unfeasible message sets, they are rejected. Conversely, if the resulting message set is schedulable, the QoS manager calculates the new periods and updates the Synchronous Requirements Table accordingly. Since the SRT is used both by the QoS manager and the Scheduler, it was used a mutex to enforce atomic updates.

The experimental set-up consists on 6 PC's, one acting as FTT Master, four as slaves, each producing a message stream associated to one camera, and finally one PC dedicated to collecting network traffic data. The communication infrastructure was Ethernet at 10Mbps.

The simulated cameras have a resolution of 384*288 pixels and a color depth of 8 bits, yielding a frame size of 884.7 Kbit. The camera data frames are sent without any kind of compression. Since the image frame size is larger that the maximum Ethernet packet size, each image frame is split in 1000 Byte packets. A header containing the camera ID, frame and packet number,

$Cam.$	$C_i(FTT/ET)$	T_{i_0}	$T_{i_{min}}$	$T_{i_{max}}$	E_i
1	0.89/0.84	10	5	30	1
2	0.89/0.84	10	5	30	2
3	0.89/0.84	10	5	30	4
4	0.89/0.84	10	5	30	6

Table 7.6: Task set parameters used in the experiments. (Periods and transmission times in milliseconds)

$Camera$	$t \leq 2s$	$2s < t \leq 5s$	$t > 5s$
1	10	5	10
2	10	10	10
3	10	15	10
4	10	20	10

Table 7.7: Periods of each message (ms) during the experiments.

and packet data size is added to each packet, yielding a total Ethernet packet data size of 1010 Bytes.

The task set parameters used in the experiment are shown in Table 7.6, where C_i represents the message transmission time (at 10Mbps) both for the FTT and Ethernet case, T_{i_0}, $T_{i_{min}}$ and $T_{i_{max}}$ are the nominal, minimum and maximum periods respectively and E_i is the message's elastic coefficient.

At the beginning of the experiment all cameras send data at the nominal rate. At time $t = 2s$ camera 1 requests an increase in its QoS. This request is found to be feasible by the elastic guarantee mechanism as long as cameras 3 and 4 decrease their QoS. The elastic task model finds a feasible set with $\{T_1 = 5ms\,; T_2 = 10ms\,; T_3 = 15ms\,; T_4 = 20ms\}$. At time $t = 5s$, the QoS requirement of camera 1 is reset to its nominal value, causing all the cameras to return to their nominal QoS.

The resulting message periods during the experiments are summarized in Table 7.7.

Practical experiments with this traffic pattern were made using both FTT-Ethernet as well as Hub and Switch based Ethernet.

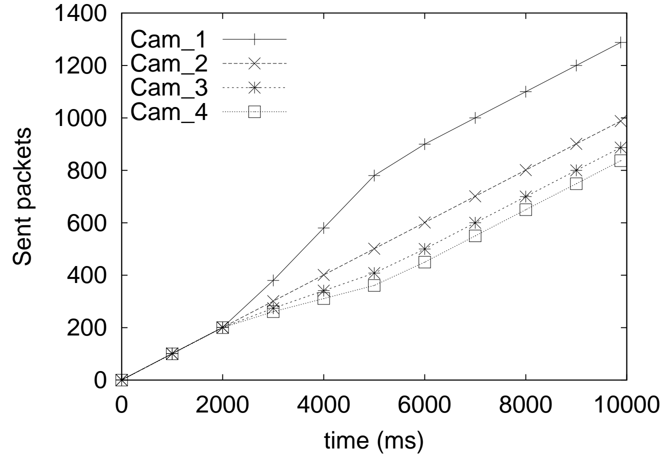


Figure 7.12: Packets sent using FTT-Ethernet.

7.4.2 Results with FTT-Ethernet

In the FTT-Ethernet setup the EC duration was set to 5ms (E=5ms) and the synchronous window was upper bounded to 37% of the EC (LSW=1.85ms), representing a maximum bandwidth of 3.7Mbps available for the synchronous traffic (SMS). This type of traffic was scheduled according to the EDF policy.

As referred in Section 4.3, it is important to characterize and bound the communication overheads per message transmission/reception and include them in each message transmission time, for admission control and scheduling purposes. These overheads depend on both network properties, such as length and number of hubs, as well as on variable latencies imposed by the node's hardware and operating system in the transmission and reception of messages. The combined effect of these aspects was experimentally measured and upper bounded to $50\mu s$. Furthermore, each synchronous message also includes a specific FTT-Ethernet header (Section 7.1.3) with additional control bytes. The resulting packet size, for 1000 data bytes, is 8896 bits resulting in a transmission time of approximately 0.890ms at 10 Mbps.

Figure 7.12 presents the number of packets transmitted by each of the nodes as a function of time, during the experiment. Initially, all cameras send packets at the same rate. However, at time $t = 2s$, the accumulated number of packets sent by each camera starts to diverge as a consequence of a request from camera 1 to increase its QoS. The elastic mechanism finds

$Camera\,ID$	1	2	3	4
Rel. release jitter (avg) (%)	0.53	0.45	1.85	2.83
Absolute release jitter(%)	8.66	7.80	9.79	21.39

Table 7.8: Message jitter with FTT-Ethernet.

a feasible set, which results in an increase of the bandwidth assigned to this camera and a decrease in the bandwidth assigned to cameras 3 and 4. At $t = 5s$, camera 1 requests a QoS reduction to its nominal value. This implicitly causes the QoS of the remaining cameras to be increased to their nominal value, too. Consequently, from that moment on, all cameras start sending packets at the same rate again.

Table 7.8 summarizes the figures concerning the jitter suffered by the messages sent by each of the cameras. The values are presented in percentage and normalized to the respective message period. Despite the occurrence of changes in the message set, these values are relatively small due to the control of transmission instants, preventing the occurrence of message collisions.

7.4.3 Results with hub-based Ethernet

A second experiment was carried out using the same communication infrastructure as in the previous section, but without the use of the FTT-Ethernet layer. In each node a task was configured to reproduce the same data rate described above, at approximately the same instants, but without synchronization.

In this scenario, the Ethernet packet is composed of the data bytes plus a header, 10 bytes long, conveying information required to allow the consumers to identify and reassemble the data. The total packet size amounted to 8384 bits, corresponding to a transmission time of approximately 0.84 ms.

The number of packets sent by each node during the experiment follows a pattern very similar to the one obtained with FTT-Ethernet (fig. 7.12). However, as can it be observed in Table 7.9, there are, now, lost packets and an absolute release jitter that is considerably greater than the one experienced in the previous case.

It is interesting to observe that, despite using a relatively light load (around 35%), the event-triggered nature used in this approach leads to situations where, at some instants, several messages become ready simul-

Camera ID	1	2	3	4
Rel. release jitter (avg) (%)	0.66	1.71	1.13	0.69
Absolute release jitter(%)	66.44	91.65	90.33	90.81
Lost packets (%)	1.65%			

Table 7.9: Message jitter (shared Ethernet).

Camera ID	1	2	3	4
Rel. release jitter (avg) (%)	6.13	0.32	11.00	17.01
Absolute release jitter (%)	66.61	74.61	83.30	126.41

Table 7.10: Message jitter (switched Ethernet).

taneously, originating collisions. In turn, these collisions result in a strong increase in the jitter figures and sometimes in lost packets.

7.4.4 Results with switched Ethernet

In this case, the experimental setup is similar to the one described in the previous section, except that a switch was used to interconnect the nodes, instead of a hub. Again, the number of packets sent by each node during the experiment follows roughly the same pattern as in both previous cases. However, when comparing with the results obtained in the hub-based experiment, there are no lost packets, now. This result was expected, since the use of a switch avoids message collisions and the total bandwidth requested was well below the network maximum throughput.

Concerning the jitter figures, shown in Table 7.10, it can be observed that the values for camera 4 are the greatest among all the experiments, with some messages delayed by more than one period. This phenomenon is explained by the buffering made at the switch ports.

7.4.5 Experimental results analysis

This Section presented the application of the Elastic Task Model to message scheduling on a communication network using the FTT-Ethernet real-time communication protocol. The Elastic Task Model was integrated in the FTT-Ethernet protocol, acting both as QoS and admission control manager, providing a framework in which periodic messages can be served by distinct

QoS during system's normal operation. This model is particularly useful for distributed systems supporting dynamic environments, in which applications have to adapt to the varying operational conditions, leading to variations both in internal computational activities and messages exchanged by the underlying communication system. The policy for selecting a solution during run-time is implicitly encoded in elastic coefficients provided by the user at system configuration time.

The results obtained have shown that the architecture herein presented is able to handle dynamic sets of periodic messages, without jeopardizing the systems timeliness. The same set of experiments was carried out also on hub and switch-based Ethernet, with the same traffic pattern coded in each node. In both of these methods the real-time performance was worse than the one provided by FTT-Ethernet, because either large jitter as well as frame losses.

7.5 Conclusion

This chapter presents the implementation of the FTT paradigm over the Ethernet network protocol.

The synchronous traffic analysis and scheduling only requires a small adaptation, which consists in the addition of a fixed time lapse to message's transmission times to account for the propagation delay that messages may suffer in Ethernet networks. With this adaptation, the FTT-Ethernet implementation follows strictly both the model and analysis developed for the FTT paradigm.

This chapter also presents the asynchronous message system arbitration scheme, which is implementation dependent. The adopted scheme is based in mini-slotting. This scheme enforces the transmission of messages strictly according to their priority, as required by the FTT paradigm. Moreover, this chapter also includes the adaption of the generic response time analysis. Thus, FTT-Ethernet is able to support real-time asynchronous messages.

Some experiments have been carried to assess the performance of the FTT-Ethernet implementation. These experiments were based on the simulation of a video-surveillance system, with video streams having dynamic QoS requirements. Besides FTT-Ethernet, the same set of experiments was carried also over shared and switched Ethernet. The results obtained allow

to conclude that in such conditions FTT-Ethernet performs better, providing collision-free message transmission, with low jitter and no lost packets.

Chapter 8

Conclusions and future work

8.1 Contributions

The research presented in this dissertation focuses on the quest for real-time communication paradigms and protocols able to efficiently support the requirements of flexible real-time distributed systems used in control applications. The following requirements have been identified:

- Support for on-line message scheduling of time-triggered messages based on dynamic requirements;
- Support for on-line message scheduling of time-triggered messages with different scheduling policies;
- Timeliness guarantees concerning the real-time traffic, based on on-line admission control;
- Support for time and event-triggered traffic with temporal isolation;
- Low protocol overhead;
- Scalability

None of the existing protocols efficiently fulfills all these requirements, and thus a new paradigm is proposed, the Flexible Time-Triggered communication paradigm, which attempts to overcome such limitations. Chapter 4, which is the heart of this dissertation, is completely devoted to the study of the FTT paradigm. The system architecture is specified, including the

software architecture both in master and slave nodes, the required data structures and the scheduling and arbitration mechanisms. Moreover, schedulability tests for the real-time traffic, both synchronous and asynchronous, are also presented .

The proposed FTT communication paradigm architecture is based on on-line centralized scheduling of the synchronous traffic, combined with a master/multi-slave transmission control technique. The arbitration mechanism used for the asynchronous traffic is network dependent, and thus it is not specified by the FTT paradigm. However, it is required to be deterministic, i.e., messages should be transmitted in a bounded time and strictly according to their priority.

Having the communication requirements and scheduling centralized in a single node facilitates changes on the message requirements, since there is no need to perform complex and resource demanding operations to update distributed databases and synchronize events. A simple binary mutual exclusion primitive is used to provide atomic updates on the message set properties database. On other hand, the transmission control technique is independent of the particular scheduling algorithm employed, therefore changes to the message set properties or even to the message scheduling policy are only felt within the master node. Since slave nodes strictly follow the EC-Schedule conveyed in the TM, they need not to be explicitly aware of the current communication requirements or about the scheduling policy being used.

Moreover, having the communication requirements centralized in a single node also facilitates the integration of on-line admission control, since the communication requirements are locally available, thus reducing the difficulty of the integration of schedulability tests.

Other important feature of the FTT paradigm is the support for synchronous and asynchronous traffic, with temporal isolation. This framework allows to reconcile the benefits of the time-triggered and event-triggered models. This is particularly relevant since in many real-time distributed systems there are commonly activities that occur at pre-defined instants in time at a rate determined by the dynamics of the environment under control, which are more efficiently handled by the time-triggered communication model, and asynchronous activities that are more efficiently handled by the event-triggered communication model.

The FTT paradigm is not tied to any particular medium-access protocol. Any communication infrastructure that supports message broadcasts and bounded message transmission times can be used. Furthermore, if desired, the native MAC arbitration mechanism may be bypassed by the FTT arbitration mechanism. For instance, the FTT-CAN implementation relies on the native CAN MAC to perform arbitration within the EC, reducing the protocol overhead, while in the FTT-Ethernet implementation the native Ethernet MAC is completely avoided. The possibility of using different communication mediums contributes to the communication system flexibility, since it allows to choose the communication medium that better serves the particular application requirements. For instance, CAN, which supports up to 8 data bytes per frame, can be used in applications that need to exchange short data packets. On the other hand, Ethernet, which supports up to 1500 data bytes per frame, can be used in applications requiring the exchange of large blocks of data. The same is true concerning the bandwidth required. For instance, CAN may be used in applications that require a bandwidth up to 1 Mbps, while applications requiring higher bandwidths can be supported by Ethernet.

Finally, the FTT paradigm allows to achieve high bandwidth efficiency due to the combination of the following factors:

- A master/multi-slave transmission control technique, that allows to reduce considerably the protocol overhead associated with the traditional master-slave technique, since a single control message may trigger several synchronous messages;
- The existence of on-line admission control and dynamic traffic scheduling mechanisms, allowing to change on-line the communication requirements, an thus to adapt the communication requirements to suit the effective needs of the system;
- The possibility of using more efficient scheduling policies, such as EDF.

This set of properties exhibited by the FTT paradigm support the thesis, stated in section 1.3, that it is possible to combine in the same communication system different traffic with hard, soft and non-real-time timeliness requirements and change its properties and/or the respective scheduling pol-

icy during system run-time, without relinquishing predictability guarantees and achieving efficient use of network bandwidth.

Many real-time protocols broadly used at the field level provide limited bandwidth, frequently up to 1Mbps. The recent expansion on the application domains of fieldbus technologies (e.g. automotive, machine tools, process and manufacturing industry) in which there is an increasing number of interconnected devices with increasing level of integration, results in a larger amount of data to be shared and therefore the available bandwidth becomes scarce. On other hand, certain applications contain different message streams that should be handled with similar QoS, a feature that is not supported by the scheduling schemes of several of such protocols. Scheduling policies have a particular relevance in this issue, since they impact both on the maximum bandwidth utilization that can be achieved with timeliness guarantees and also on the QoS that can be delivered to the distinct message streams, in terms of either network delay and jitter. The FTT paradigm is not tied to any particular scheduling policy. To assess the impact of the scheduling policy in the network utilization both fixed priority (RM) and dynamic priority (EDF) schedulers were implemented. For the FTT-CAN case, the results obtained, both experimental and simulation, show that it is possible to achieve significant gains in bandwidth utilization by using EDF instead of the RM scheduling policy. For example, with a synchronous bandwidth limited to 80%, simulation results with randomly generated sets of messages show an utilization gain of 6% when EDF is used instead of of RM for the scheduling of the synchronous messages. Considering the sufficient schedulability conditions presented in Chapter 4, the gain in the respective threshold is 20% higher for EDF than for RM.

In real-time systems research, schedulability analysis deserves a particular attention, since the timeliness requirements of real-time activities must be fulfilled in all anticipated circumstances. Systems that support dynamic changes to the activity requirements, such as FTT systems, present demanding challenges in what concerns this issue. In fact, such analysis must be performed on-line, frequently in nodes with constrained resources, nevertheless with low latency, in order to not compromise the system response time to change requests. Concerning the synchronous traffic, a previously pro-

posed generic task model was adapted to the FTT framework, allowing the use of well known utilization based analysis which, despite being pessimistic, have very low computational complexity and thus are well suited for on-line use. With respect to the asynchronous traffic, a response-time based analysis was derived for the generic paradigm and then adapted for both CAN and Ethernet implementations. Moreover, the asynchronous traffic analysis also provides upper bounds to the memory requirements for messages with no deadlines or deadlines longer than the respective minimum inter-arrival time, allowing the communication system to reserve in advance the necessary number of buffers. This feature considerably eases the application development, since the occurrence of message buffering becomes completely transparent to the application.

In many application domains there has been a trend towards higher flexibility in order to support dynamic configuration changes arising from evolving requirements and on-line Quality-of-Service (QoS) management. The FTT framework provides an adequate support for such requirements since relevant parameters of messages, such as periods, can be dynamically adjusted. This subject has been explored in this thesis, both in conceptual and implementation terms. It has been shown that arbitrary QoS management policies can be easily integrated in the FTT architecture, provided that QoS parameters can be mapped onto standard properties such as periods and deadlines. A prototype implementation shows, for the particular case of a video-based system, the effectiveness of this approach in dynamically assigning specific QoS parameters to specific video streams while automatically allocating the best QoS possible to the remaining video streams.

The flexibility exhibited by the FTT paradigm also concerns the support for distinct platforms, with wide ranges of performance capabilities. The FTT paradigm has been implemented over Controller Area Network and Ethernet, leading respectively to the FTT-CAN and FTT-Ethernet protocols. The FTT-CAN protocol targets mainly real-time applications based on low processing-power micro-controllers, typically found in distributed embedded systems. Due to the constraints presented by this environment, in particular concerning the limited resources available (network bandwidth, CPU processing power, memory), the implementation of the FTT-CAN protocol was

biased towards simplicity and resource economy. A prototype implementation made on 11MHz 8051-based boards was successfully performed, showing that the price to pay for the flexibility of the FTT paradigm is in the range of current low-end embedded systems. On the other hand, Ethernet is nowadays considered as a strong candidate to support demanding applications, ranging from embedded command and control systems to computer vision, robotics, process supervision, etc. This observation fostered the implementation of the FTT-Ethernet protocol. These applications are particularly demanding concerning the flexibility of the communication subsystem, thus in the scope of the FTT-Ethernet protocol most of the work addressed QoS management. A prototype implementation shows the possibility of using elaborated QoS management mechanisms, such as the Elastic Task Model, originally developed for task scheduling in single microprocessors, leading to a system highly dynamic but still capable of providing real-time guarantees.

8.2 Future research

Some promising extensions to the work developed in the scope of this thesis are:

Implementation of the FTT-Ethernet over switched Ethernet

Although the use of a switch by itself is not enough to support real-time guarantees on Ethernet, the FTT-Ethernet protocol could take advantage of it. In first place, in a switch-based network it is not necessary to enforce the start of message transmissions in disjoint time instants. Thus, in this case neither it is necessary to include the message lengths in the trigger message nor it is necessary to set-up timers associated to each message transmission in sender nodes. Thus, the implementation would consume less network bandwidth and less overhead in slave nodes. In second place, the asynchronous message arbitration is based on mini-slotting, which is a mechanism that consumes bandwidth. Switches may provide prioritized message transmission (IEEE 802.1p), but the number of such priorities (eight at most) is not sufficient to implement an efficient priority-based scheduling mechanisms. Nevertheless, such possibility could help in enhancing the performance of the asynchronous message arbitration used in the FTT-Ethernet protocol. For instance, assigning distinct priority levels to each traffic class (hard, soft

and non-real-time) can potentially reduce the arbitration overhead.

Wireless implementation of the FTT paradigm

Wireless transmission has been used for years to link mobile devices such as mobile robots and automated guided vehicles to their respective control computers. Besides the mobility issue, for which wireless is unquestionably the most adequate approach, currently this type of technology is also regarded as the next logical step in the evolution of the fieldbus in industrial automation. In fact, one of the main reasons of the success of fieldbuses in this domain is the drastic reduction of wiring complexity, and thus wireless technologies just constitute another advance in the same direction. The IEEE 802.11 standard for local area networks defines an extension of Ethernet to the wireless medium, and thus it is an interesting challenge to investigate the possibility to implement the FTT paradigm on this protocol and to study how the FTT paradigm can tackle with some specific problems of the wireless technology, deriving from the natural openness concerning the participating nodes. For instance, wireless networks usually exhibit considerably higher bit-error rates and more frequent and longer inaccessibility periods than wired networks.

Joint scheduling of synchronous and asynchronous message streams

In real world DCCS applications communication activities that are periodically activated (synchronous) and others that result from unforeseen events (asynchronous), e.g. alarms, are often found. However the nature of the communication activities does not necessarily constrain their timeliness requirements; critical activities can be either of synchronous or asynchronous nature. In the FTT paradigm the synchronous and asynchronous traffic are scheduled independently. Although there is support for hard real-time asynchronous traffic, it requires the static reservation of a share of the EC to exclusive use by the asynchronous traffic, performed during system set-up, which is not an optimum solution since it reduces the schedulability of synchronous traffic. Therefore an important system schedulability enhancement can potentially be achieved by employing methodologies allowing to perform the joint scheduling of both of synchronous and asynchronous message streams. In particular, the evaluation of the potential of sporadic servers in this context seems an interesting line of research.

Routing protocols

Real-time distributed applications are becoming increasingly complex, due to both an increase in the number of interconnected devices and increased amount of data to be shared between them. A well-known technique used to manage such framework consists in decomposing the system in different functional units, comprising e.g. sets of sensors, actuators and controllers that cooperate closely to achieve a particular goal. The components of these functional units are interconnected by independent sub-networks. The whole system can be modeled by a set of such functional units, hierarchically organized. The communication between different functional units is performed by gateway nodes that filter the traffic going inward and outward.

Timeliness requirements can be found either in the communication between functional units and within the functional units themselves. Therefore this approach leads to a hierarchical real-time scheduling problem, with real-time messages found at the different system levels. There is ongoing research in this field, particularly concerning task scheduling in microprocessors, and it seems an interesting line of research to study the compatibility of such results with the FTT architecture. On the other hand, there are also some recent research work in the scope of general networks (e.g. IP based) concerning the implementation of the Publisher/Subscriber model using content-based addressing/routing. It seems also an interesting line of research to evaluate the suitability of the FTT architecture to support such framework.

Bibliography

[ABR$^+$93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engeneering Journal*, 8(5):284–292, September 1993.

[ABRW91] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta,GA,USA, May 1991.

[AF98] Luis Almeida and J. A. Fonseca. The planning scheduler: compromising between operational flexibility and run-time overhead. In *Proceedings of the INCOM'98 (IFAC International Symposium on Information Control in Manufacturing)*, Nancy, France, June 1998.

[AF99] Luis Almeida and J. A. Fonseca. Schedulability analysis in the fip fieldbus accounting for inserted idle-time. In *Proceedings of Euromicro Real-Time Systems Conference (ECRTS'99) WIP Session*, York, England, June 1999.

[AF01] Luis Almeida and J. Fonseca. Analysis of a simple model for non-preemptive blocking-free scheduling. In *Proceedings Euromicro Conference on Real-Time Systems RTS'01)*, Delft, Netherlands, June 2001.

[AFF98] Luis Almeida, Jose Fonseca, and Pedro Fonseca. Flexible time-triggered communication on a controller area network. In *Proceedings of Work-In-Progress Session of RTSS'98 (19th*

IEEE Real-Time Systems Symposium), Madrid, Spain, December 1998.

[AFF99] Luis Almeida, J. A. Fonseca, and P. Fonseca. A flexible time-triggered communication system based on the controller area network: Experimental results. In *Proceedings of 3rd International Conference on Fieldbus Systems and their Applications (FET'99)*, Magdeburg, Germany, 1999.

[Alm99] Luis Almeida. *Flexibility and Timeliness in Fieldbus-based Real-Time Systems*. PhD thesis, University of Aveiro, Aveiro, Portugal, 1999.

[APF99] Luis Almeida, R. Pasadas, and J. A. Fonseca. Using a planning scheduler to improve flexibility in real-time fieldbus networks. *IFAC Control Engineering Practice*, 7:101–108, 1999.

[APF02] Luis Almeida, P. Pedreiras, and J. A. Fonseca. The ftt-can protocol: Why and how. *IEEE Transactions on Industrial Electronics*, 49(6), December 2002.

[ATFV01] Luis Almeida, E. Tovar, J. A. Fonseca, and F. Vasques. Schedulability analysis of real-time traffic in worldfip networks: an integrated approach. In *IEEE Transactions on Industrial Electronics*, 2001.

[Aud93] N. Audsley. *Flexible Scheduling of Hard Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, 1993.

[BA00] G. Buttazzo and L. Abeni. Adaptive rate control through elastic scheduling. In *39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[BLA98] Giorgio Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 286–295, Madrid, Spain, December 1998.

[BLCA02] Giorgio Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. In *IEEE Transactions on Computers*, volume 51 of *3*, pages 289–302, March 2002.

[BM01] L. Lo Bello and O. Mirabella. Design issues for ethernet in automation. In *Proceedings of ETFA'01 - 8th IEEE International Conference on Emerging Technologies and Factory Automation*, Antibes, France, October 2001.

[BMK88] D. R. Boggs, J. C. Mogul, and C. A. Kent. Measured capacity of an Ethernet: myths and reality. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, volume 18 of *4*, pages 222–34, 1988.

[But97] Giorgio Buttazzo. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, 1997.

[CCBM02] A. Carpenzano, R. Caponetto, L. Lo Bello, and O. Mirabella. Fuzzy traffic smoothing: an approach for real-time communication over ethernet networks. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems*, pages 241–248, Vasteras, Sweden, August 2002.

[CDK94] G. Coulouris, J. Dollimore, and Tim Kindberg. *Distributed Systems - Concepts and Design.* International Computer Science Series. Addison-Wesley Longman, Inc., 2nd edition edition, 1994.

[CEN96] CENELEC, European Committe for Electrotechnical Standadisation. *European Standard EN 50170: Fieldbus: Vol. 1:P-Net; Vol.2: PROFIBUS; Vol.3: WorldFIP*, 1996.

[CM95] C. Cardeiras and Z. Mammeri. A schedulability analysis of task and network traffic in distributed real-time systems. *The Journal of the International Measurement Conference IMEKO*, 1995.

[CMTV02] Salvatore Cavalieri, Salvatore Monforte, Eduardo Tovar, and Francisco Vasques. Multi-master profibus dp modelling and worst case analysis-based evaluation. In *XV IFAC World Congress on Automatic Control*, Barcelona, Spain, July 2002. IFAC.

[Con01] FlexRay Consortium. *FlexRay Requirements Specification Version 1.9.7*, September 2001.

[Cou92] R. Court. Real-time ethernet. In *Computer Communications*, volume 15, pages 198–201, April 1992.

[Dec01] J-D. Decotignie. A perspective on ethernet as a fieldbus. In *Proceedings of the 4th FeT'2001 - International Conference on Fieldbus Systems and their Applications*, pages 138–143, Nancy, France, November 2001.

[Dwo98] J. Dworzecki. Ordonnancement déterministe des tâches périodiques en présence de contraintes temporelles et de sucessions. In *Proceedings of Real-Time Embedded Systems RTS 98*, Paris, France, 1998.

[F+98] P. Fonseca et al. A dynamically reconfigurable can system. In *Proceedings of ICC'98 (International CAN Conference)*, San Jose, USA, November 1998.

[Foh93] G. Fohler. Changing operational modes in the context of pre run-time scheduling. In *IEIC Transactions on Information and Systems, Special Issue on Responsive Computer Systems*, pages 1333–1340, November 1993.

[FSMF98] P. Fonseca, F. Santos, A. Mota, and J. A. Fonseca. A dynamically reconfigurable can system. In *5th International CAN Conference*, San Jose, California, USA, 1998.

[GGAB01] Paolo Gai, M. Giorgio, L. Abeni, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[GH98] J. C. Palencia Gutierrez and Michael Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 9th IEEE Real-Time Systems Symposium*, page 26, 1998.

[GJ75] R. Garey and S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal of Computing*, 4(4):397–411, 1975.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. Research Report 2966, INRIA, September 1996.

[IEC00] IEC International Electrotechnical Comittee. *IEC International Standard 61158: Fieldbus standard for use in industrial control systems - Type 1: Existing IEC TS61158 parts3-6 (Foundation Fieldbus H1); Type 2: ControlNet; Type 3: PROFIBUS; Type 4:P-Net; Type 5: Fieldbus Foundation HSE; Type 6: SwiftNet; Type 7: WorldFIP; Type 8: Interbus-S*, 2000.

[IEEa] IEEE. *IEEE 802.3 10BASE3 standard.*

[IEEb] IEEE. *IEEE 802.3 10BASE5 standard.*

[IEEc] IEEE. *IEEE 802.3, 2000 Edition.*

[IEEd] IEEE. *IEEE 802.3c 100BASE-T standard.*

[IEEe] IEEE. *IEEE 802.3c 1BASE5 StarLan standard.*

[IEEf] IEEE. *IEEE 802.3i 10BASE-T standard.*

[IEEg] IEEE. *IEEE 802.3z 1000BASE-T standard.*

[IEE82] IEEE. *DIX Ethernet V2.0 specification*, 1982.

[Int00] International Organization for Standardization. *ISO/WD111898-4. Road Vehicles - Control Area Network (CAN) - Part 4: Time-Triggered Communication*, 2000.

[ISO93] ISO-11898. *Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication*, 1993.

[ISO94a] ISO. Time critical communication architectures - user requirements. Technical Report ISO TR 12178, ISO, Geneva, 1994.

[ISO94b] ISO-11519-2. *Road vehicles - Low-speed serial data communication - Part 2: Low-speed controller area network (CAN)*, 1994.

[JN01] J. Jasperneit and P. Neumann. Switched ethernet for factory communication. In *Proceedings of ETFA2001 - 8th IEEE International Conference on Emerging Technologies and Factory Automation*, Antibes, France, October 2001.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Computer Journal*, 29(5):390–395, October 1986.

[KG94] H. Kopetz and G. Grünsteidl. Ttp - a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1), January 1994.

[KJK98] Y. S. Kim, S. Jeong, and W. H. Kwon. A pre-run-time scheduling method for distributed real-time systems in a fip environment. *Control Eng. Practice*, 6:103–109, 1998.

[Kop93] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered ? *IEICE Transactions on Information and Systems*, 11(76):1325–32, November 1993.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[Kop99] H. Kopetz. Specification of the ttp/c protocol, version 0.5. Technical report, TTTech Computertechnik AG, July 1999. Document edition 1.0.

[KS00] S-K. Kweon and K. G. Shin. Achieving real-time communication over ethernet with adaptive traffic smoothing. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, pages 90–100, Washington DC, USA, June 2000.

[KSZ99] S-K. Kweon, K. G. Shin, and Q. Zheng. Statistical real-time communication over ethernet for manufaturing automation sys-

tems. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*, Vancouver, Canada, June 1999.

[LA99] H. Lonn and J. Axelsson. A comparison of fixed-priority and static scheduling for distributed automotive control applications. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems - ECRTS'99*, pages 142–149, 1999.

[Leh90] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[LK98] M. Livali and J. Kaiser. Edf consensus on can bus access for dynamic real-time applications. In 6th *International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'98*, Orlando, FL, USA, April 1998.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[LR93] G. LeLann and N. Rivierre. Real-time communications over broadcast networks: the csma-dcr and the dod-csma-cd protocols. Technical report, INRIA Report RR1863, 1993.

[LRM96] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. In *Multimedia Japan 96*, Japan, April 1996.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm:exact characterization and average case behaviour. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[LW82] J. Leung and J. Withehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4), 1982.

[MAR+00] P. Marti, J.C. Aguado, F. Rolando, M. Velasco, P. Colomar, and J.M. Fuertes. Distributed supervision and control

of filedbus-based industrial processes. In *Proceedings of 2000 IEEE International Workshop on Factory Communication Systems WFCS'00*, pages 11–18, Oporto, Portugal, September 2000. IEEE.

[Mar02] P. Marti. *Analysis and Design of Real-Time Control Systems with Varying Control Timing Constraints*. PhD thesis, Universitat Politecnica de Catalunya, Barcelona, Spain, July 2002.

[MD78] A. Mok and M. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the 7th Texas Conference on Computer Systems*, 1978.

[MF01] E. Martins and J. A. Fonseca. Improving flexibility and responsiveness in ftt-can with a scheduling coprocessor. In *Proceedings of 4th IFAC International Conference on Filedbus Systems and their Applications (FET'01)*, Nancy, France, November 2001.

[MF02] F. Bogenbergerand B. Muller and T. Fuher. Protocol overview. In *Proceedings of the 1st FlexRay International Workshop*, Munich, Germany, April 2002.

[MK85] M. Molle and L. Kleinrock. Virtual time csma: Why two clocks are better than one. *IEEE Transactions on Communications*, 33(9):919–933, 1985.

[Mon00] F. Monk. The future of ethernet in the manufacturing environment. In *IEEE International Workshop on Factory Communication Systems WFCS'00 - Wip Session*, Oporto, Portugal, September 2000.

[MZ95] N. Malcom and W. Zhao. Hard real-time communication in multiple-access networks. In Kluwer Academic Publishers, editor, *Real Time Systems*, volume 9, pages 75–107, Boston, USA, 1995.

[Nat00] M. Di Natale. Scheduling the can bus with earliest deadline techniques. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2000.

[NHNP01] Thomas Nolte, H. Hansson, C. Norstrom, and S. Punnekkat. Using bit-stuffing distributions in can analysis. In *IEEE Real-Time Embedded Systems Workshop*, December 2001.

[OODVA97] Inc ODVA Open DeviceNet Vendor Association. *DeviceNet Specification - release 2.0, Vol. I and II.* USA, 1997.

[PA00] Paulo Pedreiras and Luis Almeida. Combining event-triggered and time-triggered trafic in FTT-CAN: analysis of the asynchronous mensaging system. In *Workshop on Factory Communication Systems - WFCS'00*, Porto, Portugal, 2000.

[PAG02] Paulo Pedreiras, Luis Almeida, and Paolo Gai. The ftt-ethernet protocol: Merging flexibility, timeliness and efficiency. In *Proceedings of the 14th IEEE Euromicro Conference in Real-Time Systems*, Vienna, Austria, June 2002.

[PB97] P. Pedro and A. Burns. Worst case response time analysis real-time sporadic traffic in fip networks. In *Proceedings of Workshop Real-Time Systems*, pages 3–10, June 1997.

[PBG99] M. Peller, J. Berwanger, and R. Griesbach. Byteflight specification, version 0.5. http://www.byteflight.com, October 1999.

[PD00] Bernard Pavard and Julie Dugdale. An introduction to complexity in social science. In GRIC-IRIT, editor, *http://www.irit.fr/COSI/training/complexity-tutorial/complexity-tutorial.htm*, Toulouse, France, 2000.

[Ped99] Paulo Sérgio M. Pedro. *Schedulability of Mode Changes in Flexible Real-Time Distributed Systems.* PhD thesis, Universisty of York, Dep. of Computer Science, UK, York, 1999.

[Pim90] J. Pimentel. *Communication Networks for Manufacturing.* Prentice-Hall International Editions, 1990.

[RN93] P. Raja and G. Noubir. Static and dynamic polling mechanisms for fieldbus networks. *ACM Operating Systems Review*, 27(3), 1993.

[Rob91] Robert Bosch GmbH, Stuttgart, Germany. *CAN Specification version 2.0*, 1991.

[RS94] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating system support for real-time systems. *Proceedings of IEEE*, 82(1):55–67, January 1994.

[S+96] J. A. Stankovic et al. Strategic directions in realtime and embedded systems. *ACM Computing Surveys*, 28(4):751–763, 1996.

[SJH02] T. Skeie, S. Johannessen, and O. Holmeide. The road to an end-to-end deterministic ethernet. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'02)*, pages 3–9, Vasteras, Sweeden, August 2002.

[SLST99] J. Stankovic, C. Lu, S. H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 11–20. IEEE, 1999.

[Son01] Y. Song. Time constrained communication over switched ethernet. In *Proceedings of the 4th FeT'2001 - International Conference on Fieldbus Systems and their Applications*, pages 152–159, Nancy, France, November 2001.

[Spu96] M. Spuri. Analysis of deadline scheduled real-time systems. Research Report 2772, INRIA, January 1996.

[SR88] J. Stankovic and K. Ramamritham. Tutorial on hard real-time systems. *IEEE Computer Society Press*, 1988.

[SR92] C. Shen and K. Ramamritham. Scheduling in real-time systems. Technical report, Department of Computer Science, University of Massachusetts, 1992.

[SS85] Y. Shimokawa and Y. Shiobara. Real-time ethernet for industrial applications. In *Proceedings of IECON*, pages 829–834, 1985.

[SS93] L. Sha and S. Sathaye. A systematic approach to designing distributed real-time systems. *IEEE Computer*, 26(9):68–78, 1993.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1995.

[TBW95] K. Tindell, A. Burns, and A.J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, September 1995.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 50(2–3), 1994.

[TC99] J.-P. Thomesse and M. L. Chavez. Main paradigms as a basis for current fieldbus concepts. In *Proceedings of FeT'99 (International Conference on Fieldbus Technology)*, Magdeburg, Germany, September 1999.

[Tho93] J.-P. Thomesse. Time and industrial local area networks. In *Proceedings of COMPEURO'93*, Paris, France, 1993.

[Tho98] J.-P. Thomesse. The fieldbuses. In *Annual Reviews in Control*, volume 22, pages 35–45, 1998.

[Tho99] J-P. Thomesse. Filedbus and interoperability. *Control Engeneering Practice*, 7(1):81–94, 1999.

[THW94] K. Tindell, H. Hansson, and J. Wellings. Analysing real-time communication: Controller area network (can). In *Proceedings of 15th IEEE Real-Time Systems Symposium RTSS'94*, 1994.

[TTT] Tttech computertechnik ag. http://www.tttech.com.

[TV98a] E. Tovar and F. Vasques. Enhancing p-net real-time properties using priority queuing mechanisms. In *Proceedings of the WIP Session of the 4th IEEE Real-Time Technologies and Applications Symposium*, pages 27–30, Denver, Colorado, USA, June 1998.

[TV98b] E. Tovar and F. Vasques. Scheduling real-time communications with p-net. *Digest of the IEE Real-Time Systems Colloquium*, 98(306):9/1–9/5, April 1998.

[TV99a] E. Tovar and F. Vasques. From task scheduling in single processor environments to message scheduling in a profibus fieldbus network. In *Proceedings of the 7th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'99)*, pages 339–352, San Juan, Puerto Rico, April 1999.

[TV99b] E. Tovar and F. Vasques. Real-time fieldbus communications using profibus networks. *IEEE Transactions on Industrial Electronics*, 46(6):1241–1251, December 1999.

[VC94] C. Venkatramani and T. Chiueh. Supporting real-time traffic on ethernet. In *Proceedings of IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[VJ94] F. Vasques and G. Juanole. Pre-run-time schedulability analysis in fieldbus networks. In *Proceedings of IEEE IECON 94*, pages 1200–1204, 1994.

[VR01] P. Veríssimo and Luís Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[Wol00] M. Wollschlaeger. A framework for filedbus management using xml descriptions. In *Proceedings of 2000 IEEE International Workshop on Factory Communication Systems WFCS'00*, pages 3–10, Oporto, Portugal, September 2000. IEEE.

[Zim80] H. Zimmermann. Osi reference model: The iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[ZPS99] Khawar M. Zuberi, Padmanabhan Pillai, and Kang G. Shin. EMERALDS: a small-memory real-time microkernel. In *Symposium on Operating Systems Principles*, pages 277–299, 1999.

[ZR87] W. Zhao and K. Ramamritham. Virtual time csma protocols for hard real-time communications. In *IEEE Transactions on Software Engineering*, volume SE-13 of *8*, pages 938–952, 1987.

[ZS95] K. Zuberi and K. Shin. Non-preeptive scheduling messages on controller area network for real-time control applications. In *Real-Time Technology and Applications Symposium (RTAS'95)*, pages 240–249, May 1995.

[ZS97] K. Zuberi and K. Shin. Scheduling messages on controller area network for real-time cim applications. *IEEE Transactions on Robotics and Automation*, 13(2):310–314, April 1997.

Appendix A

List of publications and communications

In the scope of the research developed towards the preparation of this PhD thesis, the following journal and conference articles have been published:

A.1 Journal articles

- L. Almeida, Paulo Pedreiras, Jose A. Fonseca. "The FTT-CAN protocol: Why and How". To appear in IEEE Transactions on Industrial Electronics- special section on Factory Communication Systems, Dec. 2002;

- P. Pedreiras, L. Almeida. "EDF Message Scheduling on Controller Area Network". IEE Computing & Control Engineering Journal, Volume 13, Number 4, pp.163-170, August 2002;

- J. Ferreira, P. Pedreiras, L. Almeida, J. Fonseca. "The FTT-CAN protocol: improving flexibility in safety-critical systems". IEEE Micro, Volume 22, Number 4, pp.46-55, July/Aug 2002;

- P. Pedreiras, L. Almeida, J. A. Fonseca, "A Proposal To Improve The Responsiveness Of The Synchronous Messaging System In FTT-CAN". University of Aveiro's Electronics and Telecommunication Journal, Vol. 3, N.2, October 2000;

A.2 Conference papers

- E. Martins, J. Ferreira, L. Almeida, P. Pedreiras, J. A. Fonseca, “An Approach to the Synchronization of Backup Masters in Dynamic Master-Slave Systems”. Proceedings of the WIP session of the 23rd IEEE International Real-Time Systems Symposium, Austin, TX. (USA), December 3-5 2002.
- J. Fonseca, J. Ferreira, M. Calha, P. Pedreiras, L. Almeida. "Issues on Task Dispatching and Master Replication in FTT-CAN". Proceedings of IEEE AFRICON 2002, South Africa, Out/2002.
- J. Ferreira, P. Pedreiras, L. Almeida and J. Fonseca. "Achieving Fault-Tolerance in FTT-CAN". Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'02), pp. 125-132.Vesteras, Sweden, Aug/2002;
- P. Pedreiras, P. Gai, G. Buttazzo, L. Almeida. "FTT-Ethernet: a platform to implement the Elastic Task Model over message streams". Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'02), pp. 225-232. Vesteras, Sweden, Aug/2002;
- P. Pedreiras, L. Almeida and P. Gai."The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency". Proceedings of the 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, Jun/2002;
- P. Pedreiras, L. Almeida. "Flexibility, Timeliness and Efficiency in Ethernet". Proceedings of 1st International Workshop on Real-Time LANs in the Internet Age (RTLIA2002). Vienna, Jun/2002;
- P. Pedreiras. Session rapport on new communication paradigms. Proceedings of 1st International Workshop on Real-Time LANs in the Internet Age (RTLIA2002). Vienna, Jun/2002;
- P. Pedreiras, L. Almeida, "Flexible Scheduling on Controller Area Network". Proceedings of RTS'2002- 10eme Conference Internationale Sur Les Systemes Temps Reel, Paris, France, Mars/2002;
- P. Pedreiras, L. Almeida, "A Practical Approach to EDF Scheduling on Controller Area Network". Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop, London, Dec/2001;

- P. Pedreiras, L. Almeida, "Asynchronous Communication on FTT-CAN: experimental results". Proceedings of FeT'2001 - 4th FeT IFAC Conference, Nancy, France, Nov/2001;
- J. Ferreira, P. Pedreiras, L. Almeida, J. A. Fonseca, "FTT-CAN Error Confinement". Proceedings of FeT'2001 - 4th FeT IFAC Conference, Nancy, France, Nov/2001;
- L. Almeida, J.A. Fonseca, A. Mota, P. Fonseca, E. Martins, P. Pedreiras, J. Ferreira and F. Coutinho." Flexibility, Timeliness and Efficiency in Fieldbus Systems: The DISCO Project". Proceedings ETFA'01 - IEEE Conf. on Emerging Technologies for Factory Automation, Juan-Les-Pins, France, Oct. 2001;
- P. Pedreiras, L. Almeida, J. A. Fonseca, "Improving the Responsiveness of the Synchronous Messaging System In FTT-CAN". Proceedings of DCCS 2000, Sydney, Australia, Dec/2000.
- P. Pedreiras, L. Almeida, "Combining event-triggered and time-triggered traffic in FTT-Can: analysis of the asynchronous messaging system". Proceedings of WFCS 2000, Porto, Set/2000;
- J. Fonseca, E. Martins, L. Almeida, P. Pedreiras, P. Neves, "Flexible Time-Triggered Protocol for CAN - New Scheduling and Dispatching Solutions". Proceedings of ICC 2000, New Orleans, USA, Jun/2000.

Appendix B

List of acronyms

BA Bus Arbitrator

BAT Bus Arbitrator schedule Table

CAN Controller Area Network

CC Communication Cycle

CD Compel Data

CIP Control and Information Protocol

CSMA Carrier-Sense Multiple Access

DCCS Distributed Computer-Control System

DL Data Link

DLL Data-link layer

DM Deadline Monotonic

E Elementary Cycle Duration

EC Elementary Cycle

EDF Earliest Deadline First scheduling policy

ET Event-Triggered

FDL Fieldbus Data Link

FTT Flexible Time-Triggered protocol

FTT-CAN Flexible Time-Triggered protocol on CAN

FTT-Ethernet Flexible Time-Triggered protocol on Ethernet

LAS Link Active Scheduler

LAW Minimum Length of the Asynchronous Window

law(i) Length of the Asynchronous Window of EC i

LCM Least Common Multiple

LL Least Laxity

LS Link Scheduling

LSW Upper bound for the Length of the Synchronous Window

lsw(i) Length of the Synchronous Window of ECi

MAC Medium Access Control

MC Macro-Cycle

MEDL Message Descriptor List

NIC Network Interface Card

OSI Opens System Interconnection

OUI Organizational Unique Identifier

PLC Programmable Logic Controller

PN Probe Node

PT Pass Token

QoS Quality of Service

RM Rate Monotonic scheduling policy

TD Time Distribution

TDMA Time-Division Multiple Access

TM Trigger Message

TT Time-Triggered

Appendix C

FTT-Ethernet sample application

The sample code presented below shows the code required to generate the master node program of an application using both synchronous and asynchronous messages. The code is related to the FTT-Ethernet implementation.

```
/******************************************************************/
/* FTT-Ethernet; Paulo Pedreiras; Jul/2002                        */
/*                                                                */
/* Test application 1 (Master):                                    */
/*                                                                */
/* This test application configures a set of messages             */
/* both peridic and aperiodic.                                     */
/* SET1:  Some "slow" messages allow visualisation of its          */
/* contents for checking if everything ok.                         */
/******************************************************************/
/***********************************/
/* FTT related defines and includes */
/***********************************/
#define EC_LEN (long)20000 /* EC length (us) */
#define EDF_SCHED /* Select Earliest Deadline First Scheduler */

#include "fttetm1.c"
/****************************/
/* Application related stuff */
/****************************/
#define APP_DEBUG /* Debug information ON */

/*********/
```

```
/* main() */
/*********/
int main(int argc, char **argv)
{
    /* Auxiliary variable used to append messages to the SRT */
    SRDB_SRT_mesgtype SRT_aux_var; SRDB_ART_mesgtype ART_aux_var;

    /**********************/
    /* Init the ftt system */
    /**********************/
    ftt_minit();

    /*************************/
    /* Set-up the message set */
    /*************************/
    /* This set has a high load, with "quick" messages and one */
    /* slow message to allow displaying on the screen */
    cprintf("\n Building message set (SMS1)...");
    SET_SMESG_PROP(&SRT_aux_var,1,512,1,1,0); /* id,size,period,deadline,init */
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,2,1024,1,1,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,3,512,2,2,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,8,512,5,5,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,14,512,7,7,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,10,512,10,10,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,4,512,9,9,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,11,512,11,11,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,12,512,12,12,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,16,512,16,16,0);
    SRDB_SRT_addmesg(&SRT_aux_var);
    SET_SMESG_PROP(&SRT_aux_var,18,512,18,18,0);
    SRDB_SRT_addmesg(&SRT_aux_var);

    /* Slow message (5s period for EC=20ms) */
    SET_SMESG_PROP(&SRT_aux_var,19,100,250,250,0);
    SRDB_SRT_addmesg(&SRT_aux_var);

    cprintf(" Finished building message set (SMS1)!");

    /* Asynchronous messages :  Set 1 */
    cprintf("\n Building message set (AMS1)..."); /* Add asynch.  messages */
    SET_AMESG_PROP(&ART_aux_var,2,12,2,2,0); /*id, size, mit, ddln, init */
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_HARD);
```

```
    SET_AMESG_PROP(&ART_aux_var,5,15,5,5,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_HARD);
    SET_AMESG_PROP(&ART_aux_var,7,10,250,250,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_HARD);
    SET_AMESG_PROP(&ART_aux_var,4,14,4,4,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_SOFT);
    SET_AMESG_PROP(&ART_aux_var,6,14,6,4,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_SOFT);
    SET_AMESG_PROP(&ART_aux_var,10,14,10,4,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_SOFT);
    SET_AMESG_PROP(&ART_aux_var,11,14,11,4,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_SOFT);
    SET_AMESG_PROP(&ART_aux_var,12,14,12,4,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_SOFT);
    SET_AMESG_PROP(&ART_aux_var,13,14,13,4,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_SOFT);
    SET_AMESG_PROP(&ART_aux_var,14,14,14,4,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_SOFT);
    SET_AMESG_PROP(&ART_aux_var,15,14,15,4,0);
    SRDB_ART_addmesg(&ART_aux_var,ADATA_MESG_ID_DATA_SHORT,TMLN_SOFT);
    cprintf(" Finish building message set (AMS1)!");

    cprintf("Any key to continue");
    keyb_getchar();

#ifdef APP_DEBUG
    /* Print the initial message set */
    /* Synchronous messages */
    cprintf("\n Message set:");
    SRDB_SRT_printmesg();
    cprintf("\n Any key to continue");
    keyb_getchar();
    /* Asynchronous messages */
    SRDB_ART_printmesg();
    cprintf("\n Any key to continue");
    keyb_getchar();
#endif

    /*****************************************/
    /* Messages configured.  Start the system */
    /*****************************************/
    ftt_mstart();
    /* Main task ends but system does not shutdown since there are active tasks */
    return 0;

}
```