

**UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

**TOLERÂNCIA A FALHAS ATRAVÉS DE  
ESCALONAMENTO EM UM SISTEMA  
MULTIPROCESSADO**

**MARCOS PÊGO DE OLIVEIRA**

**BELO HORIZONTE BRASIL**

**FEVEREIRO 2004**

**MARCOS PÊGO DE OLIVEIRA**

**TOLERÂNCIA A FALHAS ATRAVÉS DE  
ESCALONAMENTO EM UM SISTEMA  
MULTIPROCESSADO**

Tese de Doutorado submetida ao Curso de  
Ciência da Computação do Instituto de  
Ciências Exatas da Universidade Federal de  
Minas Gerais, como requerimento parcial  
para a obtenção do grau de Doutor em  
Ciência da Computação.

**Orientador : ANTÔNIO OTÁVIO FERNANDES**

**Co-Orientador: SÉRGIO VALE AGUIAR CAMPOS**

**BELO HORIZONTE BRASIL**

**FEVEREIRO 2004**




UNIVERSIDADE FEDERAL DE MINAS GERAIS


## FOLHA DE APROVAÇÃO

Tolerância a Falhas Através de Escalonamento em um Sistema Multiprocessado

**MARCOS PÊGO DE OLIVEIRA**

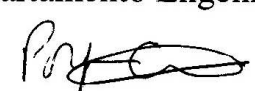
Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:


  
Prof. ANTÔNIO OTÁVIO FERNANDES - Orientador  
Departamento de Ciência da Computação - UFMG

  
Prof. SÉRGIO VALE AGUIAR CAMPOS - Co-orientador  
Departamento de Ciência da Computação - UFMG

  
Prof. JONI DA SILVA FRAGA  
Departamento de Engenharia Elétrica - UFSC

  
Prof. LUIGI CARRO  
Departamento Engenharia Elétrica - UFRGS

  
Prof. PORFÍRIO CABALEIRO CORTIZO  
Departamento de Engenharia Eletrônica - UFMG

  
Prof. CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR  
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 27 de fevereiro de 2004.

# Agradecimentos

À Adriana, Bárbara, Débora e Pâmela, minha esposa e filhas, pelo pleno apoio durante todo o desenvolvimento deste trabalho.

Aos meus pais, pela sólida educação que sempre me permitiu enfrentar os mais difíceis desafios com muita firmeza e tranquilidade.

Ao Aluísio, meu irmão, que me apoiou e conduziu a Engetron durante os momentos que dediquei ao curso de Doutorado.

Aos professores Antônio Otávio e Sérgio Campos, pela orientação, confiança em meu trabalho e constante incentivo.

Aos professores Edmund Clarke e Raj Rajkumar pela orientação durante o período de 1 ano que estive na Carnegie Mellon University como cientista visitante.

À Ana Luiza Zuquim, pelo trabalho que desenvolvemos juntos, fazendo análise crítica dos resultados, editando e corrigindo diversos textos de artigos e da tese.

À excelente equipe do Departamento de Ciência da Computação da Universidade Federal de Minas Gerais, pelo ótimo ambiente de trabalho e incontáveis contribuições para o desenvolvimento da minha linha de pesquisa.

# Acknowledgments

First and foremost I would like to thank Adriana, Bárbara, Débora e Pâmela, my wife and daughters, for the full support during the development of the whole work.

I am very grateful to my parents, for the solid education which prepared me to face challenges firmly and serene.

I would also like to thank Aluísio, my brother, for his support and for managing Engetron while I was dedicating my time to my Doctorate course.

Special thanks are due to professors Antônio Otávio and Sérgio Campos, for their guidance, enthusiasm, and trust on my work.

I wish to thank professors Edmund Clarke and Raj Rajkumar for giving me directions on my work during the one year I've been at Carnegie Mellon University as a visiting scientist.

I would like to thank Ana Luiza Zuquim, for the work we developed together and for her assistance on writing several papers and the thesis.

I would especially like to thank everyone at the Computer Science Department of the Federal University of Minas Gerais, which provided me a great workplace and uncountable contributions to the development of my research.

# Resumo

Os requisitos de tolerância a falhas devem ser incluídos no projeto inicial de sistemas de tempo real, contemplando a integração de software, hardware e restrições de tempo. Existem muitos casos em que o projeto de tolerância a falhas deve ser incluído estaticamente, devido às altas taxas de dados e severas restrições de tempo. Sistemas que exigem alta segurança necessitam de técnicas formais de prova para garantir que os requisitos de tolerância a falhas serão cumpridos. Novas abordagens onde os sistemas podem prever que as restrições de tempo serão violadas permitem que decisões sejam tomadas antes que uma pane ocorra. Priorizar tarefas em sistema de tempo real é um problema pertencente à classe NP-Hard.

Várias alternativas para resolver este problema já foram propostas. Neste trabalho, investiga-se a possibilidade de se obter um nível mais alto de tolerância à falhas com a integração de alguns trabalhos correlatos. Este trabalho apresenta técnicas para melhorar a capacidade de tolerância a falhas de sistemas de tempo real incorporando redundância de tempo, redundância de processadores e protocolos de comunicação em tempo real e tolerantes a falhas. A principal meta é garantir os requisitos de tolerância a falhas para sistemas de tempo real multiprocessados. As ferramentas utilizadas para otimizar estes requisitos de tolerância a falhas são a escalonabilidade de tarefas e a redundância de tempo.

Este trabalho, em complemento a uma abordagem teórica, foi desenvolvido utilizando uma implementação composta de um sistema multiprocessado com CPUs DSP interconectadas por um barramento CAN. A pesquisa leva em consideração o overhead introduzido pelo protocolo de comunicação de tempo real tolerante a falhas.

# Abstract

Fault tolerance must be included in the initial design of real time systems, must encompass hardware and software, and must be integrated with timing constraints. In many situations, the fault-tolerant design must be static, due to extremely high data rates and severe timing constraints. Ultrareliable systems need to employ proof-of-correctness techniques to ensure fault tolerance properties. We also see new approaches where the system predicts that timing constraints will be missed, enabling early action on such faults. Prioritizing tasks in Hard-Real-Time Systems is a problem belonging to NP-hard class.

Various alternatives for solving that problem have already been proposed. In the present study, we have investigated the possibility of obtaining a high level of fault-tolerance with the integration of some works proposals. This work presents techniques to enhance the fault-tolerance capability of hard real-time systems by incorporating time redundancy, processor redundancy and fault tolerant real-time communication protocols. The main goal is to guarantee fault-tolerance requisites for multiprocessor hard real-time systems. The resources used to optimize these fault-tolerance requisites are task schedulability and time redundancy.

This work, aside from theoretical concerns, will research the implementation of using a DSP multiprocessor system interconnected by a CAN bus. The research will consider the overhead introduced by the real-time fault tolerant communication protocol.

# Contents

<b>AGRADECIMENTOS</b> .....	<b>III</b>
<b>ACKNOWLEDGMENTS</b> .....	<b>IV</b>
<b>RESUMO</b> .....	<b>V</b>
<b>ABSTRACT</b> .....	<b>VI</b>
<b>LIST OF FIGURES</b> .....	<b>11</b>
<b>LIST OF TABLES</b> .....	<b>12</b>
<b>GLOSSARY</b> .....	<b>13</b>
<b>RESUMO EM PORTUGUÊS</b> .....	<b>15</b>
INTRODUÇÃO.....	15
TRABALHOS RELACIONADOS .....	16
<i>Teoria Rate-monotonic</i> .....	17
<i>Rate-monotonic e tolerância a falhas</i> .....	17
<i>Protocolos de comunicação de tempo real tolerantes à falhas</i> .....	18
ABORDAGEM PROPOSTA E DESENVOLVIMENTO DO TRABALHO .....	19
<i>Análise do tempo de transmissão de mensagens no barramento CAN</i> .....	21
<i>Aplicando conceitos de tolerância à falhas ao protocolo CAN</i> .....	22
<i>Limitação da taxa máxima de transferência por nó no barramento CAN</i> .....	23
COMPARAÇÃO DE PROTOCOLOS DE TEMPO REAL TOLERANTES À FALHAS .....	24
ESTUDO DE CASO – UTILIZAÇÃO DO RMCAN EM UM SISTEMA DE ALIMENTAÇÃO DE ENERGIA SEM INTERRUPÇÕES .....	25
CONCLUSÕES .....	26
<b>CHAPTER 1</b> .....	<b>28</b>
<b>INTRODUCTION</b> .....	<b>28</b>
1.1 OVERVIEW AND MOTIVATION .....	28
1.2 THESIS GOALS.....	31
1.3 RELATED WORK.....	32
1.4 CONTRIBUTIONS .....	34



1.4.1 Publications and other contributions .....	35
1.5 ORGANIZATION .....	37
<b>CHAPTER 2 .....</b>	<b>38</b>
<b>FAULT TOLERANCE IN REAL-TIME SYSTEMS.....</b>	<b>38</b>
2.1 OVERVIEW .....	38
2.2 REAL-TIME SYSTEMS.....	39
2.3 FAULT TOLERANCE .....	40
2.4 HARDWARE REDUNDANCY .....	43
2.4.1 Watchdog Timer.....	44
2.5 SOFTWARE REDUNDANCY .....	45
2.6 INFORMATION REDUNDANCY.....	46
2.7 TIME REDUNDANCY .....	47
2.8 GOALS OF FAULT TOLERANCE PROJECTS .....	48
<b>CHAPTER 3 .....</b>	<b>51</b>
<b>REAL-TIME FAULT TOLERANCE SCHEDULING .....</b>	<b>51</b>
3.1 OVERVIEW .....	51
3.2 RATE MONOTONIC THEORY .....	52
3.3 FAULT TOLERANCE AND RATE MONOTONIC SCHEDULING.....	56
3.4 MULTIPROCESSOR SYSTEMS.....	58
3.4.1 Fault tolerance in multiprocessor real-time systems.....	59
<b>CHAPTER 4 .....</b>	<b>64</b>
<b>REAL-TIME FAULT-TOLERANT COMMUNICATION PROTOCOLS .....</b>	<b>64</b>
4.1 OVERVIEW .....	64
4.2 MEDIUM ACCESS PROTOCOL CLASSES .....	65
4.3 MODEL FOR HARD REAL-TIME COMMUNICATION .....	66
4.4 TIME-TRIGGERED PROTOCOL TTP/C .....	68
4.5 FLEXRAY PROTOCOL .....	70
4.6 CONTROLLER AREA NETWORK PROTOCOL CAN .....	71
4.7 TTCAN - TIME TRIGGERED COMMUNICATION ON CAN .....	73
<b>CHAPTER 5 .....</b>	<b>74</b>
<b>THE THESIS GOALS AND APPROACHES.....</b>	<b>74</b>

5.1 OVERVIEW .....	74
5.2 SUMMARY OF GOALS .....	74
5.3 GOALS AND APPROACHES .....	75
5.3.1 <i>Enhance the fault tolerance capabilities of multiprocessor hard real-time systems</i> .....	75
5.3.2 <i>Guarantees required for fault-tolerant execution in real-time systems</i> .....	76
5.3.3 <i>Study the tradeoff between fault tolerance capability and resource utilization of hard real-time systems</i> .....	77
5.3.4 <i>Use techniques to improve system utilization</i> .....	78
5.3.5 <i>Define resiliency of the systems</i> .....	79
5.3.6 <i>Estimate the overhead introduced by the real-time fault-tolerant communication protocol</i> .....	79
5.4 SUMMARY .....	81
<b>CHAPTER 6 .....</b>	<b>83</b>
<b>IMPROVING FAULT TOLERANCE IN HRTSS .....</b>	<b>83</b>
6.1 OVERVIEW .....	83
6.2 SYSTEM, TASK AND FAULT MODELS.....	85
6.3 FAULT TOLERANCE AND RATE MONOTONIC SCHEDULING IN MULTIPROCESSOR SYSTEMS .....	86
6.4 RATE MONOTONIC SCHEDULING APPLIED TO THE CAN BUS .....	87
6.5 IMPROVING RELIABILITY IN A CAN BUS .....	90
6.5.1 <i>Applying Fault tolerance requisites to a CAN bus</i> .....	90
6.5.2 <i>Limiting the maximum transfer rate of the CAN bus</i> .....	92
6.6 FORMAL VERIFICATION OF RMCAN.....	94
6.6.1 <i>Modeling Premises</i> .....	94
6.6.2 <i>Formal verification of the capability to guarantee message delivery and of the recovery and fault tolerance capability of the protocol</i> .....	94
6.7 COMPARISON OF REAL-TIME FAULT-TOLERANT COMMUNICATION PROTOCOLS – ADVANTAGES OF THE PROPOSED METHOD .....	97
6.8 TARGET APPLICATION: DESIGN OF AN UNINTERRUPTIBLE POWER SUPPLY .....	99
6.8.1 <i>UPS Functionality</i> .....	99
6.8.2 <i>UPS Architecture and Operating Modes</i> .....	100

6.8.3 <i>Fault tolerant support</i> .....	101
6.8.3.1 <i>Tests and Results</i> .....	103
6.8.4 <i>UPSs' parallelism</i> .....	103
6.9 SUMMARY .....	107
<b>CHAPTER 7 .....</b>	<b>109</b>
<b>CONCLUSIONS.....</b>	<b>109</b>
7.1 SUMMARY .....	109
7.2 FUTURE WORK.....	111
<b>APPENDIX A .....</b>	<b>113</b>
UPS ARCHITECTURE AND OPERATION MODES .....	113
<b>REFERENCES .....</b>	<b>116</b>

# List of Figures

FIGURA A. DEFINIÇÃO DO MODELO DE FALHAS.....	18
FIGURA B. BARRAMENTO CAN EM UMA PLANTA DE CONTROLE .....	20
FIGURA C. VISÃO GERAL DA ARQUITETURA .....	24
FIGURA D. OPERAÇÃO DE UM NO-BREAK UTILIZANDO RMCAN.....	25
FIGURE 1.1: CAN SALES EVOLUTION .....	31
FIGURE 2.1: ATTRIBUTES OF FAULT CHARACTERISTICS .....	41
FIGURE 2.2: INFORMATION REDUNDANCY .....	46
FIGURE 2.3: A TOP-LEVEL VIEW OF THE SYSTEM DESIGN PROCESS.....	50
FIGURE 3.1: PROCESSOR FAULT AND RECOVERY IN THE DEADLINE.....	61
FIGURE 3.2: FAULT-FREE SYSTEM STATUS .....	63
FIGURE 3.3: FAULT RECOVERY FROM $P_1$ EXECUTED BY $P_2$ .....	63
FIGURE 3.4: PROCESSOR $P_1$ REPLACED BY $P_s$ .....	63
FIGURE 3.5: DEADLINE RECOVERS $P_1$ .....	63
FIGURE 4.1: SHARED BROADCAST BUS .....	67
FIGURE 4.2: TTP/C ARCHITECTURE.....	70
FIGURE 4.3: CAN DATA FRAME .....	72
FIGURE 5.1: SCHEDULING PROBLEMS .....	76
FIGURE 5.2: FAULT CLASSIFICATION.....	77
FIGURE 5.3 OVERLOADING TECHNIQUES .....	78
FIGURE 5.4: HARDWARE IMPLEMENTATION .....	80
FIGURE 5.5: TMS320LF2407 BLOCK DIAGRAM.....	81
FIGURE 6.1: FAULT MODEL DEFINITION .....	84
FIGURE 6.2: CAN BUS IN A CONTROL PLANT.....	85
FIGURE 6.3: RMCAN REPRESENTATION .....	95
FIGURE 6.4: PROPERTIES CHECKED FOR THE RECOVERY AND FAULT TOLERANCE CAPABILITY OF THE PROTOCOL.....	97
FIGURE 6.5: UPS FUNCTIONALITY – INPUT AND OUTPUT VOLTAGES.....	100
FIGURE 6.6: UPS ARCHITECTURE - DATA SYNCH THROUGH A CAN BUS .....	101
FIGURE 6.7: UPS ARCHITECTURE – OPERATION IN A POWER FAULT SCENARIO.....	101

FIGURE 6.8: UPS ARCHITECTURE – OCCURRENCE OF A CAN FAULT IN A POWER FAULT SCENARIO .....	102
FIGURE 6.9: DETERMINING THE REFERENCE CURRENT OF THE SLAVE UPSs .....	105
FIGURE 6.10: OUTPUT VOLTAGE.....	106
FIGURE 6.11: DISTRIBUTION OF THE CURRENT ON THE INVERTERS .....	106
FIGURE 6.12: DISTRIBUTION OF THE ACTIVE POWERS .....	106
FIGURE 6.13: DISTRIBUTION OF THE REACTIVE POWERS .....	106
FIGURE 6.14: TOTAL CURRENT IN THE LOAD DURING A FAILURE IN THE UPS MASTER.....	107
FIGURE 6.15: VOLTAGE IN THE LOAD DURING A FAILURE IN THE UPS MASTER .....	107
FIGURE A1: UPS ARCHITECTURE CONTROL DETAILED VIEW .....	113

## List of Tables

TABLE 1.1: 1999 WORLD MARKET FOR MICROPROCESSORS [16].....	28
TABLE 6.1: COMPARISON BETWEEN TTP/C, CAN, TTCAN, RMCAN AND FLEXRAY PROTOCOLS.....	99
TABLE 6.2: RESULTS OBTAINED FROM AN 80KVA UPS .....	103

# Glossary

Term	First appears on page	Notes
ALU	27	Arithmetic Logical Unit
B	72	Blocking time
$C_j$	72	Worst-case time taken to physically transmit the message on the bus.
$C_m$	72	Transmission Delay
CAN	14	Controller Area Network
CPU	12	Central Processing Unit
CRC	28	Cyclic Redundancy Check
CSMA/CA	46	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	46	Carrier Sense Multiple Access with Collision Detection
CTT	17	Completion Time Test
$D_m$	72	Deadline of message $m$ .
DMA	49	Direct Memory Access
DSP	12	Digital Signal Processor
$E(R_m)$	73	Probable bound on the error recovery overheads before a message $m$ arrives at its destination
FIFO queue	49	First In First Out queue
FlexRay	14	FlexRay Protocol
FTRMFF	17	Fault-Tolerant Rate-Monotonic First-Fit
$h_p(m)$	72	Set of all messages in the system of higher priority than message $m$ .
HRTSs	14	Hard Real-Time Systems
IMAGES	16	Integrated Modeling for Analysis and Generation of Embedded Software
I/O	32	Input/Output
$J_j$	72	Jitter on the queuing of message $m$
$m$	72	A Message
MOBIES	16	Model-Based Integration of Embedded Systems
NMI	26	Non-Maskable Interrupt
$R_m$	72	Worst-case response time
RAM	21	Random Access Memory
RM	34	Rate-Monotonic
RMA	17	Rate-Monotonic Algorithm
RMCAN	21	Rate-Monotonic Controller Area Network
RMFF	17	Rate-Monotonic First Fit heuristic
RMS	33	Rate-Monotonic Scheduling
ROM	27	Read-Only Memory
SRTSs	13	Soft Real-Time Systems
$\tau_{\text{bit}}$	72	Time taken to transmit a bit on CAN

$T_j$	72	The period of message m
$t_m$	72	Longest time a message can be queued in a station an be delayed because other messages are being sent on the bus
TDMA	18	Time Division Multiple Access protocol
TTA	49	Time-Triggered Architecture
TTCAN	14	Time-Triggered Controller Area Network
TTP/A	14	Time-Triggered Protocol for class A
TTP/C	14	Time-Triggered Protocol for class C

# Resumo em Português

## Introdução

Sistemas de tempo real são sistemas cuja correta execução depende não só do resultado correto da computação, mas também do instante de tempo em que este resultado é alcançado. Exemplos de sistemas de tempo real incluem os processadores de sinais, controladores de vãos e de processos, aplicações em telecomunicações, sistemas automotivos e sistemas médicos de suporte à vida. Sistemas de tempo real críticos (HRTSs), mais especificamente, possuem restrições de tempo rígidas, e o não cumprimento dos prazos das tarefas pode ser catastrófico. Estes sistemas têm como premissas a confiabilidade, disponibilidade, segurança, desempenho, entre outras características. Falhas em tais sistemas podem ocasionar perdas humanas, ecológicas e econômicas.

Escalonamento e alocação de recursos em sistemas de tempo real são problemas difíceis em função das restrições de tempo das tarefas envolvidas. Tolerância a falhas é um requisito vital no desenvolvimento de sistemas de tempo real críticos. Políticas de escalonamento nestes sistemas devem garantir que as tarefas cumprirão seus prazos finais sob quaisquer circunstâncias, mesmo na presença de falhas transientes e permanentes. Uma característica de sistemas de tempo real está no fato de que estes sistemas devem ser capazes de prever que um prazo não será atendido e tomar as ações necessárias antes que uma pane ocorra. Os requisitos de tempo e o modelo de falhas dependem do conhecimento preciso da aplicações e do ambiente na qual esta está inserida.

A base de todas as técnicas de tolerância a falhas está relacionada ao conceito de redundância. Este conceito era implementado, inicialmente, através da simples replicação de componentes. Ao longo do tempo notou-se que outros tipos de redundância levam ao mesmo resultado e, muitas vezes, de forma mais eficiente.

Neste trabalho, uma análise de técnicas de tolerância a falhas no contexto de sistemas de tempo real multiprocessados é apresentada, incluindo escalonamento e protocolos de comunicação. Uma nova técnica baseada em escalonabilidade de tarefas e incorporando redundância de tempo foi proposta, podendo esta ser utilizada em conjunto com redundâncias de hardware e software. O trabalho foi desenvolvido sobre o protocolo CAN (Controller Area



Network) por se tratar de um protocolo que está presente em mais de 90% das aplicações envolvendo microcontroladores e DSPs que incorporam protocolos de tempo real. Um problema conhecido do barramento CAN está na entrega de mensagens de baixa prioridade, a qual pode ser comprometida caso mensagens de mais alta prioridade ocupem todo o *bandwidth*. Foi proposta assim uma extensão ao protocolo CAN através da aplicação de técnicas que buscam melhorar a confiabilidade do protocolo.

A extensão proposta, denominada RMCAN, provê garantias de falhas no barramento CAN serão toleradas. Para tanto, é definida uma taxa máxima de transmissão para cada nó, ao invés de utilizarmos *slots* de tempo pré-definidos (como é feito no TTP/C). Com esta condição de contorno, as transmissões de um nó não ficam restritas a um *slot* de tempo, mas podem ocorrer até que a taxa máxima de transmissão seja atingida.

Outra importante contribuição deste trabalho está na definição de uma arquitetura tolerante à falhas de tempo real baseada em DSPs, onde um conjunto de processadores executando em paralelo troca informações através de um barramento CAN.

## Trabalhos relacionados

Um algoritmo de escalonamento consiste em um conjunto de regras que determinam a tarefa a ser executada em um momento em particular. A abordagem tradicionalmente adotada é aquela onde o escalonamento é preemptivo e baseado em prioridades. Nesta abordagem, as tarefas possuem prioridades que lhes são atribuídas dinâmica ou estaticamente. Em um determinado momento, se uma tarefa de mais baixa prioridade está sendo executada e uma tarefa de maior prioridade entra na fila, a tarefa de menor prioridade é colocada em espera enquanto o processador é liberado para a execução da tarefa de maior prioridade. Desta forma, a especificação de algoritmos de escalonamento preemptivos baseados em prioridades está diretamente relacionado à especificação de algoritmos de atribuição de prioridades. Um algoritmo é dito estático se as prioridades são atribuídas uma única vez; da mesma forma, um algoritmo é dito dinâmico se as prioridades são alteradas entre uma requisição e outra. Existem ainda algoritmos mixtos onde algumas tarefas têm suas prioridades definidas estaticamente enquanto outras têm suas prioridades definidas dinamicamente. Caso tenhamos as prioridades definidas sistematicamente, utilizando a abordagem *rate-monotonic*, por exemplo, limites de utilização podem ser determinados. Nesta situação, se um conjunto de

tarefas não exceder os limites, podemos dizer que as tarefas poderão ser executadas sem que os prazos finais sejam perdidos.

## **Teoria *Rate-monotonic***

O termo “análise *rate-monotonic*” surgiu em 1973 através da publicação de um artigo denominado "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment" por Liu e Layland [5]. Este artigo propôs o que seria a base para um teste simples a ser aplicado em sistemas de tempo real para se determinar se um conjunto de tarefas seria executado antes de seus prazos finais. Foram considerados prazos finais das tarefas o final de seus períodos, e nenhuma tarefa poderia bloquear o sistema enquanto tivesse sendo executada. Além disto, foram atribuídas única e monotonicamente prioridades a cada tarefa do conjunto.

O algoritmo *rate-monotonic* (RMA) consiste em um algoritmo de escalonamento de tarefas que atribui mais altas prioridades a tarefas com menor período; é ótimo se o conjunto de tarefas for independente [5]. Assume-se que todas as tarefas no sistema são periódicas, têm seus prazos finais no final de seus períodos e são totalmente independentes umas das outras[51].

Vários trabalhos foram desenvolvidos baseados na teoria RMA: Joseph and Pandya[10] demonstraram o Completion Time Test (CTT) para verificar a escalonabilidade de um conjunto fixo de tarefas periódicas em um processador. Bertossi e Mancine [15] propuseram o Fault-Tolerant Rate-Monotonic First-Fit (FTRMFF) para prover tolerância a falhas a um HRTS utilizando um sistema multiprocessado onde cada tarefa tem uma cópia ativa ou passiva em outro processador e todo o conjunto de tarefas é escalonado pelo RMFF, suportando falhas *fail-stop* de um ou mais processadores.

### ***Rate-monotonic* e tolerância a falhas**

Em função da natureza crítica das tarefas em sistemas de tempo real, é essencial que falhas sejam toleradas. Uma falha em um sistema pode se manifestar de várias formas, tornando seu diagnóstico muitas vezes complicado. Buscando reduzir este problema, um sistema pode ser projetado de forma a seguir um modelo de falhas, tornando o problema de diagnosticá-las simplificado. Nem todas as falhas que ocorrem em um sistema são tratadas

pelo modelo de falhas, sendo que existem modelos de falhas que cobrem um grande percentual das falhas possíveis. Um modelo genérico é apresentado na figura abaixo:

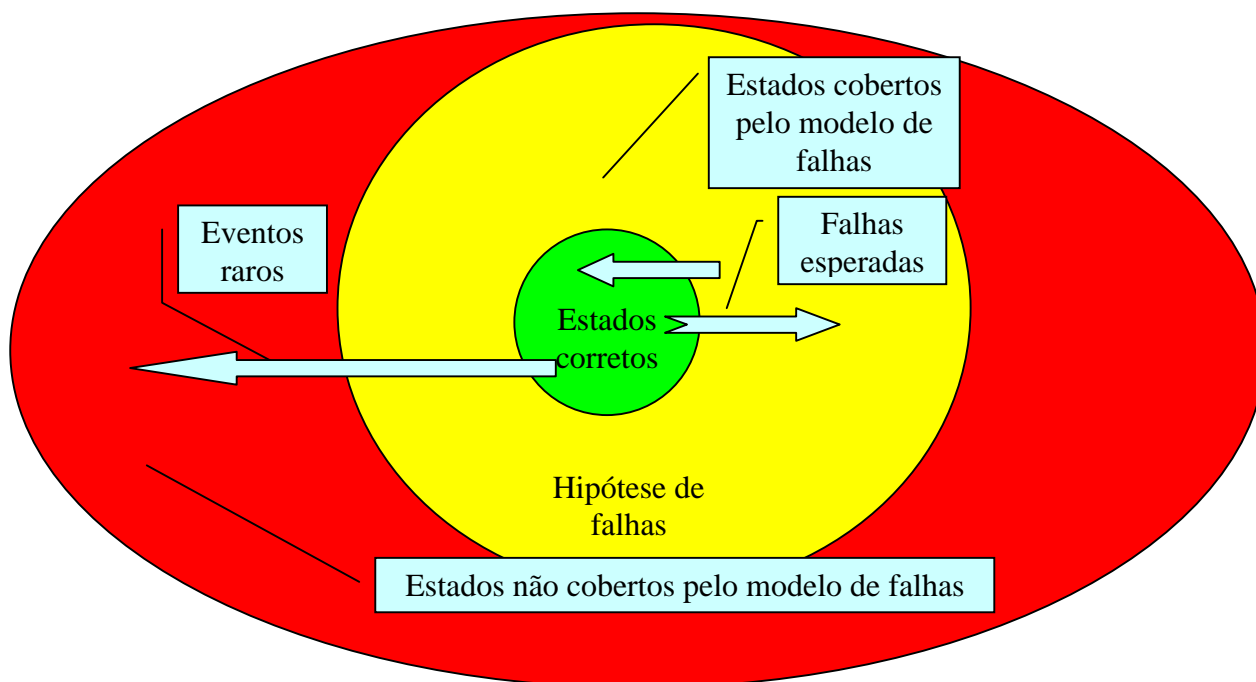


FIGURA A. DEFINIÇÃO DO MODELO DE FALHAS

Falhas transitientes são toleradas, geralmente, utilizando-se redundância de tempo, a qual envolve a re-execução das tarefas que estavam executando durante a falha transitiente.

Ghosh propôs em [48] um algoritmo onde a execução de tarefas de tempo real é garantida mesmo na ocorrência de falhas transitientes e intermitentes. A abordagem geral para se tolerar falhas consiste em reservar tempo suficiente no escalonamento de forma que qualquer instância de uma tarefa possa ser re-executada caso uma falha ocorra durante sua execução. Se nenhuma falha ocorrer, as tarefas serão executadas seguindo o esquema usual do *rate-monotonic*.

Ghosh propôs ainda um algoritmo para escalonamento de tarefas tolerante a falhas em sistemas multiprocessados. O algoritmo garante a execução de uma tarefa antes do prazo final mesmo na presença de falhas no processador.

## Protocolos de comunicação de tempo real tolerantes à falhas

Um serviço essencial provido por arquiteturas distribuídas de tempo real tolerantes à falhas é a troca de informação entre os componentes distribuídos. Estas arquiteturas têm como importante componente o barramento de comunicação e protocolos utilizados no

controle e comunicação estão entre seus principais mecanismos. Em sistemas distribuídos de tempo real, a comunicação entre diferentes processadores deve ocorrer em um tempo pré-determinado.

Barramentos redundantes são utilizados com frequência em ambientes críticos para lidar com as falhas nos dispositivos. Existem vários protocolos de comunicação utilizados em sistemas de tempo real, cada qual apresentando uma complexidade. Exemplos mais representativos são o protocolo TTP/C (Time-Triggered Protocol) [20] FlexRay [10], CAN (Controller Area Network) [8] and TTCAN (Time-Triggered CAN) [5][7][15]. Estes protocolos têm sido utilizados na indústria da aviação (Airbus e Boeing), na indústria automotiva (BMW e a Audi), entre outras.

Alguns protocolos citados acima são basicamente “time-triggered”, como por exemplo o TTP/C. Isto significa que todas as atividades envolvendo o barramento e os componentes anexados a ele são baseadas na passagem do tempo. Outros protocolos são basicamente “event-triggered”, como por exemplo o CAN; nestes protocolos as atividades envolvendo o barramento são baseadas na ocorrência de eventos, respondendo a estímulos externos e interagindo com o ambiente. Existem ainda protocolos mistos, como o TTCAN e o FlexRay, onde os dois conceitos se misturam.

## **Abordagem proposta e desenvolvimento do trabalho**

Sempre que um sistema tolerante à falhas é projetado, uma forma de redundância deve ser incorporada. Escalonabilidade de tarefas e redundância de tempo são ferramentas importantes para se garantir que os requisitos de tolerância a falhas sejam atendidos para um determinado sistema. Com o custo de processadores reduzindo gradativamente e o desenvolvimento de inúmeros protocolos de comunicação de tempo real, tais como CAN [29], TTP/C [26], TTP/A [25] e FlexRay [27], tornou-se importante pesquisar aspectos relacionados a sistemas de tempo real multiprocessados, onde testes de escalonabilidade devem garantir a execução de todas as tarefas do conjunto antes de seus prazos finais e considerando um modelo de falhas.

O principal objetivo deste trabalho é garantir requisitos de tolerância a falhas em sistemas multiprocessados de tempo real crítico. Os recursos utilizados para otimizar estes requisitos são a escalonabilidade de tarefas e redundância de tempo. Este trabalho apresenta

técnicas para se melhorar a capacidade de tolerar falhas de tais sistemas incorporando ainda redundância de processadores e protocolos de comunicação de tempo real.

A abordagem utilizada consiste em prover tolerância à falhas através do escalonamento de tarefas e da adição de *slacks* de tempo no processamento. Se uma falha for detectada pelo sistema operacional durante a execução de uma tarefa, esta tarefa deverá ser re-executada dentro deste *slack* de tempo ou uma tarefa backup deve ser ativada para recuperação da falha.

A implementação de um sistema multiprocessado baseado em DSPs interconectado por um barramento CAN é apresentada.

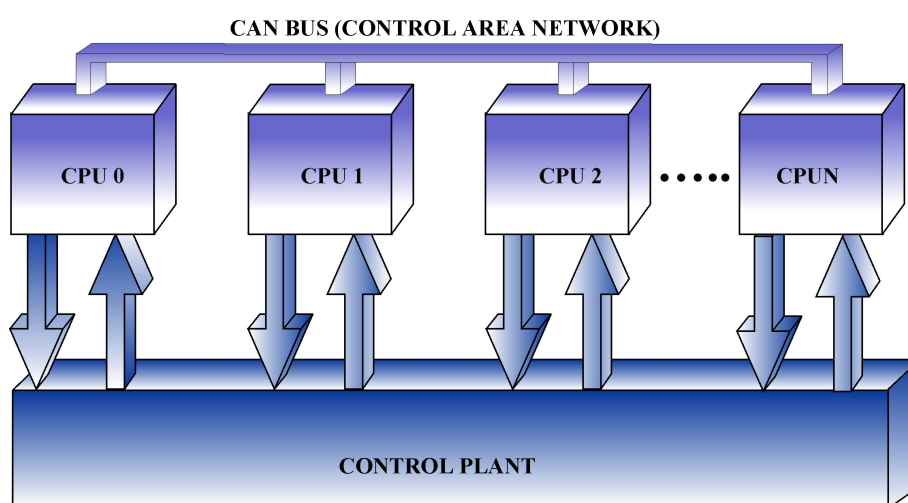


FIGURA B. BARRAMENTO CAN EM UMA PLANTA DE CONTROLE

Na figura acima, cada processador executa aplicações de tempo real periódicas escalonadas através do RMA com tempo máximo de execução e período de execução pré-definidos. Por se tratarem de tarefas de controle, consistem em um conjunto de tarefas independentes, onde o início de uma tarefa não depende da execução de nenhuma outra tarefa.

O modelo de falhas foi definido considerando as seguintes características do sistema:

(F1) Falhas transientes e permanentes podem ocorrer na execução de uma tarefa ou até mesmo em um processador;

(F2) Apenas falhas transientes do barramento de comunicação são toleradas;

(F3) Processadores livre de falhas podem comunicar entre si;

(F4) O hardware provê isolamento das falhas no sentido de que uma falha em um processador não provocará falhas em outros processadores;

(F5) A falha de um processador é detectada pelos demais através da não execução de alguma tarefa alocada para o processador que falhou, e é sinalizada pela ausência de mensagem no barramento CAN.

O modelo inclui falhas que podem ocorrer no barramento ou no protocolo de comunicação, uma vez que estes influenciam o funcionamento do sistema como um todo. Considerando o barramento CAN, é necessário garantirmos que um pacote será entregue dentro do seu prazo final ou, no pior caso, que o remetente saberá que o envio foi sem sucesso e que nova transmissão deverá ser feita. Uma mensagem deve ser retransmitida um número de vezes no caso de falha na transmissão e ainda assim cumprir seu prazo final. No barramento CAN, uma falha é sinalizada através da ausência de mensagens, isto é, se uma mensagem não for recebida pelos processadores dentro de um intervalo de tempo, uma falha no processador primário é assumida e a recuperação desta falha é feita através de uma tarefa backup. Quaisquer erros nas mensagens podem indicar que uma falha ocorreu, e uma tarefa alternativa será executada consequentemente, evitando assim uma pane no sistema. Falhas permanentes no barramento CAN não são toleradas, causando uma interrupção no processo de comunicação.

## **Análise do tempo de transmissão de mensagens no barramento CAN**

O algoritmo de escalonamento dinâmico utilizado pelo protocolo CAN é praticamente idêntico aos algoritmos de escalonamento comumente utilizados em sistemas de tempo real para escalonar tarefas em processadores [18]. A análise do comportamento destes sistemas pode ser aplicada quase que sem modificações na solução do problema de se determinar o pior caso do tempo de transmissão de uma dada mensagem no barramento CAN.

Tindell *et. al.* [19] desenvolveu uma análise sobre o barramento CAN baseada na análise *rate-monotonic*, mostrando como calcular o pior tempo de resposta para mensagens transmitidas pelo barramento.

A ocorrência de erros de transmissão também deve ser considerada. Em um barramento CAN, um erro detectado tanto pelo emissor quanto pelo receptor da mensagem é sinalizado ao emissor, o qual deve retransmitir a mensagem.

Um problema conhecido do protocolo CAN é o fato de não se poder garantir que mensagens de mais baixa prioridades serão entregues antes do seu prazo final em caso de sobrecarga. Enquanto o protocolo CAN é muito eficiente na transmissão de dados mais urgentes, estas mensagens podem sobrecarregar o barramento de tal forma que as mensagens de mais baixa prioridade não conseguirão cumprir seus prazos de transmissão [11][12][13].

Neste sentido, é necessário garantirmos que a sobrecarga não ocorrerá, para que todos os prazos finais sejam cumpridos.

## **Aplicando conceitos de tolerância à falhas ao protocolo CAN**

Falhas no barramento CAN implicam na re-transmissão de mensagens ou até mesmo na execução de ações alternativas para a reconfiguração do barramento.

Ghosh [6] desenvolveu um esquema para recuperação de uma ou mais falhas que garante a re-execução de qualquer tarefa assim que uma falha for detectada. Os mesmos conceitos apresentados por Ghosh foram aplicados ao protocolo CAN, buscando garantir que uma mensagem seja re-transmitida quando uma falha na transmissão for detectada. Estes conceitos, utilizados em escalonamento *rate-monotonic*, foram aplicados ao barramento CAN uma vez que:

[S1] As mensagens são independentes ou assíncronas.

[S2] As mensagens têm suas prioridades definidas pelo RMA.

[S3] Uma mensagem de mais alta prioridade não deve ocupar todo o *bandwidth* do barramento impedindo que mensagens de mais baixa prioridade executem

Segundo a abordagem apresentada por Ghosh, deve ser mantido um intervalo de tempo suficiente para que uma mensagem seja re-transmitida. Caso nenhuma falha ocorra, as mensagens são transmitidas normalmente, seguindo o escalonamento *rate-monotonic* e este *slack* não é utilizado. Caso ocorra algum erro no processo de transmissão da mensagem, esta deve ser re-transmitida antes do fim do seu período de transmissão. Para que os conceitos apresentados por Ghosh pudessem ser aplicados ao protocolo CAN, as seguintes condições devem ser satisfeitas:

[S1] Deve ser reservado um intervalo de tempo suficiente para que cada instância de cada mensagem possa ser re-transmitida.

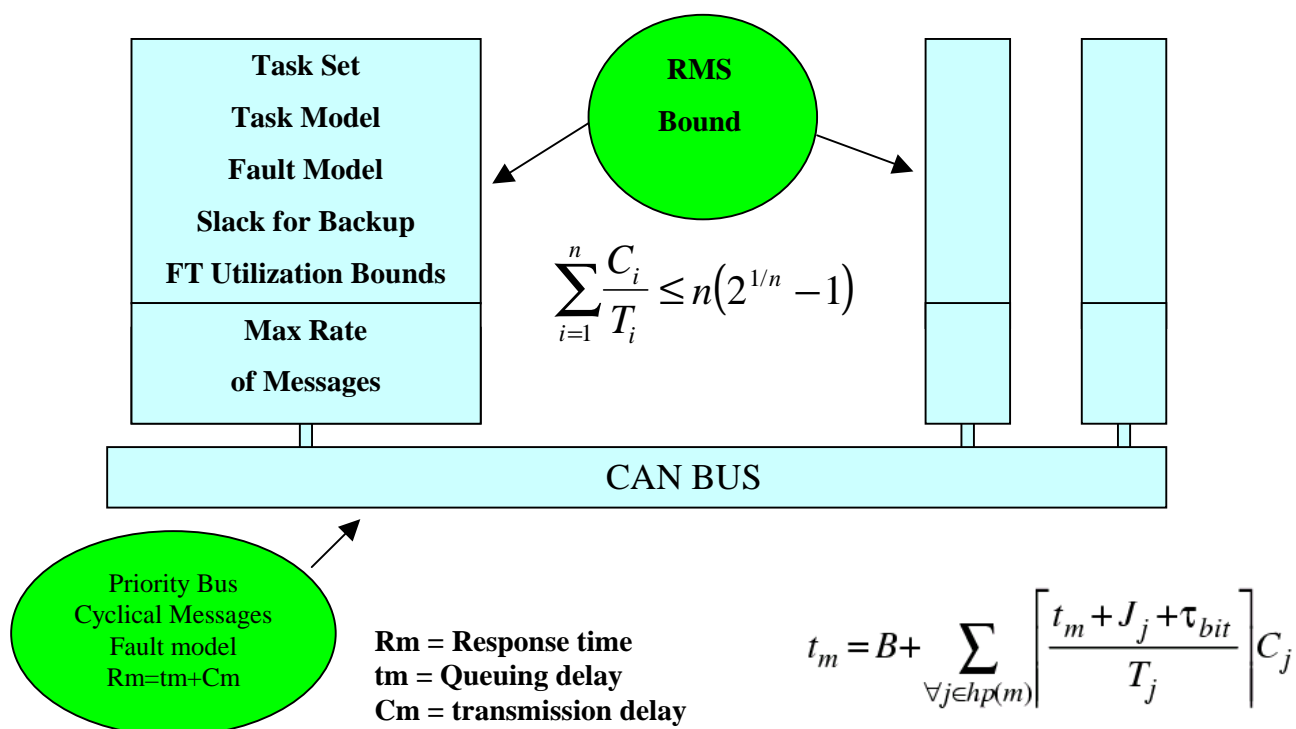
[S2] Quando uma instância de uma mensagem é transmitida, uma quantidade suficiente do *slack* disponível dentro do seu período deve estar disponível para que esta mensagem possa ser re-transmitida antes do prazo final, caso uma falha seja detectada.

[S3] Quando uma mensagem é re-transmitida, esta não deve interferir na transmissão de nenhuma outra mensagem, ou seja, a re-transmissão de uma mensagem de mais alta prioridade não interferirá no cumprimento dos prazos finais de outras mensagens.

## Limitação da taxa máxima de transferência por nó no barramento CAN

Considerando os resultados obtidos por Tindell em [69], onde mostrou-se ser possível determinar o tempo máximo de transmissão para o barramento CAN no pior caso, foi demonstrado no Capítulo 6 que as três premissas podem ser atendidas.

Foi proposta assim uma extensão ao protocolo CAN, denominada **Rate Monotonic CAN (RMCAN)**, onde foram definidos slacks de tempo de tamanho suficiente para que, mesmo no pior caso, todas as mensagens possam ser transmitidas ou até mesmo re-transmitidas se necessário. Cada mensagem tem sua transmissão limitada a um tempo máximo e um período pré-definido, sendo definido um limite para a taxa máxima de transmissão para um nó do barramento CAN. Podemos assim garantir que é possível re-transmitir uma mensagem que apresentou falha e ainda cumprir o prazo final. Desta forma, ao invés de limitar a transmissão a um período pré-definido, como é feito pelo TTP/C, um nó pode distribuir sua transmissão durante vários intervalos de tempo até que seu limite seja atingido.





## FIGURA C. VISÃO GERAL DA ARQUITETURA

A independência das mensagens é garantida pelo fato de se tratarem de mensagens geradas por tarefas de controle que são independentes. Uma visão geral da arquitetura, considerando todos os conceitos apresentados, é mostrada na figura abaixo.

## **Comparação de protocolos de tempo real tolerantes à falhas**

Podemos dizer que o protocolo TTP/C provê o projeto estático assegurando um tempo máximo de transmissão para todas as mensagens, mas ao mesmo tempo ele apresenta baixa flexibilidade uma vez que a largura de banda é distribuída em tempo de projeto através da atribuição de frames de tamanho específico a cada nó. Em um barramento CAN, por outro lado, as prioridades podem ser determinadas em tempo de execução através da atribuição de identificadores únicos e um controle completo da aplicação sobre a distribuição da largura de banda disponível. O protocolo CAN é um protocolo altamente flexível e largamente disponível, apesar de que algumas extensões devem ser feitas para se garantir um mecanismo confiável para se construir sistemas tolerante à falhas confiáveis. TTCAN é um compromisso e representa a necessidade de evolução do CAN, para que sobrecargas no barramento possam ser tratadas corretamente sem prejudicar a transmissão de mensagens. Por outro lado, a sincronização de nós exigida pelo TTCAN não é uma tarefa simples, exigindo hardware adicional. RMCAN foi desenvolvido com o objetivo de se obter a eficiência do TTCAN, mas sem incorporar nenhum hardware adicional ou outras restrições. Tanto o TTCAN quanto o RMCAN podem ser implementados utilizando controladores CAN. No caso do RMCAN, o controle da taxa de transmissão é feito por software, simplificando a implementação do protocolo. O protocolo FlexRay pode ser considerado o estado da arte em se tratando de protocolos de tempo real tolerantes a falhas, apesar de ainda não ter sido lançado no mercado. É um protocolo que promete uma taxa de transmissão maior do que o TTCAN e maior flexibilidade se comparado ao TTP/C.

## Estudo de caso – Utilização do RMCAN em um sistema de alimentação de energia sem interrupções

Um no-break é um exemplo de sistema de tempo real crítico onde mecanismos de tolerância à falhas são essenciais. A função primária de um no-break é garantir continuidade de operação especialmente durante falhas de energia ou perturbações no fornecimento de energia elétrica.

Uma aplicação onde o protocolo RMCAN foi utilizado consiste em um no-break trifásico de 80kVA, o qual deve fornecer energia sem interrupções às suas cargas mesmo quando uma falha no fornecimento de energia pela companhia de energia elétrica ocorrer.

A operação do no-break é controlada por lógica microprocessada: todas as tarefas são executadas por três CPUs independentes (retificador, inversor e chave estática), cada qual consistindo de um DSP. A escolha de DSPs para esta aplicação foi feita em função de características da própria aplicação. Um número maior de CPUs poderia ser utilizado considerando, por exemplo, redundância de hardware.

Apesar das CPUs operarem de forma autônoma, estas trocam informações entre si através de uma interface CAN, permitindo assim a monitoração e controle das informações. O protocolo RMCAN foi utilizado internamente para comunicação entre os processadores. Falhas de leitura de parâmetros em um dos processadores são repassadas aos outros, que tomam as ações devidas.

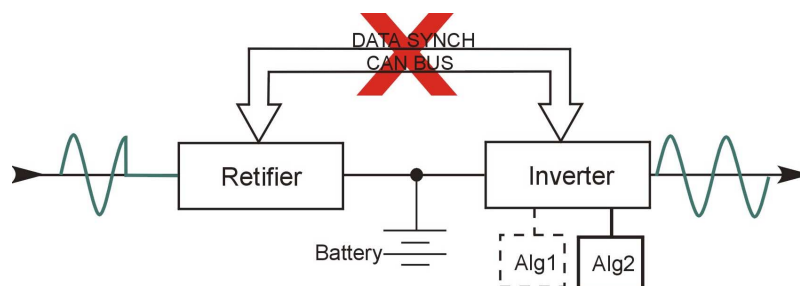


FIGURA D. OPERAÇÃO DE UM NO-BREAK UTILIZANDO RMCAN

A figura acima representa o processo de tratamento de uma falha no barramento CAN. Como as CPUs do retificador e inversor trocam informações periodicamente, o não recebimento de uma mensagem indica uma falha no barramento. Caso a re-execução indique que a falha permanece, é executado um algoritmo alternativo de forma que seja possível tolerar determinada falha. Falhas permanentes no barramento não são toleradas.

A colocação de duas ou mais unidades de fontes de alimentação em paralelo fornece ao sistema uma maior tolerância a falhas aumentando substancialmente a confiabilidade. Entretanto, como os módulos de potência não são idênticos, as correntes fornecidas por cada módulo são diferentes. Alguns dos fatores que contribuem para este desequilíbrio são a tolerância dos componentes e a impedância dos circuitos de distribuição da corrente de saída de cada módulo. O circuito de controle das fontes de alimentação deve ser capaz de regular a tensão de saída das fontes de alimentação e simultaneamente distribuir uniformemente a corrente de carga entre os diversos módulo de potência.

Uma aplicação do método de paralelismo de conversores utilizando a técnica de controle do tipo mestre-escravo com comunicação através de interface CAN foi simulada, mostrando que é possível utilizarmos o RMCAN como protocolo de comunicação entre dois no-breaks em paralelo que utilizam o barramento CAN para sincronização. Foram simuladas situações onde ocorriam falhas no no-break mestre e no escravo, bem com falhas no barramento CAN.

## Conclusões

Tarefas em sistemas de tempo real devem cumprir seus prazos sob quaisquer circunstâncias, mesmo na presença de falhas transientes e permanentes. Neste trabalho utilizou-se conceitos como redundância de tempo e escalonamento de tarefas ao lidarmos com falhas em sistemas de tempo real.

Foram analisados ainda protocolos de comunicação de tempo real, sendo considerado o protocolo CAN por estar disponível em mais de 90% das aplicações envolvendo protocolos de tempo real. O protocolo CAN possui, muitas vezes, tempo de resposta não-determinístico para mensagens de baixa prioridade. O protocolo RMCAN foi proposto neste trabalho como uma extensão ao protocolo CAN, buscando assim resolver os problemas de não determinismo do protocolo bem com garantir que todas as mensagens serão entregues dentro de seus prazos finais ou até mesmo que não serão entregues, caso alguma falha ocorra. Limitou-se a taxa de transmissão de cada nó no barramento CAN tornando assim determinístico o tempo de transmissão de uma mensagem, mesmo para mensagens de baixa prioridade. Vale ressaltar que a transmissão de mensagens por um nó não ficou restrita a um intervalo de tempo, e sim a uma taxa máxima de transmissão. É possível assim garantir que haverá tempo suficiente para que uma mensagem seja transmitida, ou até mesmo re-transmitida, se necessário. Caso uma

mensagem não chegue ao seu destino, o destinatário saberá que uma falha ocorreu e que uma ação será executada de acordo com o modelo de falhas definido para a aplicação, como por exemplo, reconfiguração do barramento CAN.

As extensões ao protocolo CAN propostas aumentam a confiabilidade do protocolo bem como permitem que seja explorada sua grande presença no mercado. Foi feita ainda a verificação formal do RMCAN, garantindo assim que a transmissão das mensagens ocorrerá dentro do seu período, mesmo considerando que falhas ocorrerão no sistema.

# Chapter 1

## Introduction

### 1.1 Overview and Motivation

Real-time systems are different from general purpose computing systems in several aspects. The processes in a real-time system have time related attributes such as ready times, deadlines, computation times and periods. A real-time system must provide predictable response times. Therefore, the worst case behavior of real-time systems is more important than the average response time. Real-time systems are systems that depend on the result of computation as well on the deadline by which this result is reached.

Architectures that support real-time applications tend to be specialized. However, the current trend is to develop more generic real-time architectures. The market trend itself, which demands an ever-increasing participation of microcontrollers and DSPs (digital signal processors), attests to the need for more generic real-time architectures. In 1992, the world market for conventional CPUs (Central Processing Units) was estimated at US\$ 4.9 billion while the market for of micro controllers and DSPs was US\$ 5.4 billion [20].

Table 1.1 shows the results of microcontrollers sales in 1999, when the number of microcontrollers completely outnumbered the desktop chips in terms of units shipped.

TABLE 1.1: 1999 WORLD MARKET FOR MICROPROCESSORS [16]

Chip Category	Number Sold
Embedded 4-bit	2000 million
Embedded 8-bit	4700 million
Embedded 16-bit	700 million
Embedded 32-bit	400 million
DSP	600 million
Desktop 32/64 bits	150 million

Only about 2% of the total number of chips used were in desktop and server systems, although they represent a much larger share of the revenues, since the processors per-chip costs is around two orders of magnitude of microcontrollers cost.

Most real-time systems are developed to attend a specific and complex need, requiring a high degree of fault tolerance, and they are typically embedded in a large system. Real-time systems typically include a high knowledge of the application and its environment [19], which is included statically in the project phase. The new generation of real-time systems must be designed to be dynamic and flexible regarding knowledge of the application and of the environment. They also need to guarantee the safety of the components and critical characteristics of the system [18].

Real-time systems can fail due to hardware and/or software faults, as well as by not answering within the required time constraints, which are usually imposed by the environment [13]. When the specification of a system demands that a certain task is executed by a certain deadline, the inability of the system to meet this specified constraint might be seen as a system fault, which can cause catastrophic consequences [30]. However, the simple approach of assuming the design method of a fault-tolerant system will treat missing a deadline as a system time fault; it will not attend to the needs of fault tolerance in real-time systems. The fundamental difference is that real-time systems have to be able to predict that the deadline will be missed, enabling thus the possibility of taking a certain action before such a fault occurs [13][18].

Fault tolerance is the attribute that systems must have to be able to accomplish tasks correctly in the presence of faults. Fault tolerance must be included in real-time system specifications, considering the software and hardware integration, and it must be integrated with timing constraints. Systems that demand high safety require formal proving techniques that will guarantee the characteristics of fault tolerance [19].

In this sense, fault tolerance and real-time specifications should be considered concomitantly in all phases of these types of projects. It is essential to adopt a project methodology that considers prediction in all phases, including time for fault detection, isolation, reconfiguration of the system, and recovery. Furthermore, requisites of fault tolerance can add even greater constraints to the system. For example, frequent tests and recovery routines increase the characteristic of fault tolerance; at the same time, however, they may increase the possibility of the system missing the specified deadline.

Due to the flexibility of the deadlines involved, soft real-time systems (SRTSSs) rarely require proof that the system meets its real-time performance objective. Their timing requirements are often specified in probabilistic terms. Examples of such systems include electronic games, multimedia systems and telephone switches.

Hard real-time systems (HRTSSs) have stringent timing constraints, and the consequence of missing task deadlines may be catastrophic. Fault tolerance is an especially vital requirement for HRTS development. Many embedded systems are hard real-time systems, and task deadlines in an embedded system are typically derived from the required responsiveness of the sensors and actuators, which are monitored and controlled by the embedded system. Examples of such systems are process controls, flight control, automotive systems, and life support systems. In this work, only HRTSSs are considered.

Several methods of priority have already been proposed for providing fault-tolerance to HRTSSs on a uniprocessor or multiprocessor platforms. Problems with task priority in HRTSSs have already been shown as being NP-hard class problems [1], [2], and [3].

The base of all fault tolerance techniques is contained in the redundancy concept. Initially, in the early projects with fault tolerance requisites, it was thought that redundancy was obtained through a simple replication of components. In the course of time, it was noticed that, although redundancy was fundamental to acquire this capacity, there were other redundancy types which lead to the same result, in a perhaps more efficient way.

Whenever a fault-tolerant system is designed, a redundancy type must be incorporated into the equivalent system, which does not incorporate fault tolerance requisites. Since time is the central resource of any fault-tolerant hard real-time system, task schedulability and time redundancy become the basic tools to guarantee fault tolerance requisites for a given hard real-time system.

With processor costs dropping off and the emergent development of real-time communication protocols, such as CAN (Controller Area Network)[29], TTCAN (Time-Triggered Controller Area Network)[55], TTP/C (Time-Triggered Protocol for Class C)[26], TTP/A (Time-Triggered Protocol for Class A)[25] and FlexRay [27], a great need has consequently arisen to research multiprocessor hard real-time systems and fault tolerance, where the task schedulability needs to be guaranteed for a certain fault model specified.

Single processor schedulability analysis for fixed priority tasks has received considerable attention, and it has been considerably extended by relaxing many of the

assumptions of the original computation model. Schedulability analysis for communications is much less complete.

Schedulability analysis of a distributed hard real-time system, where tasks with arbitrary deadlines communicate by message passing through a real-time bus, must consider not only the processing delays, but also the communications delays and the complications introduced by communications costs. The delays for messages being sent between processors must be accurately bounded, and the overheads due to communications must be strictly bounded.

Considering the communication issues, the spread adoption of the CAN protocol stimulated the development of this work, once it is present in most microcontrollers and DSPs incorporating real-time protocols. The sales of CAN nodes increased to a total of 200 million nodes (sold in 2001) and the conservative estimate for upcoming years is a continuous growth rate of at least 30% (Figure 1.1) [82].

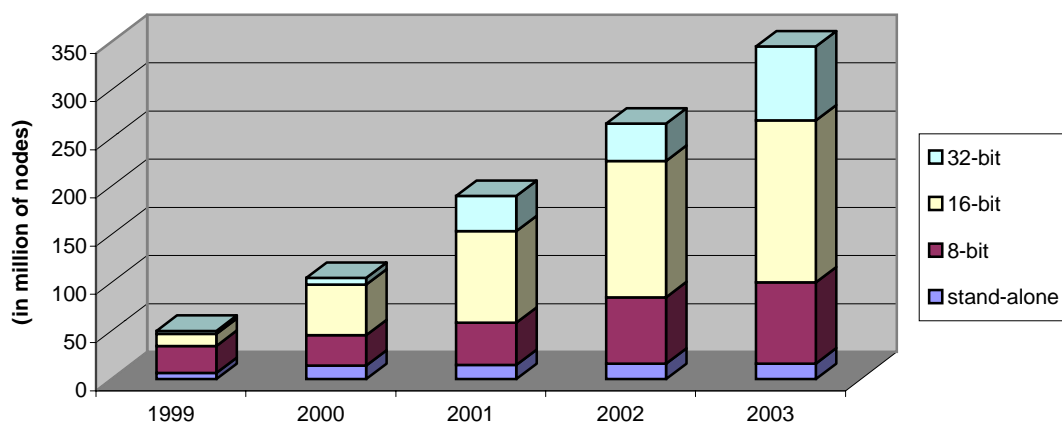


FIGURE 1.1: CAN SALES EVOLUTION

However, a well-known problem of the CAN protocol is that the delivery of low priority messages may be compromised if the bus is flooded with higher priority messages, reason why its use is avoided in hard real-time applications. The use of the CAN protocol in hard real-time applications would bring an enormous benefit to developers, enlarging the possibilities for a hardware choice and simplifying the implementation process as a whole.

## 1.2 Thesis Goals

This work presents an overview, extension and application of techniques to enhance the fault tolerance capability of multiprocessor hard real-time systems. An analysis of



techniques for fault tolerance in the context of multiprocessor real-time systems is presented, including scheduling and communication protocols.

The main goal of this work is to guarantee fault-tolerance requisites for multiprocessor hard real-time systems. Since time is the central resource of any fault-tolerant hard real-time system, task schedulability and time redundancy become the basic tools to guarantee fault tolerance requisites for a given hard real-time system.

We explore how time redundancy can be used in conjunction with hardware and software redundancy to tolerate faults in hard real-time systems. The integration of these diverse techniques into a fault-tolerant multiprocessor real-time system is shown.

An holistic analysis on distributed embedded hard real-time systems must consider the worst-case response times of each of the tasks and the complications introduced by communications costs.

From the processors point of view, the schedulability analysis for single processor systems developed by Liu and Layland [5] needed to be extended to guarantee that no task will miss its deadline due to the occurrence of a fault.

In this work we integrate fault tolerance techniques into a multiprocessor hard real-time system. This multiprocessor system considered consists of a set of processors interconnected by a CAN (Controller Area Network) bus. The basic analysis tool for fault tolerant real-time systems is task scheduling; scheduling considers early action on the prediction that timing constraints may be missed. We introduce extensions to the rate-monotonic scheduling algorithm applied to the CAN bus communication protocol. A new method to solve the CAN protocol problems is also proposed, allowing its use in hard real-time applications. The incorporation of time redundancy in conjunction with hardware and software redundancy are used to enhance the fault-tolerance capability of the CAN protocol to tolerate faults in hard real-time systems.

## 1.3 Related Work

Embedded systems exhibit stringent constraint including physical size, code size, timeliness, power and cost constraints. At the same time, if these systems fail, damage to lives and/or property can result, thereby requiring a high degree of reliability. The development of embedded systems has been one of the important research lines in the Carnegie Mellon University, where a Specification and Verification Center [24] was created and projects like

IMAGES (Integrated Modeling for Analysis and Generation of Embedded Software) [22], MoBIES (Model-Based Integration of Embedded Systems) [23] are going on.

The variety of metrics suggested for real-time systems indicates the different types of real-time systems that exist in the real world as well as the types of requirements imposed on them. Tasks can be associated with computation times, resource requirements, importance levels (sometimes also called priorities or criticalness), precedence relationships, communication requirements, and of course, timing constraints.

Several methods of priority have already been proposed for providing fault-tolerance to HRTSs on a uniprocessor or multiprocessor platform. Problems with task priority in HRTSs have already been shown as being NP-hard class problems [1], [2], and [3]. A further difficulty that occurs in the NP-hard class of problems is the scheduling of periodic tasks with arbitrary deadlines [4]. Many heuristics have been proposed to prioritize periodic tasks.

Effective scheduling involves allocating resources and time to activities in a way that allows a system to meet certain performance requirements. Scheduling is perhaps the most widely researched topic in real-time systems because many researchers believe that ensuring that tasks will meet deadlines are the key factor that distinguishes real-time systems from non-real-time systems.

The traditionally adopted dynamic approach is priority-based preemptive scheduling. In this approach, tasks have priorities that may be statically or dynamically assigned. At any given time, the task with the highest priority is executed. Preemption is necessary if a low-priority task is being executed and a higher-priority task arrives. If priorities are assigned systematically, - using the rate-monotonic approach [5], for example, - utilization boundaries can be derived. If a set of tasks does not exceed the boundaries, they can be scheduled without missing any deadlines.

Liu and Layland [5] introduced the Rate-Monotonic Algorithm (RMA) to prioritize periodic tasks in a unique processor. RMA is an algorithm for preemptively scheduling periodic tasks that designates the highest priorities to tasks with shorter periods. It is optimum when considering independent tasks running on a single processor.

The problem of scheduling periodic tasks in multiprocessors systems is considered in [1], [4], and [7]. In [1], Dhall and Liu showed that RMA is not optimum for scheduling tasks in multiprocessor systems. None of the algorithms were shown to be optimum for scheduling periodic tasks in multiprocessor systems.

Joseph and Pandya [10] demonstrated the Completion Time Test (CTT) in order to verify the schedulability of a fixed set of periodic tasks in a processor. RMA was generalized for multiprocessor systems by Dhall and Liu [1], who proposed the heuristic Rate-Monotonic First-Fit (RMFF).

Pandya and Malek [14] showed that in a process for re-executing a task that has failed in a fault-free instance, as proposed in [11], the priorities should be maintained by RMA. Under those conditions, none of the tasks will lose its deadline, even in the presence of faults, if the processor utilization factor is not higher than 0.5.

Bertossi and Mancine [15] proposed the Fault-Tolerant Rate-Monotonic First-Fit (FTRMFF) to provide fault-tolerance to a HRTSs. They use a multi-processed system where each task has a passive or active copy in another processor, and the whole set of tasks is prioritized by RMFF, supporting fail-stop of one or more processors.

The emergent development of real-time communication protocols stimulated the research in multiprocessor hard real-time systems and fault tolerance fields, where the task schedulability needs to be guaranteed for a certain fault model specified.

Tindell and Clark presented in [9] a holistic schedulability analysis for distributed hard real-time systems, extending the current analysis associated with static priority preemptive based scheduling to address the wider problem of analyzing schedulability of a distributed hard real-time system. In their work, a simple Time Division Multiple Access (TDMA) protocol is assumed and analysis developed to bound not only the communications delays, but also the delays and overheads incurred when messages are processed by the protocol stack at the destination processor.

## 1.4 Contributions

In this work, we use task schedulability and time redundancy to optimize fault-tolerance requisites for multiprocessor hard real-time systems. The schedulability analysis for single processor systems developed by Liu and Layland [5] was extended to guarantee that no task will miss its deadline due to the occurrence of a fault. This extension, called RMCAN, provides guarantees that a CAN bus can tolerate faults. A maximum transmission rate for each message instance is defined instead of pre-defining a time-driven slot (as is the main idea of TTP/C). With this restriction, a node transmission is not limited to its slot, but it may occur at any time if its transmission rate permits.

Another important contribution of this thesis is the definition of a Fault Tolerant Hard Real-Time (FTHRT) architecture consisting of processors executing in parallel exchanging data through a CAN bus. The incorporation of fault tolerance in a scenario with this characteristic includes the definition of how a scheduling approach such RMS must be applied in the tasks scheduling process and how the communication bus must be adapted to provide a guaranteed response time for the CAN bus. These enhancements introduced new possibilities for the development of systems and applications. A DSP based FTHRT architecture was implemented and a case study shows clearly the application of the concepts and ideas developed.

From the main contribution, other contributions follow:

- Definition of RMCAN, which is an extension to the CAN protocol where a recovery scheme for single or multiple faults is applied to the bus in conjunction with Rate Monotonic message scheduling.
- Application of the Fault-Tolerant Hard Real-Time architecture proposed for multiple configurations in the power electronics field.
- Application of the concepts developed and RMCAN to a real case where an Uninterruptible Power Supply is implemented to provide power to its loads.
- Make it possible to provide guarantees required for fault-tolerant execution in real-time systems.
- Formal integration of time and processor redundancies, based on the Rate Monotonic Analysis, to enhance the fault tolerance capability of multiprocessor Hard Real-Time Systems.

### 1.4.1 Publications and other contributions

A list of publications related to the thesis is present below:

- 5th IEEE Latin-American Test Workshop - **RMCAN - A protocol for multiprocessor fault tolerant architectures**. M. P. Oliveira, A. O. Fernandes, S. V. A. Campos, A. L. A. P. Zuquim, A. R. Beckler, Cartagena, Colombia, 2004.

- 9<sup>th</sup> ICC - **Guaranteeing fault tolerance through scheduling on a CAN bus.** Marcos Pêgo de Oliveira, Antônio Otávio Fernandes, Sérgio Vale Aguiar Campos, Ana Luiza de Almeida Pereira Zuquim, José Monteiro da Mata. Proceedings of the 9<sup>th</sup> International Can Conference, October/2003.
- WTF2003 – **Guaranteeing Fault Tolerance through Scheduling on a CAN Bus.** Marcos Pêgo de Oliveira, Antônio Otávio Fernandes, Sérgio Vale Aguiar Campos, Ana Luiza de Almeida Pereira Zuquim. Proceedings of the IV Tests and Fault Tolerant Workshop, May/2003.
- VIII SCTF - **Deadline: Um Núcleo Multi-Tarefas Tolerante a Falhas para Sistemas de Tempo Real.** Marcos Oliveira, Antônio Otávio Fernandes, Claudionor J. N. Coelho Jr.. VIII Simpósio de Computação Tolerante a Falhas pp.49-53. Instituto de Computação, Universidade Estadual de Campinas 1999.

An important contribution of this work is the use of the concepts and ideas in the development of a Three-phase 80Kva UPS as a fault-tolerant multiprocessor system. The concepts and ideas presented here culminated on the development of a fault-tolerant multiprocessed UPS and complemented the work developed on several other researches:

- **"Desenvolvimento de um sistema de energia ininterrupta monofásica"** - Leandro Oliveira. Master's Thesis DELT/EEUFMG 2003.
- **"Controle Digital de Uma Ups Trifásica"** - Cláudio Henrique Fortes Felix, Master's Thesis DELT/EEUFMG 2003.
- **"Power Management for Communication Intensive Real-Time Embedded Systems"** - Ana Luiza de Almeida Pereira Zuquim, Master's Thesis DCC/UFMG 2002.
- **"Desenvolvimento de uma UPS trifásica / monofásica de 6kVA"** - Paulo de Tarso Paixão Lopes, Master's Thesis DELT/UFMG 2000.
- **"Agente Proxy Embutido para Gerência de Ups"** - Wilton de Castro Padrão, Master's Thesis DCC/UFMG 1999.

- **“SupWeb-Gerenciamento de No-breaks Baseado na Web”** - Carlos Leonardo Mendes, Master’s Thesis DCC/UFMG 1999.

## **1.5 Organization**

This work is organized in the following form. In Chapter 2 real-time systems and the main concepts of fault tolerance in real-time systems are introduced. The goals of fault-tolerance projects are introduced and exemplified. In Chapter 3 the main schedulability concepts of periodic and aperiodic tasks in real-time systems are introduced and discussed. In Chapter 4 real-time protocols, with fault tolerance requisites are introduced and discussed. In Chapter 5 the proposed approaches, models, perspectives and schedule algorithms are shown. The developed approach is presented in Chapter 6. In Chapter 7 we presented some conclusions and discuss the main contributions of this work.

# Chapter 2

## Fault Tolerance in Real-Time Systems

### 2.1 Overview

An erroneous concept about real-time computation is that fault tolerance is orthogonal to the needs of real-time. It is frequently assumed that the characteristics of availability and of reliability of the system are independent of characteristics of time. However, this position does not take into consideration an inherent characteristic of real-time systems: that is, correct operation of the system is not dependent on only the logical correct result of the computation, but also on the deadline by which this logical result is reached. In other words, real-time systems can fail due to hardware and/or software faults, as well as by not answering, due to characteristics of the system, within the required time constraints, which is usually imposed by the environment [13].

In fact, if the correct logical result is dependent on characteristics of time, then separating the functional specifications of the time specifications can be a very difficult task. Furthermore, requisites of fault tolerance can add even greater constraints to the system. For example, frequent tests and recovery routines increase the characteristic of fault tolerance; at the same time, however, they can increase the possibility of the system missing the specified deadline.

When the specification of a system demands a certain service to be attended to by a certain deadline, the inability of the system to meet this specified constraint may be seen as a fault of the system. However, the simple approach of assuming the project method of a fault-tolerant system will treat missing a deadline as a time fault of the system; it will not attend to the needs of fault tolerance in real-time systems. The fundamental difference is that real-time systems have to be able to predict that the deadline will be missed, enabling thus the possibility of taking a certain action before such a fault occurs [13][18]. In this way, fault tolerance and real-time specifications should be considered concomitantly in all phases of these types of projects. The challenge is to include the needs of time and fault tolerance in the

specifications of the project on all abstract levels. It is essential to adopt a project methodology that considers prediction in all phases, including the time of the fault detection, isolation, reconfiguration of the system, and recovery.

## 2.2 Real-time Systems

Typically, a real-time system consists of a controlled system and a driver system. For example, in an automated factory the controlled system is the factory floor plus their robots, mounting stations and pieces to be mounted. The controller system is the computer and the man-machine interface, which manages and coordinates the activities on the factory floor. Therefore, the factory itself can be seen as the environment with which the computer must interact.

Real-time systems are characterized by the fact that severe consequences will occur if the system's logical results and time constraints are not reached. As explained, real-time systems differ from traditional systems because of the deadline involved and because of other characteristics of time, which are linked to the tasks specific constraints. Therefore, the system is in a position of attributing an explicit commitment between performance and a correct logical result. Faults, including time faults, can cause catastrophic consequences [30]. This means that, different from systems that separate a correct result from a deadline, real-time systems strongly interlink a correct logical result and performance in accordance with a deadline. In this way, real-time systems need to solve the problem of missing the deadline, taking into consideration the specific context of the target application. The sooner that the system determines that it will miss the deadline, the sooner the system can readjust its decision-making based on the new context of the application [18].

Real-time systems are based on the premise that the worst-case execution time of the program is precisely known. The worst-case execution time of the program is dependent on the system hardware, the operating system, the compiler and the programming language used [18]. Many authors proposed different applications of the worst-case execution-time analysis for different processor architectures [83][84][85]. Many characteristics of underlying hardware that have been introduced to increase performance affect the problem of finding the worst-case execution time [31]. In this way, the omnipresence of caches as well as of pipelines, dynamic RAMs, virtual memory, etc., makes the behaviour of the hardware non-deterministic. Likewise [32], compilers with optimizers that have been implemented targeting



to obtain the best performance for a specific architecture can contribute to a poor prediction of the execution time of the generated code. These techniques, although useful in other ways, are not compatible with the goal of predicting the worst-case execution times. Examples of these optimization techniques include “constant folding”, “value propagation”, “redundant-assignment elimination”, “partial-redundancy elimination”, “common sub-expression elimination”, “flow optimization”, “dead-code removal”, “loop-invariant code motion”, “strength reduction”, “induction variable elimination”, and “register allocation by coloring”[18][33][34]. System interference in the form of servicing interrupts, referring to shared memory and context-switching also complicates the situation. In short, any attempt to determine execution time statically in real-time programs reveals a high level of complexity.

The characteristics of time with regard to a task can be arbitrarily complex, but among the most common characteristics is the one of periodicity. A periodic task has a deadline by which it should initiate or conclude; it can also have a double characteristic at the beginning and at the terminus. This characteristic of periodicity of tasks allows one to separate verification of the correct logical operation of the computation from verification of the deadline at which this result is reached, as the next chapter explains in detail.

## 2.3 Fault Tolerance

There are a great number of referring concepts, which underly fault-tolerant computation. It is important to have these concepts well defined in order to fully understand the subject.

First we will define the concepts of fault, error and failures. A fault corresponds to a physical defect in the hardware or in the software. For example, we may have two tracks of a printed circuit board connected in a short circuit or a loop that does not change the output terms and so does not consequently finish when initiated. An error is a manifestation of a fault. In other words, the program does not go out of the loop because there is a fault, which does not verify correctly the termination terms. Finally, there is failure, which is the manifestation of the error. Perhaps the computer stops because the program stayed in the loop infinitely. Note that, clearly, there is a cause and effect relationship, in sequential order, among these three definitions.

Another definition is for the latency of a fault, which is the time between the occurrence of the fault and the occurrence of the error. A latent fault is a fault that is present in

the system, but the error associated with it has not yet been manifested. Likewise, error latency is the time between the occurrence of the error and of the failure. Thus, the time between the occurrence of a fault and the failure in the system is the sum of the latency of the fault and the latency of the error.

An important aspect for a fault-tolerance study consists on describing exactly the characteristics of faults. There is a set of attributes that serves this purpose exactly; they are: cause, nature, duration, extension and value. Figure 2.1 [21] illustrates this characterization, showing the different types for each fault characterization attribute.

Regarding the cause, a fault can originate in specification problems, implementation problems, defects of components (which are not uncommon for electronic devices), or in external factors, such as, storms, dust, temperature, etc.

Regarding its nature, a fault can belong to software or hardware. In the latter, the fault can be in the analog part, for example, in transducers and amplifiers, or in the digital part, for example, in the arithmetic logic unit.

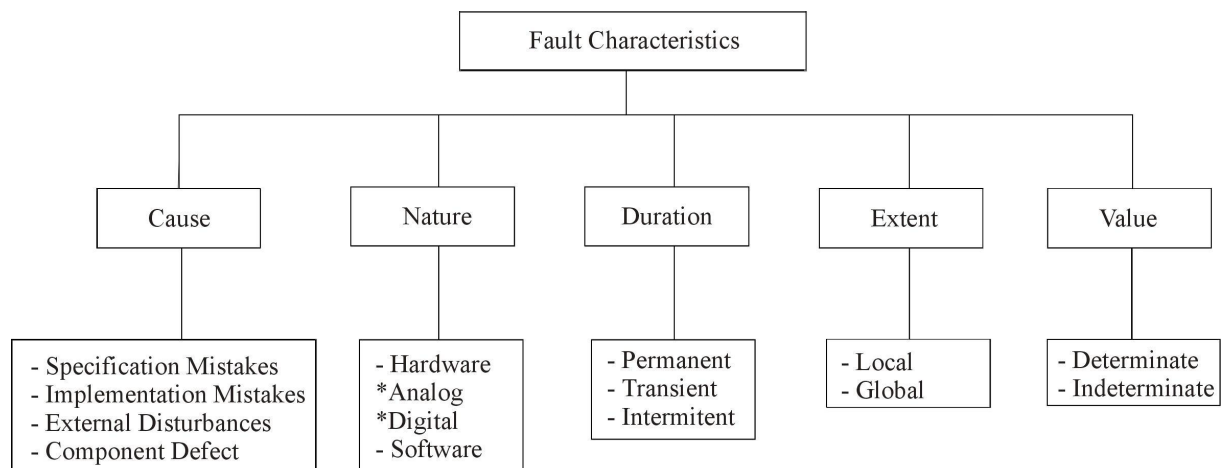


FIGURE 2.1: ATTRIBUTES OF FAULT CHARACTERISTICS

Regarding the duration, a fault can be constant, which means that once that it has occurred it persists in the system until the adequate maintenance is done. It can still be transient, whereby it occurs for a time period and then disappears. This type of fault is usually provoked by external causes. Lightning, for example, can provoke a sudden error in a transmission line, but after the lightning, the line will go back to its normal operation. Finally, there are faults, which are called “intermittent”; these occur for short periods of time and disappear, but then they come back again. It is possible for this process to repeat itself indefinitely. An intermittent fault is the repeated occurrence of a transient fault. This last type

can be caused by a project specification that enables a certain component to work on certain critical conditions, such as temperature, for example.

The extent of a fault can be global or local. That is, a fault may affect the whole system or be restricted to a given block. The value of a fault can be certain or not determined. That is the relative values of a fault can be constant or not. For example, a fault that always provokes a given output in “1” is called a determinate fault. A fault without this characteristic is said to be indeterminate. Correlated faults are those, which occur simultaneously in all hardware modules. For example, electromagnetic radiation may simultaneously affect all hardware modules in a space shuttle.

Another important concept is the fault model. A fault in the system can assume several forms, and this makes a system’s test process for fault detection extremely complicated. To reduce this problem, the faults that occur in the system are supposed to follow a certain standard, or a model. Thus, the problem of how to treat the faults becomes much more simplified. Although not all faults in a real system occur according to a fault model, there are fault models that are able to cover a very large percentage of the faults that do occur. Like the fault-model example, the model known as the Logical Stuck-Fault assumes that a fault is always an input or an output of some logic port that becomes stucked in “0” or “1”. This model is widespread in the literature, precisely for its simplicity and for the excellent practical results obtained by using it even though it does not cover all scenarios [21].

Fault tolerance techniques include concepts such as Fault-masking and Reconfiguration. Fault-masking is a technique that allows a fault that may occur internally in the system to be completely ignored. An example of a situation of this type is the memory case with an error correction circuit. Reconfiguration is a process, which eliminates a defective unit in a system and restores it to some condition or operational state. If this technique is used, the project designer must be concerned with the following:

- Fault detection is the recognition process of the occurrence of a fault.
- Fault location is the process of determining where a fault has occurred in the system.
- Fault containment is the process of isolating a fault so that it does not produce an error.

- Fault recovery is the final process that keeps a system operational or returns it to an operational state in the presence of a fault.

The base of all fault tolerance techniques is contained in the redundancy concept. Initially, in the early projects with requisites of fault tolerance, it was thought that redundancy was implied in a simple replication of components and that, having that, a fault-tolerant system would exist. In the course of time, it was noticed that, although hardware redundancy was fundamental to acquire this capacity, there were other redundancy types which lead to the same result, in a perhaps more efficient way. Therefore, the redundancy concept in the context of fault-tolerance does not involve simply duplication, but it also involves increasing any resource that is not explicitly necessary to accomplish the task [21]. Therefore, different forms of redundancy have been created, such as following:

- **Hardware redundancy** is the addition of extra hardware; it is frequently used for the purpose of detecting and tolerating faults.
- **Software redundancy** is the addition of extra software in addition to what it is simply necessary for the execution of the task.
- **Information redundancy** is an increase of information in the system besides that which is essential data for accomplishment of the desired task.
- **Time redundancy** is the utilization of additional operation cycles to obtain higher indices of fault tolerance.

## 2.4 Hardware redundancy

There are three hardware redundancy types for utilization in fault tolerance [21]: passive, active and hybrid.

The techniques of passive redundancy are those in which the extra hardware is added to provide fault masking. In these methods fault tolerance is obtained by masking, which dispenses a detection action, locating the faults and recovery. The faults are just masked and the system proceeds with its normal operation.

Active techniques, also called the dynamic method, implement fault tolerance by detecting and performing an action to remove the faulty hardware from the system. Retroactive techniques are those in which reconfiguration of the system is necessary.

Hybrid techniques combine the advantages of the other two. Fault masking is used to prevent the production of an error, while detection and fault location are used to identify the faulty hardware to be replaced by a spare.

This work is particularly interested in active hardware redundancy because improvements in fault tolerance attributes obtained up to now have been based on the utilization of specific types of active redundancy hardware: the watchdog timer and processor.

### **2.4.1 Watchdog Timer**

The fundamental concept of this technique is that the absence of an action means that there is a fault. The circuit is designed so that it periodically signals that it is, indeed, operating. If the signal takes longer than the specified time, then the watchdog timer assumes that a fault has occurred, and the recovery procedure is activated [21].

The watchdog timer should be rebooted periodically in a given frequency. A failure in the system is identified by the absence of rebooting of the watchdog timer. The fundamental idea is that the system is fault-free if the timer is rebooted in the specified frequency.

The frequency in which the timer has to be rebooted is a function of the system. In a flight control system, for example, a fault needs to be detected by 100 ms from the moment of its occurrence [21]. Consequently, the watchdog timer must be rebooted at intervals of 100 ms or less to allow a fault to be detected before some catastrophic effect of the fault occurs.

The watchdog timer provides good fault detection for some fault types. For example, the watchdog timer can detect if the processor simply stops working. If the processor is overloaded, additional time will be necessary to complete its tasks, and the watchdog timer can detect this abnormality. In addition, the watchdog timer is particularly efficient in detecting the absence of an answer.

At present, the use of the watchdog timer concept can be extended because those that have already been implemented in modern micro-controllers and DSP's allow for the detection of rebooting of the timer at a frequency higher than what has been specified [35][36]. Thus, the system should reboot the watchdog timer within a certain frequency. This also enables the detection of a fault in the case in which the processor fails but continues rebooting the watchdog timer at a rate greater than the one specified, which means that some fault has occurred.

The example of conventional postal service is similar. If a person stops receiving mail for three or four days, he does not know whether there was really any mail for him or if the

post office service failed. However, if the person makes an arrangement with the mail carrier to leave a piece of blank paper in the box daily, even if there is no mail, then he will know that everything is going well when finding a piece of white paper in the box. If nothing is found in the box, it means that the postal service failed. Similarly, finding more than one piece of paper in the box a day also means that something is wrong because the mail carrier delivers only once a day.

The watchdog timer can be used to detect faults not only in software but also in hardware. In many applications, tasks should be run in intervals of specified times. In systems with digital control, the tasks are run repetitively at an interval of specified time. If the task is suddenly run on a frequency band outside the specification, a fault has probably occurred and the watchdog timer will send a signal, which can be treated by a priority interruption, such as NMI (non-maskable interrupt) or the RESET of the CPU.

## **2.5 Software redundancy**

Software redundancy is very difficult to obtain. It is important to notice that, different from hardware, it is not enough that the software be duplicated to obtain fault tolerance since the faults would be duplicated as well.

There are three basic ways to obtain software redundancy that can be applied in order to obtain fault tolerance [21].

The first is the consistency check. Consistency tests use a priori knowledge about the characteristics of information to verify if they are correct. For example, in a given application, it is defined that a certain information field never surpasses a certain magnitude. Consistency tests can verify if this error type is present. This technique, although simple, is a form of very important redundancy.

There are also capacity checks (operation) to find out whether the memory or the arithmetic logical unit (ALU) in a given system is working. In order to do that, it runs an operation set (with the memory or with the ALU), and by itself compares the results obtained with the expected results (which can be recorded in ROM for the operations case with ALU). Thus, it is possible to identify faults that may occur in these units by using software redundancy.

Finally, there is the technique called N-Version Programming, which produces “n” versions of the software and compares the obtained results. However, this technique is very

difficult to implement in practice because it requires different teams, which can't communicate about software design, its implementation, or the overall project itself. There are two main problems with this method. The first one is that programmers tend to make the same mistakes; in other words, it is not possible to guarantee that two programs developed entirely independently will not contain the same errors [21]. Second, both programs will be produced from the same specifications; in other words, errors originated in the specifications will be shared by several modules [21].

## 2.6 Information redundancy

Information redundancy appends information to the data in a way that allows detection, masking, and even fault tolerance [21]. The way to do that is through utilization of codes for the data, and the simplest example is parity detection codes.

A code is a way to represent information, following a set of well-defined rules. For example, the different whistle combinations of a traffic policeman represent different codes. A code word is a collection of symbols that represents particular data; in the previous example, one short whistle means, "Proceed". A code is called binary if there are just two symbols used to represent the data. A word is considered valid if it adheres to all the rules specified for the code or is otherwise disabled.

The coding process determines the code word that represents particular given data, and the decoding process accomplishes the inverse operation. In other words, a given code determines what the given representation is.

Coding can be used not only for the information that it wishes to represent, but also for error detection. This way, a coded piece of information (which also adds a code used to detect errors) occupies an extra, redundant space, as shown Figure 2.2.

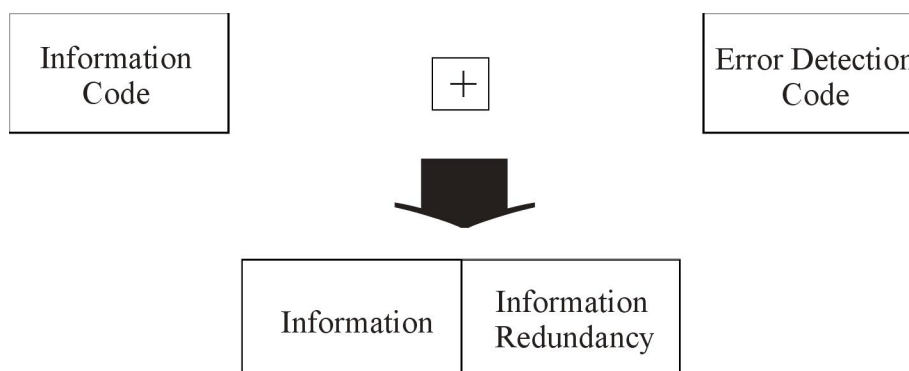


FIGURE 2.2: INFORMATION REDUNDANCY

An error detection code is a code where the set of specified rules allows for error detection, and an error correction code allows for the given correction to be recovered from a word of the wrong code. A fundamental concept for the characterization of these codes is the Hamming distance. The Hamming distance between any two binary words is the number of bit positions in which the two words differ. The code distance is the minimum Hamming distance between any two valid code words. We clearly notice that for a code to allow for the fault detection of a single bit, the distance should be two. This way, a bit changed within a code word would become an invalid word. In general, a code can correct up to “c” bit errors and detect up to “d” additional bit errors if and only if

$$2c + d + 1 \leq H_d,$$

where  $H_d$  is the Hamming distance of the code [37].

As examples of error detection codes, we can cite the parity code, the checksum and the cyclic redundancy check (or CRC). The latter is much used in packet transfer in computer networks [38].

## 2.7 Time Redundancy

For hardware and information redundancy an extra quantity of hardware must be appended to the project so that it can attend the fault tolerance requisites. Hardware has a direct impact on project factors such as weight, size, power consumption and cost. Therefore, time redundancy has been taken into consideration in more recent projects, mostly with the processor performance gain that we have seen in the last few years.

In Johnson [21], this redundancy type runs a computation more than once and verifies the result obtained in each. If a discrepancy occurs, the computation can be run again with the goal of verifying whether the discrepancy exists or not. Thus, the system is protected against errors of the transient type besides allowing the identification of transient, intermittent and constant errors.

The use of time redundancy decreases the weight and cost of systems while providing tolerance for transient, intermittent and correlated faults. Computations shifted in time have been used earlier to tolerate correlated faults [40][41]. A motivation for the use of time redundancy can also be found in [42].



## 2.8 Goals of Fault Tolerance Projects

Among applications where fault tolerance is a particularly important attribute for a project, one can highlight applications of long duration, such as implanted systems on board spaceships or satellites, which need to remain in orbit or continue their journeys for long periods of time. We can also cite applications of critical computation, where an error can provoke a disaster, for example, airplane control, dangerous chemical processes, or even nuclear plants or military applications. There are also high availability applications, such as computers for banks or sale of airline tickets, etc.

There is still another application class in which faults need to be supported until the required maintenance can be accomplished. This class includes telephone systems, which are often in distant locations and where the cost of maintenance is high. Incorporating fault tolerance in these systems makes scheduling periodic maintenance feasible, and these systems will be better able to tolerate the faults that may occur between technicians' visits.

Real-time applications demand that the project design encompass certain performance characteristics. It is precisely this subset of characteristics that defines the degree of fault tolerance demanded by the system. These characteristics, as follows, are designated goals of the project [21].

- **Reliability** is a function of time  $R(t)$ , and it represents the probability of the system working correctly in the time interval  $[t_0, t]$  given that the system is operating correctly in the time instant  $t_0$ . It is important to point out the difference between fault tolerance and reliability. A system with low reliability can have this characteristic improved by fault tolerance; the faults will still occur with a certain probability but, because of the tolerance, the system will not fail. In addition, a system with high reliability may not be fault tolerant, since even though the probability that a fault occurs is low, there will be a failure in the system when the fault occurs.
- **Availability** is another function of time,  $A(t)$ , which defines the probability of the system working correctly in the time instant  $t$ . Although faults can occur with a high frequency, if the intervals of non-operation are short enough, the availability of the system could still be high. This characteristic is particularly

significant in applications where it is important for the system to be available when it is requested to do so.

- **Safety** is the probability of the system either running the operation correctly or discontinuing its functions in a manner that does not disrupt the operation of other systems or compromise the safety of any people associated with the system. In other words, if the system is going to fail, it should at least fail in a safe situation; for example, if the automatic pilot of a plane fails, then the system must not lock the pilot's commands.
- **Performability** is a function  $P(L,t)$  that defines the probability of a system being at a level  $L$  of performance in the instant  $t$  of time. This characteristic is classic for multiprocessor systems. When a processor of the system fails, the system simply decreases its performance, but it continues running its tasks correctly. That is called graceful degradation of the system.
- **Maintainability** is a function  $M(t)$  that is the probability of a system with a fault coming back to a functional state within the period  $t$  of time. The restoration process includes the location of the problem, the repair and the return of the system to a functional state.
- **Testability** is the capacity of testing some attributes of a system. Testing is a way to determine the existence and the quality of some system attributes.
- **Dependability (global robustness dependence)** encompasses all the previous concepts. This characteristic determines the quality of the service provided by a certain system while the others are ways to quantify the dependability of a system.

The design of a system is often conducted with a set of goals to be reached. For many of these goals, principally those cited above, fault tolerance techniques can be very helpful. Figure 2.3 [21] shows a top-level view of a system project. Note that several goals that the project is intended to reach are verified in two ways: the design itself and the evaluation of the system.

Within the design of the system, there are basically two ways of fulfilling the requisites of the design. They are fault prevention and fault tolerance. These two ways, used together,

enable the attainability of high reliability indices, availability, or whatever goal is sought for the system. Fault prevention involves techniques such as a rigorous selection of devices and utilization of project rules. Tolerance techniques involve various levels of redundancy.

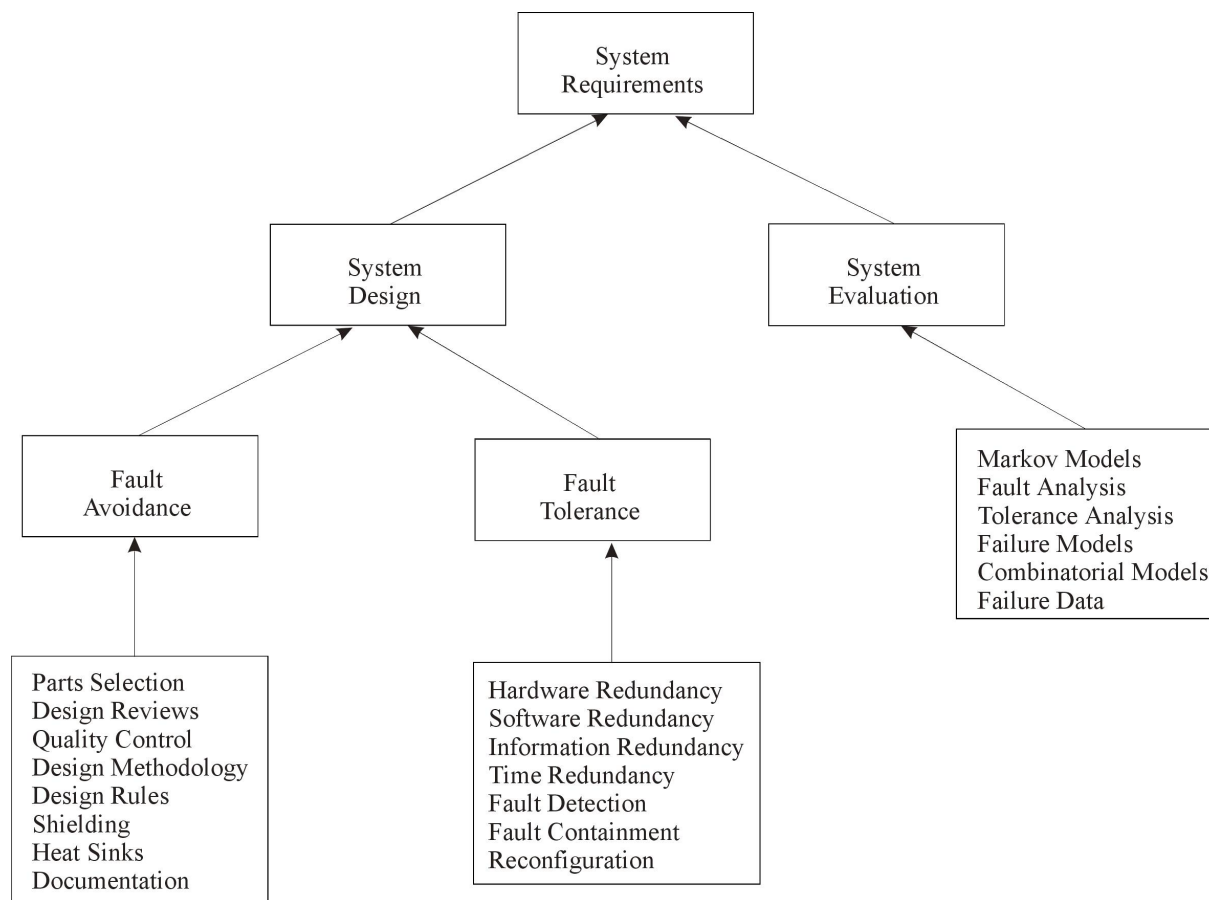


FIGURE 2.3: A TOP-LEVEL VIEW OF THE SYSTEM DESIGN PROCESS.

Evaluation of the system must be conducted along with the project in order to be successful. In the evaluation, including critical analysis that follows the recommendations of ISO9001 [43], many problems can be discovered and corrected before the system is implemented.

# Chapter 3

## Real-Time Fault Tolerance Scheduling

### 3.1 Overview

A scheduling algorithm is a set of rules that determine the task to be executed at a particular moment. The traditionally adopted dynamic approach is priority-based preemptive scheduling. In this approach, tasks have priorities that may be statically or dynamically assigned. At any given time, the task with the highest priority is executed, and preemption is thus necessary: if a low-priority task is being executed and a higher-priority task arrives, the former is preempted and the processor is given to the new arrival. Thus, the specification of priority-based preemptive scheduling algorithms amounts to the specification of the method of assigning priorities to tasks. A scheduling algorithm is said to be static if priorities are assigned to tasks once and for all. A static scheduling algorithm is also called a fixed-priority scheduling algorithm. A scheduling algorithm is said to be dynamic if priorities of task might change from request to request. A scheduling algorithm is said to be a mixed scheduling algorithm if the priorities of some of the tasks are fixed, yet the priorities of the remaining tasks vary from request to request. If priorities are assigned systematically, using the rate-monotonic approach [5], for example, utilization boundaries can be derived. If a set of tasks does not exceed the boundaries, they can be scheduled without missing any deadlines.

Scheduling policies in real-time systems need to ensure that tasks will meet their deadlines under all circumstances, even in the presence of faults. These real-time scheduling schema may be used for the processor, as well as for tasks, communication and other resources that are used by real-time systems, including I/O in general. The tasks in a real-time system have time constraints, such as arrival times, ready times, deadlines, periods and execution times. A real-time system must provide predictable response times. Therefore, the worst-case behaviour of real-time systems is of primary importance, as opposed to average response time and user convenience, for example, which are important issues in general-purpose computing systems [17].

Scheduling and resource allocation in real-time systems are difficult problems due to the timing constraints of the tasks involved. The order in which the tasks are scheduled or dispatched is very important in order to determine if the tasks set will meet their deadlines. Many real-time scheduling problems are known to be NP complete [3][44].

Fault tolerance requirements make a real-time system even more complicated to implement, since faults must be detected and tolerated within the system timing constraints. If a fault triggers a backup task for recovery purposes, the backup task must also be executed before the task deadlines. Due to these complexities, most present days real-time systems only deal with timing constraints, and not with the catastrophic consequences that a system failure might cause.

Effective scheduling involves allocating resources and time to activities in a way that allows a system to meet certain performance requirements. Scheduling is perhaps the most widely researched topic in real-time systems, because many researchers believe that deadlines are the key factor that distinguishes real-time systems from non-real-time systems. Thus, they reason that the basic problem in real-time systems is to ensure that tasks meet their deadlines.

The metrics that guide scheduling decisions depend on the application areas. The need for minimizing schedule length pervades static non-real-time systems, while minimizing response times and increasing throughput are the primary metrics in dynamic non-real-time systems. In static and dynamic real-time systems, however, the main goal is to achieve timeliness, what introduces quite different metrics according to the type of real time system and the requirements imposed on them.

Consequently it is, sometimes, hard to compare different scheduling algorithms, once different types of real time systems and task characteristics occur in practice. Tasks can be associated with computation times, resource requirements, importance levels (sometimes also called priorities or criticalness), precedence relationships, communication requirements, and of course, timing constraints. If a task is periodic, its period becomes important; if it is aperiodic, its deadline becomes important. Both periodic and aperiodic tasks may have start time constraints.

## **3.2 Rate Monotonic Theory**

The term rate monotonic analysis was born in 1973 with the publishing of "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment" by Liu and Layland

[5]. Their paper laid out the basis for a simple test to be applied to a real-time system to determine if a set of periodic tasks would be guaranteed to meet their deadlines. Tasks deadlines are taken to be at the end of the periods of the tasks, and tasks are not permitted to block at run-time. Furthermore, each task is assigned a unique priority monotonically with task period, and hence the name rate monotonic scheduling (RMS).

Rate monotonic scheduling systems use rate monotonic (RM) theory for scheduling sets of tasks. Rate monotonic analysis can be used on tasks scheduled by many different systems to reason about schedulability. A task is schedulable if the sum of its blocking, preemption and execution time is less than its deadline [47]. A set of tasks is schedulable if all tasks meet their deadlines. Rate monotonic analysis provides a mathematical model to check the schedulability of a set of tasks.

The rate-monotonic algorithm (RMA) is a task schedule algorithm that assigns higher priorities to the tasks with shorter periods, and the RMA is optimum when the tasks are independent [5]. It assumes that all processes in the system are periodic, have deadlines at the end of their periods, and are totally independent of each other [51].

These premises allow the complete characterization of a task by its period and its run-time, which is the maximum processing time for that task. A set of periodic tasks is called schedulable if every periodic task finishes its execution before the end of its period. This way, any set of independent periodic tasks is schedulable by the rate-monotonic algorithm if the condition of Theorem 1 is met [5].

**Theorem 1:** A set of  $n$  independent periodic tasks scheduled by the rate-monotonic algorithm will always meet its deadlines, for all task instances if:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \quad (3.1)$$

Where:

$C_i$  – Worst-case run-time of a given task  $i$ ;

$T_i$  – Period of a given task  $\tau_i$ .

**Proof:** See Liu and Layland [5].

Theorem 1 offers sufficient conditions (worse case) to characterize schedulability by RMA. At its limit, the theorem converges to 69.3% when the number of tasks tends towards

an infinite number [6]. The analysis provides a schedulability test by giving a utilization bound. In common practice, the rate monotonic algorithm can often successfully schedule task sets having total utilization higher than 0.693. The utilization bound of Theorem 1 is very pessimistic because the worst-case task set is contrived and rather unlikely to be encountered in practice. For a randomly chosen task set, the likely bound is 88% [6]. This suggests that the average case behaviour is substantially better than the worst-case behaviour. The behaviour is strongly dependent upon the relative values of the periods of the tasks comprising the task set.

Lehoczky presents in [45][46] a stochastic analysis that gives the probability distribution of the utilization boundary of randomly generated task sets. Specifically, a task set is generated randomly, and the computation times are scaled to the point at which a deadline is first missed. Theorem 2 provides an exact criterion to test for schedulability of independent and periodic tasks using RMA. In fact, the theorem checks if each task can complete its execution before its first deadline by checking all the scheduling points.

**Theorem 2:** A set of  $n$  independent periodic tasks scheduled by the rate-monotonic algorithm will always meet its deadlines, for all task instances, if and only if:

$$\forall i, 1 \leq i \leq n, \frac{\min_{(k,l) \in R_i} \sum_{j=1}^i C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil}{lT_k} \leq 1 \quad (3.2)$$

Where:

$C_j$  – Execution time;

$T_j$  – The period of task  $\tau_j$ .

$$R_i = \left\{ (k, l) \mid 1 \leq k \leq i, l = 1, \dots, \left\lfloor \frac{T_i}{T_k} \right\rfloor \right\}$$

**Proof:** It can be found in [45][46].

To know if a set of given tasks with utilization greater than the bound of Theorem 1 can meet its deadlines, the conditions of Theorem 2 can be checked [6][45][46].

An important improvement in the theory was made by including aperiodic events. Sha *et al* [50][52] contributed with a number of strategies, ranging from the simple casting of aperiodic events into a pessimistic periodic framework to the more elaborate strategy of involving implementation in the careful rationing of time to incoming aperiodic events [52].

Another important enhancement was made by Sha *et al* [52] to take care of inter-task cooperation. The most common case of inter-task cooperation is the situation in which more than one process must use a single resource, requiring some kind of resource allocation policy. One major problem with the state-of-the-art resource allocation up to this point had been priority inversion, once it can prevent a highest priority task from meeting its deadline. Sha *et al* [52] derived a run-time algorithm to permit tasks to lock and unlock semaphores according a protocol, termed the Priority Ceiling Protocol. With this protocol, a system is guaranteed to be free of deadlock (on single processor systems), and a given task can be blocked at most once by a lower priority task. Sha *et al* [52] extended the rate monotonic analysis to account for the behaviour of this protocol, adding a blocking factor – worst-case time a given task can be blocked - to the schedulability equations.

These three works provide the basis for an exact schedulability test for sets of independent periodic tasks under the rate monotonic algorithm.

Pandya and Malek [14] analyzed the schedulability of a set of periodic tasks that is scheduled by the RMS policy and is susceptible to a single fault. The recovery action is the re-execution of all uncompleted tasks. They showed that in a process for re-executing a task that has failed in a fault-free instance, the priorities should be maintained by RMA. Under those conditions, none of the tasks will lose its deadline, even in the presence of faults, if the processor utilization factor is not higher than 0.5.

Tindell [8] derived analysis for static priority pre-emptive systems that permit tasks to have arbitrary deadlines, release jitter, and behave as sporadically periodic tasks. The derivation of his analysis illustrated how a window approach to finding worst-case response times for these tasks is an appropriate way of obtaining an analysis tailored to the behaviour of real-time tasks.

The priority pre-emptive dispatching algorithm has also been analyzed by Joseph and Pandya [10] to find the worst-case response time of a given task. Analysis is derived that finds the worst-case time between a task being released and completing the execution or a worst-case required computation time. This permits tasks deadlines to be less than task periods. Joseph and Pandya showed also that the Completion Time Test (CTT) can be used in order to verify the schedulability of a fixed set of periodic tasks in a processor.

Dhall and Liu [1] proposed, among others, the Rate-Monotonic First-Fit (RMFF) heuristic. It is a partitioning algorithm, where tasks are first assigned to processors following the RM priority order and then all the tasks assigned to the same processor are scheduled with



the RM algorithm. The problem of scheduling periodic tasks in multiprocessor systems is also considered [4], [7], and [49]. RMA was generalized for multiprocessor systems by Dhall and Liu [1], and it was also shown that RMA is not optimum for scheduling tasks in multiprocessor systems. None of the algorithms were shown to be optimum for scheduling periodic tasks in multiprocessor systems.

Bertossi et al [15] extended the Completion Time Test so as to check the schedulability on a single processor of a task set including backup copies. They used a multiprocessed system where each task has an active and a passive copy in another processor, and the whole set of tasks is prioritized by Rate-Monotonic First-Fit (RMFF). Fail-stop of one processor is thus supported. Bertossi et al proposed the Fault-Tolerant Rate-Monotonic First-Fit algorithm to provide fault-tolerance to a Hard Real Time System where all the task copies, including the backup copies, are considered by Rate-Monotonic priority order and assigned to the first processor in which they fit.

De Oliveira and Fernandes [11][12] proposed the Deadline, a portable multitasking kernel and fault tolerant for hard real-time systems applications. In their work, a processor and RMA are used to designate priorities and to schedule tasks, providing fault tolerance to the system through the inclusion of only one hardware redundancy, a “watchdog timer”.

### 3.3 Fault Tolerance and Rate Monotonic Scheduling

Due to the critical nature of the tasks in hard real-time systems, it is essential that faults be tolerated. Several studies have shown that space applications, which have very high reliability requirements, also have very high fault frequencies [17]. Therefore, tolerance of transient faults is essential in such applications.

Transient faults in real-time systems are generally tolerated using time redundancy, which involves the re-execution of any task running during the occurrence of a transient fault [53]. Ghosh *et al* presented in [48] a scheme to guarantee that the execution of real-time tasks can tolerate transient and intermittent faults assuming any queue-based scheduling technique.

The general approach to fault tolerance is to maintain enough slack (backup time) in the schedule, so that any task instance can be re-executed if a fault occurs during its execution. If no faults occur, tasks are executed following the usual RMS scheme and the slack is not used. If a fault occurs in a task, a recovery scheme is used to re-execute that task. The ratio of slack  $S$  available over an interval of time  $L$  is thus constant and can be imagined to be the

utilization of a backup task B, where  $S/L$  is the backup utilization. If the backup utilization is  $U_B$ , and the slack available during an interval  $L$  is denoted by  $B_L$ , then  $B_L = U_B L$ .

A backup task can be seen as occupying a slack time slot between every two consecutive period boundaries, where a period boundary is the beginning of any period. Therefore, the length of the backup slot between the  $k^{\text{th}}$  period of  $\tau_i$  and  $l^{\text{th}}$  period of  $\tau_j$  is given by  $U_B(IT_j - kT_i)$ , where there is no period boundary of any other task in the system between times  $kT_i$  and  $lT_j$ . It is important to note that the backup slot is merely an abstraction to help reason about slack in the processor.

In [17] Ghosh showed a recovery scheme for single and multiple faults that ensures the re-execution of any task after a fault has been detected. The following conditions must be satisfied:

[S1]: There should be sufficient slack for every instance of each task to re-execute. That is, the slack between  $kT_i$  and  $(k + 1)T_i$  should be at least  $C_i$  for any value of  $k$  and  $i$ .

[S2]: When any instance of  $\tau_i$  finishes executing, all the slack available within its period (at least  $C_i$  if [S1] holds) should be available for the re-execution of  $\tau_i$ .

[S3]: When a task re-executes, it should not cause any other task to miss its deadline.

[S1] ensures the availability of sufficient slack for a task to re-execute, and [S2] ensures that this slack can be used after a task finishes executing to re-execute that task before its deadline if a fault is detected. [S3] allows all tasks to meet their deadlines even when a high priority task needs to re-execute.

**Proof:** For single and multiple faults, it can be found in [17].

If these three conditions are met, then it is possible to re-execute a faulty task and meet its deadline. However, a recovery scheme must define also how the slack should be used and a very straightforward scheme consists on the faulty task simply being re-executed at its own priority.

This approach to distribute slack in the schedule can be applied to any non-fault-tolerant scheduling scheme for preemptive, periodic tasks where the RMS assumptions hold. As shown by Ghosh, any computation time  $C_i$  in the non-fault-tolerant scheme can be split into two parts for the fault-tolerant scheme: a new computation time  $C_i' = C_i(1 - U_B)$  (where  $U_B$  is the backup utilization) and a slack equal to  $C_i U_B$ . To ensure that in the worst case

scenario each task  $\tau_i$  can re-execute before its deadline,  $\tau_i$ 's critical instance - defined as the time at which  $\tau_i$ 's response time is maximized - is considered. The critical instance of a task  $\tau_i$  happens when  $\tau_i$  arrives simultaneously with all higher priority tasks [5]. The total slack available for any task  $\tau_i$  at its critical instance is equal to the total slack available within a period boundary, which is defined as the beginning of a period. Ghosh showed also that, by splitting up each transmission time  $c_i$  into a new transmission time  $c_i'$  and slack, as described above, the utilization of each task  $\tau_i$  is reduced to  $U_i (1 - U_B)$ , and thus the following general fault tolerance boundary for an RMS ( $U_{G-FT-RMS}$ ) is obtained:

$$U_{G-FT-RMS} = n (2^{1/n} - 1) (1 - U_B) = U_{LL-RMS}(1 - U_B) \quad (3.3)$$

The above equation is a general one applicable to a RMS for any value of  $U_B$ . If  $U_B = \max\{U_i\}$ ,  $i = 1, \dots, n$ , then any task instance in the system can tolerate a single fault. Any number of faults can be tolerated if [S1] holds.

Multiple faults within two consecutive period boundaries are also guaranteed to be tolerated using the scheme described above. If several backups are provided in the system, and the total backup utilization is  $U_{BT}$ , then a general boundary for the task set can be derived by replacing  $U_B$  with  $U_{BT}$  in (3.1); that is, the new boundary is  $U_{LL-RMS}(1 - U_{BT})$ .

The hyperperiod is the least common multiplier of tasks periods. The release pattern will be repeated at every hyperperiod. We may define the hyperperiod of a task as the least common multiplier of the periods of the tasks of higher or equal priority than that task from the point of view of a task at level the pattern of invocations of higher priority tasks is repeated every time units. Hyperperiod is the minimum time interval after which the schedule repeats itself.

If [S1], [S2], and [S3] holds, Equation 3.1 will also apply for the hyperperiod of a task as well as for the hyperperiod of a task set.

### 3.4 Multiprocessor Systems

The easiest way to provide fault tolerance in a multiprocessor system is to use spare processors. The well-known RMS techniques for uniprocessor systems can be used to schedule tasks on individual processors, and spare processors can be added to tolerate permanent processor faults.

The approach of adding slack to a schedule of real-time tasks is not appropriate for tolerating permanent processor faults in multiprocessor systems. Slack can be used only to tolerate transient faults. To tolerate permanent faults, multiple copies of each task must be scheduled on different processors.

Any critical real-time system must tolerate permanent faults in addition to transient faults, and transient faults need to be detected before they can be tolerated. A combination of time and space redundancy can be used to detect and tolerate both transient and permanent faults. For example, two processors can be used to execute the tasks, and at the end of the tasks' execution, their results can be compared. If their results do not match, then a transient fault has occurred. Using the slack reserved when the tasks were scheduled, the faulty task is re-executed on both processors, and the results compared again. If the fault is transient, then the re-execution will allow a correct result to be generated before the task's deadline. During the comparison, if one of the processors does not generate a result, then a permanent fault has occurred and the results from the non-faulty processor can be used. A spare processor can be used to replace the faulty processor, if desired.

A new algorithm for fault-tolerant scheduling on multiprocessor systems is proposed by Ghosh et al in [48]. The algorithm guarantees the completion of a scheduled task before its deadline in the presence of processor failures. It schedules several backup tasks overlapping one another and dynamically deallocates the backups as soon as the original tasks complete executions, thus increasing the utilization of processors.

As an abstraction we can assume that preemption and fault detection costs are negligible. In practice, however, these costs are not negligible. The fault detection cost for the fault tolerant scheme will include the communication cost between the processors and the comparison time. The comparison time is constant and has to be significantly lower than the task computation times in order to be cost effective. The communication time is dependent on how many faults have occurred in the communication protocol layers. This additional time can be added to the computation time of each task a priori, so that fault detection times are taken into account.

### **3.4.1 Fault tolerance in multiprocessor real-time systems**

The problem of scheduling periodic tasks in multiprocessor systems is considered in [4], [7], and [49]. It was shown that RMA is not optimum for scheduling tasks in

multiprocessor systems. No algorithms were shown to be optimum for scheduling periodic tasks in multiprocessor systems.

Joseph and Pandya [10] demonstrated the Completion Time Test (CTT) in order to verify the schedulability of a fixed set of periodic tasks in a processor. RMA was generalized for multiprocessor systems by Dhall and Liu [1], who proposed the heuristic Rate-Monotonic First-Fit (RMFF), where first, the tasks are designated to the processors as per priority designated by RMA and second, all tasks designated to the same processor are scheduled by RMA.

Pandya e Malek [14] showed that in a process for executing again a task that has failed in a fault-free instance, as proposed in [11], the priorities should be maintained by RMA and, under those conditions, none of the tasks will lose its deadline, even in the presence of faults, if the processor utilization factor is not higher than 0.5.

Bertossi and Mancine [15] proposed the Fault-Tolerant Rate-Monotonic First-Fit (FTRMFF) to provide fault-tolerance to a HRTS, by using a multiprocessor system where each task has a passive or active copy in another processor and the whole set of tasks is prioritized by RMFF, supporting fail-stop of one or more processors.

Oliveira and Fernandes[11],[12] proposed the Deadline, where a processor and the RMA are used to designate the priorities and to schedule the tasks, providing fault-tolerance to the system with the inclusion of only one hardware redundancy, the watchdog timer. An improvement to the Deadline was further proposed in order to apply it to multiprocessor systems, resulting in a system capable of supporting transient faults and processors fail-stop.

#### ***3.4.1.1 The Deadline – a multiprocessor real-time kernel with fault tolerance***

The Deadline, as it was proposed, is a multitask, fault-tolerant core system for applications in HRTS with only one processor. The objective of the system is to expand directly the availability, reliability, maintainability, testability and indirectly the safety, performability and dependability of the introduced tasks and of the system as a whole. A complete description of Deadline can be found in [11] and summarized descriptions in [12].

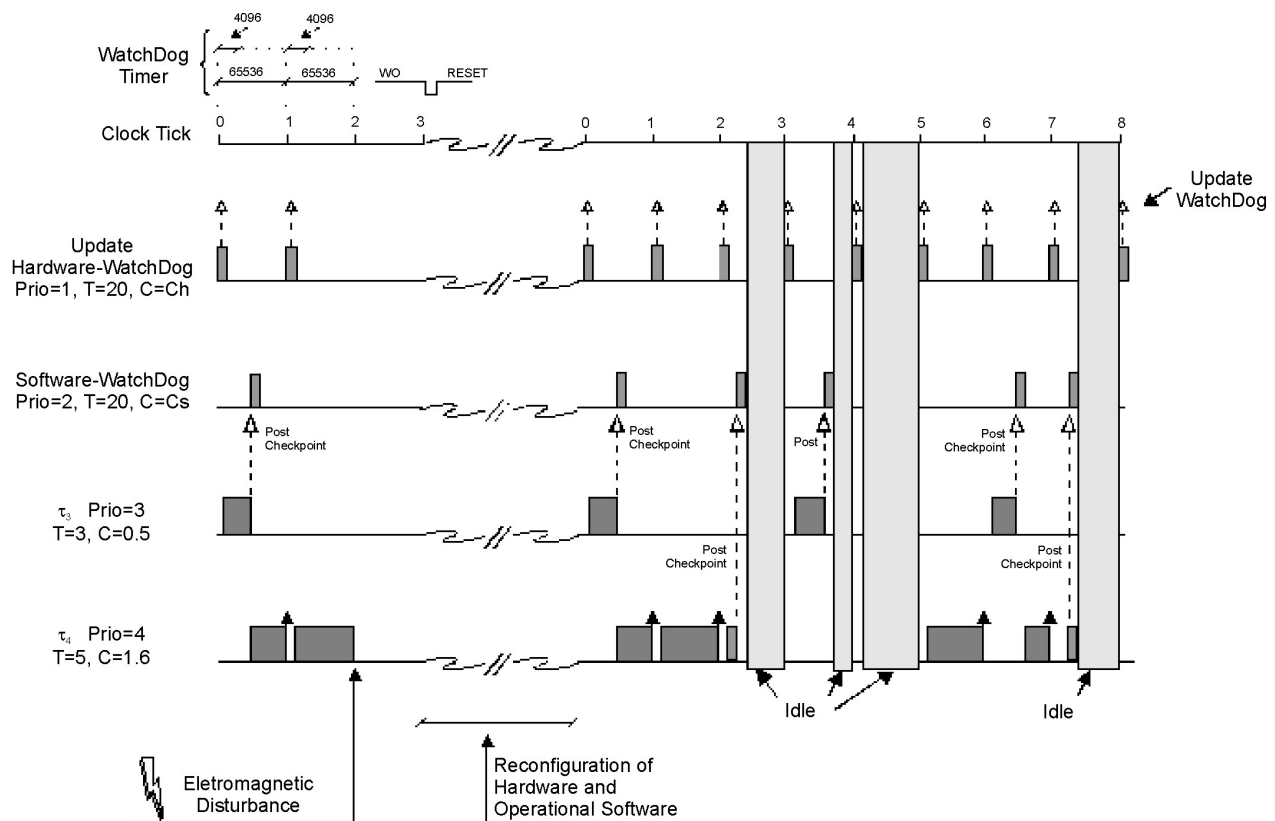


FIGURE 3.1: PROCESSOR FAULT AND RECOVERY IN THE DEADLINE.

Figure 3.1 [11] shows an example on how watchdog timer detects a fault and triggers the recovery process after a loss of consistency in the operational system layer or in a processor fault occurs. On that example, after occurrence of a more severe electromagnetic disturbance (example: lightning affecting the local electric installation) at the clock time= 2, all the tasks of the system, including the ones that implement the fault-tolerance attributes in the Deadline, are not executed.

The processor is out of routines, which should be in process or even in an undefined status, under a condition of abnormal hardware created by the electromagnetic disturbance. In that case, after two consecutive clock ticks, without the watchdog timer receiving signaling from the operating system that everything is fine, an electrical signal is emitted via its WO physical port. That output signal is connected to the RESET entry of the processor. It reinitializes the whole system, placing the CPU in valid status. Thus, the CPU starts the operating system layer, which finally introduces again all the tasks of the target application.

One Deadline restriction is that it works with only one processor, what limits the specific fault-tolerance of the processor exclusively to transient faults and still leads to a restriction of the processor recovery time, operational system and application layer. To eliminate this restriction, Oliveira and Fernandes [11], [12] showed also how the Deadline

could be complemented by FTRMFF - Fault-Tolerant Rate-Monotonic First-Fit [15] - in order to tolerate faults in HRTS. In the FTRMFF, proposed by Bertossi and Mancini [15], faults are implemented in a multiprocessor system by using a technique where each task scheduled in a processor has a passive or active copy in another processor. The active copy is always executed while the passive one is only executed in case of occurrence of faults in the primary task processor. With RMFF, all tasks and their active copies can be assigned to processors and scheduled with no loss of deadlines. Passive copies are chosen whenever possible, the active ones being used only in cases where the schedule time of primary task is near its deadline, not being possible the inclusion of synchronization time required between the primary copy and the passive copy. Passive copies can share the same interval of time in a same processor since they do not need to be executed. They are always utilized in detriment of active copies, thus reducing the total number of processors required for supporting the HRTS.

As example of fault recovery, consider the set of tasks shown in Figure 3.2 [15], which is in a fault-free status. In processor  $P_1$ , we verify the existence of 3 primary tasks being executed ( $\tau_1, \tau_2$  e  $\tau_4$ ). In processor  $P_3$ , there are two tasks being executed: one primary ( $\tau_3$ ) and one active copy ( $\beta_4$ ). The processor  $P_2$  is available for executing the passive copies ( $\beta_1, \beta_2$  e  $\beta_3$ ), in case a failure occurs with  $P_1$  or  $P_3$ .

Figure 3.3 [15] shows the occurrence of a fail-stop failure in processor  $P_1$  at the period of time 0. Processor  $P_2$  detects that fault at the period of time 2. At that moment, passive copies of primary tasks that were executing in  $P_1$ , begin to be executed in  $P_2$ . On that example, it can be observed that this proposal complements the Deadline, since in case of transient fault, as shown in Figure 3.1, tasks are not executed during several periods of time until the processor, the operational system and the applications are again under execution in a fault-free instance. With that proposal, tasks begin to execute in processor  $P_2$ , thus not losing their deadlines, what may occur in a system with only one processor.

In figure 3.4 [15],  $P_s$  represents an available backup processor that starts the operation in order to replace the processor lost due to permanent fault  $P_1$ . As the faults model proposed by [15] only permits fail-stop, we can verify that the Deadline complements this proposal since it permits that transients faults in the processor can be also tolerated. The advantage is to recovery the processor, as shown in Figure 3.5, where the processor  $P_1$  is recovered from a transient fault at period of time  $t = 12$ , returning to be available for processing.

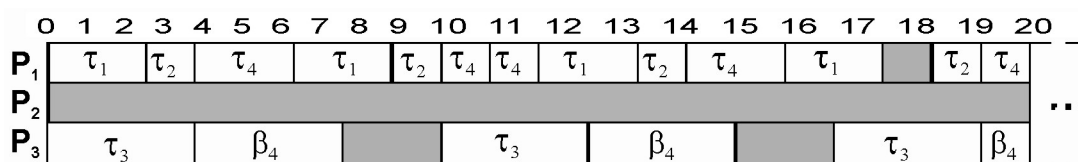
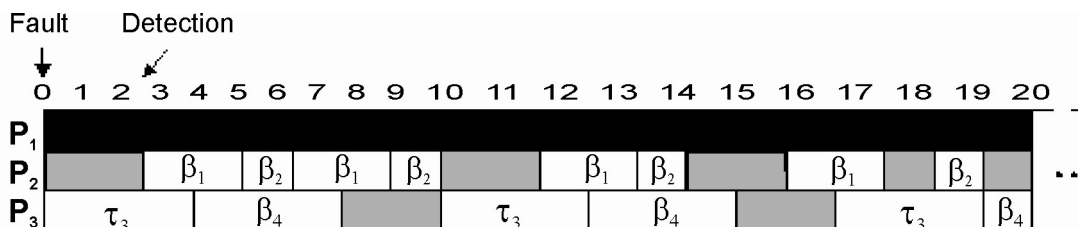
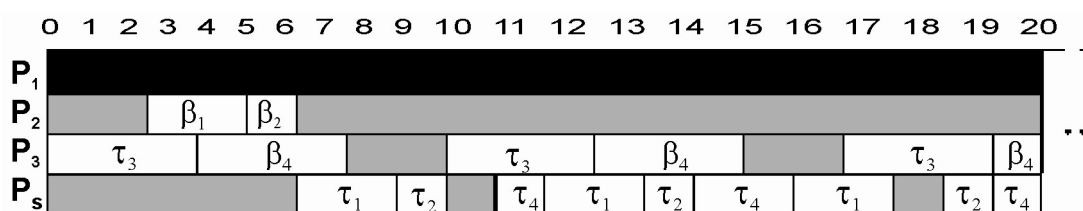
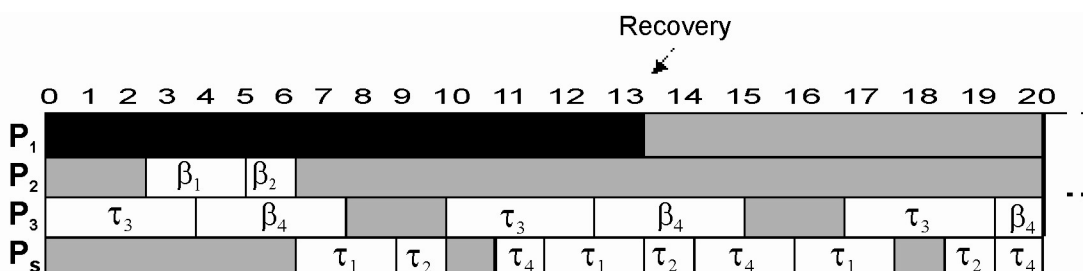


FIGURE 3.2: FAULT-FREE SYSTEM STATUS

FIGURE 3.3: FAULT RECOVERY FROM  $P_1$  EXECUTED BY  $P_2$ FIGURE 3.4: PROCESSOR  $P_1$  REPLACED BY  $P_s$ .FIGURE 3.5: DEADLINE RECOVERS  $P_1$ .

The Fault-Tolerant Rate-Monotonic First-Fit concept, first introduced by Bertossi and Mancini in [15], complements the Deadline and represents an evolution, since it covers a higher number of types of faults, makes processors with transient faults available again for processing and avoids interruption of tasks when a transient or permanent fault occurs in a certain processor.



# Chapter 4

## Real-Time Fault-Tolerant Communication Protocols

### 4.1 Overview

One of the essential services provided by real-time fault-tolerant distributed architecture is communication of information from one distributed component to another, so a communication bus is one of its principal components, and the protocols used for control and communication on the bus are among its principal mechanisms. In reality, these architectures are the safety-critical core of the applications built above them, and the choice of services to provide to those applications, and the mechanisms of their implementation, are issues of major importance in the construction and certification of safety critical embedded systems [58][59].

Digital communication busses are frequently used to connect mission critical components. The failure of such a bus is not tolerable; therefore redundant busses are often used in safety-critical environments to handle device faults. Besides fault tolerance, many applications require real-time guarantees such as bounded message latency. There are various protocols for such purposes, with different complexities, which are used by the avionics industry, such as Airbus and Boeing, and automobile industry, such BMW and Audi. Boeing is using the ARINC 629 bus for the new fly-by-wire aircraft, such as the Boeing 777 (<http://www.boeing.com>). Other protocols for this domain are the Time Triggered Architecture (TTA) protocols [25] (adopted by Audi for automobile applications), FlexRay [27] (which is being developed by a consortium of BMW, DaimlerChrysler, Motorola, and Philips), Controller Area Network CAN [29], TTCAN [55][56][57], Honeywell SAFEbus [54] (the backplane data bus used in the Boeing 777 Airplane Information Management System). Some of the busses considered here are primarily time triggered, which means that all activities involving the bus, and often those involving components attached to the bus, are

driven by the passage of time. In event-triggered busses, the activities are driven by the occurrence of events. A time-triggered system interacts with the world according to an internal schedule, whereas an event-triggered system responds to stimuli. The time-triggered and event-triggered approaches to systems design find favor in different application areas, and each has strong advocates [59].

## 4.2 Medium access protocol classes

More than one hundred real-time communication protocols have been implemented in the last twenty years [60]. The main concern of all these protocols is the same: how to assign the bandwidth of the single communication medium for short intervals of time exclusively to a node of the distributed system such that certain system properties can be maintained. One important issue in a real-time communication protocol is thus the medium access control [60]. The known protocols can be assigned to one of six protocol classes on the basis of the medium access control: carrier sense multiple access with collision detection (CSMA/CD), carrier sense multiple access with collision avoidance (CSMA/CA), token control, mini-slotting, central control, and time division multiple access (TDMA). The medium access strategy of a communication protocol determines which node is allowed access to the bus at a particular point in time.

Carrier Sense Multiple Access Collision Detection Protocols (CSMA/CD classical example ETHERNET) are distributed medium access protocols that do not require any central locus of control. Although it is not a real-time protocol, a good example for a protocol from this class that is targeted for real-time systems in building automation is the LON Protocol from Echelon [65].

Carrier Sense Multiple Access Collision Avoidance Protocols (CSMA/CA) are distributed medium access protocols that avoid the occurrence of collisions, e.g., by bit arbitration. A good example of a CSMA/CA protocol is the CAN Protocol developed by Bosch targeted for automotive real-time applications [62].

In a token bus system the right to transmit is contained in a special control message, the token message. Whoever is in possession of this token message is allowed to transmit. A serious error in any token system is the loss of the token, e.g., if the station that possesses the token fails. An example of a token bus protocol proposed for real-time systems is the Profibus[66] .

Minislotting is a time-controlled medium access strategy, where the time is partitioned into a sequence of minislots; each one being the length of the propagation delay of the channel. Every node has to wait a different number of minislots before it is allowed to transmit. A good example of a protocol based on mini-slotting is the ARINC 629 used by the aircraft industry for real-time communication [61].

A central master Protocol relies on a central master to control the access to the bus. In case the central master node fails, another node can take over the role of the central master (multi-master systems). A good example for a central master protocol is the FIP protocol [63].

TDMA is a distributed static medium access strategy where the right to transmit is controlled by the progression of real-time. This requires that a (fault-tolerant) global time base be available at all nodes. An example of a TDMA protocol proposed for real-time applications is the Time-Triggered Protocol TTP [64].

### 4.3 Model for Hard Real-Time communication

In a distributed hard real-time system, communications between tasks on different processors must occur in bounded time. The inevitable communication delay is composed of both the delay in transmitting a message on the communications media, and also the delay in delivering the data to the destination task.

In [71] Tindell derives schedulability analysis bounding the media access delay and the delivery delay. Tindell considered two access protocols: a simple timed token passing approach, and a real-time priority broadcast bus. Some of concepts that he described we are going to introduce in this section.

A hard real-time system is often composed from a number of periodic and sporadic tasks, which communicate their results by passing messages; in a distributed system these messages are sent between processors across a communications device. In order to guarantee that the timing requirements of all tasks are met, the communications delay between a sending task queuing a message, and a receiving task being able to access that message, must be bounded. This total delay is termed the end-to-end communications delay. Tindell defines the end-to-end communications delay to be made up of four major components:

- **The generation delay:** the time taken for the application task to generate and queue the message.

- **The queuing delay:** the time taken by the message to gain access to the communications device after being queued
- **The transmission delay:** the time taken by the message to be transmitted on the communications device
- **The delivery delay:** the time taken to process the message at the destination processor before finally delivering it to the destination task

The generation delay is the worst-case time taken between the arrival of the sender task and the queuing of the message. This represents some element of application processing to generate the contents of the message, and the time taken to queue the message. The queuing delay is the time the message spends waiting to be removed from the queue by the communications device. With a point-to-point communication link, the message must contend with other messages sent from the same processor; with a shared communications link, the message must also contend with messages sent from other processors. The transmission delay is the time taken for the message to be sent once it has been removed from the queue. The delivery delay is the amount of time taken to process the incoming data and deliver it to destination tasks. The Tindell work [71] includes such functions as decoding packet headers, re-assembling multi-packet messages, copying message data between buffers, and notifying the dispatcher of the arrival of a message. This latter function is important, since the destination task may be blocked awaiting the arrival of the message. In practice the delivery delay can form a significant part of the end-to-end communications delay.

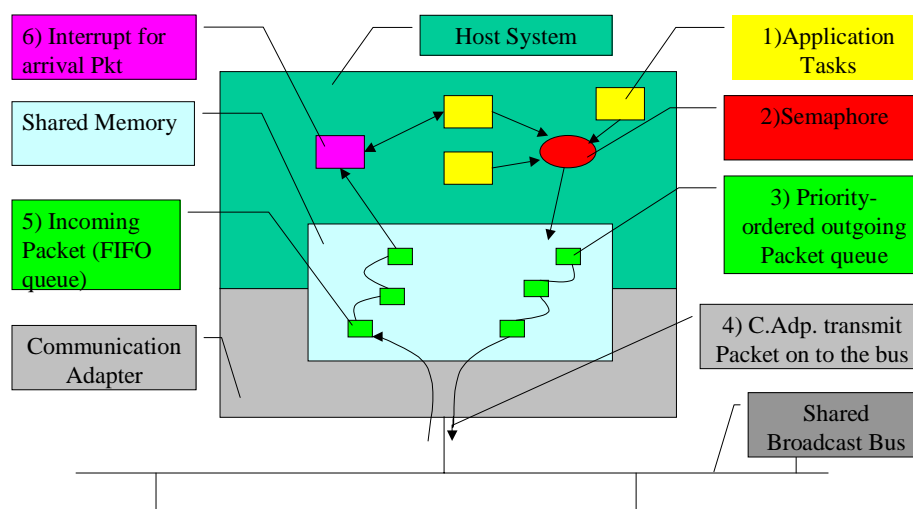


FIGURE 4.1: SHARED BROADCAST BUS

Figure 4.1 shows one typical arrangement for the transmission and reception of packets in a shared broadcast bus. One complete end-to-end sequence can be seen following the numbers steps in the figure. For this typical architecture the packet queue is stored in shared memory. Shared memory can be accessed by normal read and write instruction in application software, except that extra 'wait states' are incurred. The communications adapter is a single reader of packets. Writes to the queue are atomic - a packet can be written to a spare slot in the buffer; when complete it can be atomically inserted into the queue by simple pointer manipulation. According to project constraints, a node will not insert packets in the bus in a higher load than the bus throughput. Thus, one can assume that there is always sufficient space to store these packets. A protected object (semaphore) ensures concurrency control between application tasks queuing packets. The communications adapter removes the packet at the head of the queue (by simple pointer manipulation) and transmits it on to the bus. In the worst-case there is time  $p$  between subsequent packet transmissions. For each packet removal the communications adapter may be blocked in the worst case for a few processor cycles while the host processor completes pointer manipulation (and vice versa). This time is included in  $p$ . Incoming packets are stored in a FIFO queue. An interrupt is raised for each arrival of a packet, requiring computation time  $C_{\text{packet}}$  to process (the costs of copying the packet from the shared network buffer, stripping headers, etc.). The interrupt handler on the host processor removes the packet from the buffer; when all the packets of a message have arrived the interrupt handler releases the task blocked awaiting the message. This approach avoids the need for direct memory access (DMA) to transfer the data.

## 4.4 Time-Triggered Protocol TTP/C

The Time Triggered Architecture (TTA) was developed by Hermann Kopetz and colleagues at the Technical University of Vienna [64]. Commercial development of the architecture is undertaken by TTTech and it is being deployed for safety-critical applications in cars by Audi and Volkswagen, and for flight-critical functions in aircraft and aircraft engines by Honeywell.

In a Time-Triggered Architecture the communication system decides autonomously and according to a static schedule when to transmit a message. Every station contains its own control data that specifies at which instant a message must be transmitted by the controller. A

TTP/C network consists of a set of electronic modules that are connected by two replicated channels as shown in Figure 4.2 [25].

Time-triggered architectures are driven by the progression of the global time. All tasks and communication action are periodic, and external state variable are sampled at predefined points in time.

The Communication Network Interfaces implement the TTP/C protocol [26], providing clock synchronization, and message sequencing and transmission functions. The interconnect bus is duplicated and each controller drives both of them through partially independent bus guardians. The bus guardian is an autonomous subsystem of the controller that protects the communication channels from temporal transmission failures. Physically, the bus guardian may reside on the same silicon die as the TTP/C protocol controller; it can also be implemented as an independent device. In any case, a TTP/C controller must have a bus guardian to achieve fail-silence in the temporal domain [26]. The guardians, which have independent clocks, therefore rely on their controllers for a "start of frame" signal. This compromises their independence somewhat (they also share the power supply and some other resources with their controllers), so forthcoming implementations of TTA use a star interconnect. In this case, the guardian functionality is implemented in the central hub, which is fully independent of the controllers: the hubs and controllers comprise separate fault containment units in this implementation. Hubs are duplicated for fault tolerance and located apart to withstand spatial proximity faults.

TTA employs algorithms for group membership and clique avoidance [67], what enables its clock synchronization algorithm to tolerate multiple faults, by reconfiguring to exclude faulty members, and combine with its use of checksums to provide a form of interactively consistent message broadcasts [58].

The TTP/C protocol has been designed to tolerate any single internal physical fault in any one its constituent parts without an impact on the operation of a properly configured cluster. As long as an external fault impacts only a single TTP/C subsystem, the TTP/C system will tolerate such a fault.

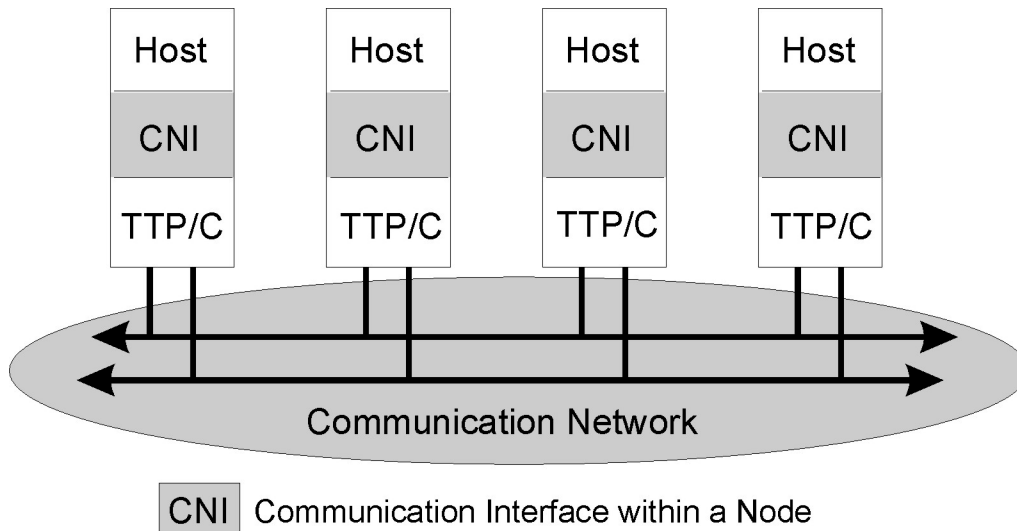


FIGURE 4.2: TTP/C ARCHITECTURE

## 4.5 FlexRay Protocol

FlexRay is a new real-time protocol, not yet released to the public, being developed by a consortium of companies (BMW, DaimlerChrysler, Motorola, and Philips). Although used primarily for automotive applications, it is representative of state-of-the-art safety critical real-time protocols.

FlexRay aims to be more flexible than TTP/C, introducing some dynamism through the combination of time-triggered and event-triggered operation. FlexRay partitions each time cycle into a "static" time-triggered portion, and a "dynamic" event-triggered portion. The division between the two portions is set at design time and loaded into the controllers and bus guardians. Communication during the event-driven portion of the cycle uses the Byteflight protocol [28]. Unlike TTA, FlexRay does not install the full schedule for the time-triggered portion in each controller. Each time-triggered portion of the cycle in each controller is divided into a number of slots of fixed size and each controller and its bus guardians are informed only of those slots allocated to their transmissions. Controllers learn the full schedule only when the bus starts up. Each node includes its identity in the messages that it sends; during startup, nodes use these identifiers to label their input buffers as the schedule reveals itself [58]. FlexRay does not use a membership algorithm to exclude faulty nodes. FlexRay provides no services to its applications beyond best-efforts message delivery; in particular, it does not provide interactively consistent message broadcasts. This means that all

mechanisms for fault-tolerant applications must be provided by the applications programs themselves [58].

FlexRay is more flexible because of its mixture of time-triggered and event-triggered operation, and potentially important because of the industrial clout of its developers [58]. There is a conflict between Safety and Flexibility. FlexRay puts available the option of to work between these two concepts that TTP/C does not. The designer can take care of safety and still have enough flexibility.

## 4.6 Controller Area Network Protocol CAN

The controller area network (CAN) uses a serial multimaster communication protocol that efficiently supports distributed real-time control with a very high level of data integrity, and communication speeds of up to 1 Mbps. The CAN bus is ideal for applications operating in noisy and harsh environments, such as in the automotive and other industrial fields that require reliable communication.

Prioritized messages of up to eight bytes in data length can be sent on a multimaster serial bus using an arbitration protocol and an error-detection mechanism for a high level of data integrity.

The CAN protocol supports four different frame types for communication:

- **Data frames** that carry data from a transmitter node to receiver node(s)
- **Remote frames** that are transmitted by a node to request the transmission of a data frame with the same identifier
- **Error frames** that are transmitted by any node on a bus error detection
- **Overload frames** that provide an extra delay between the preceding and the succeeding data frames or remote frames

In addition, CAN specification version 2.0B defines two different formats that differ in the length of the identifier field: standard frames with an 11-bit identifier and extended frames with a 29-bit identifier.

CAN standard data frames contain from 44 to 108 bits, and CAN extended data frames contain 64 to 128 bits. Furthermore, up to 23 stuff bits can be inserted in a standard data frame and up to 28 stuff bits in an extended data frame, depending on the data-stream coding. The



overall maximum data frame length is 131 bits for a standard frame and 156 bits for an extended frame. Figure 4.3 [36] shows the CAN Data Frame, bit fields within the data frame identify:

- Start of the frame
- Arbitration field containing the identifier and the type of message being sent
- Control field containing the number of data
- Up to 8 bytes of data
- Cyclic redundancy check (CAC)
- Acknowledgment End-of-frame bits

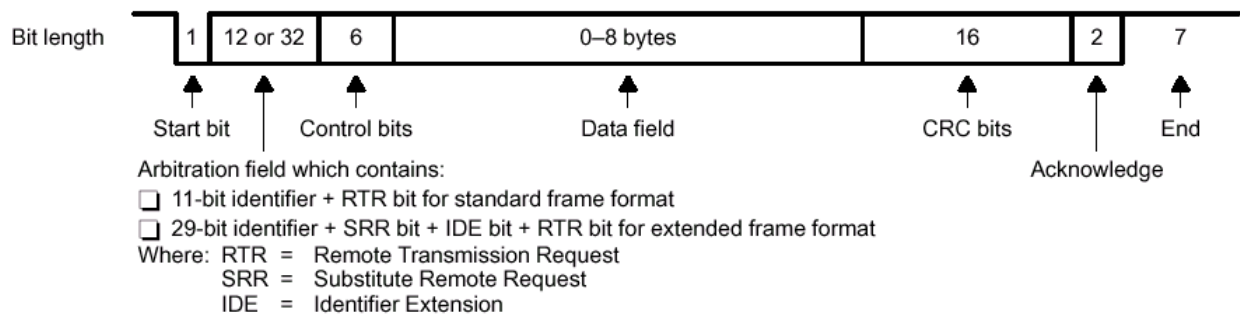


FIGURE 4.3: CAN DATA FRAME

A perceived problem with CAN for use in distributed real-time control applications is its inability to bound the response times of messages. While CAN is very good at transmitting the most urgent data, it is unable to provide guarantees that deadlines are met for less urgent data [60][64][68], once the most urgent data may flood the bus avoiding the transmission of the less urgent data. As presented by Kopetz and Griinsteidl [64], the CAN protocol may have a unbounded response time for an arbitrary low priority packet, what does not happen with TTP/C; nevertheless, the dynamic scheduling algorithm used by CAN is virtually identical to scheduling algorithms commonly used in real-time systems to schedule computation on processors [70]. In fact, the analysis of the timing behaviour of such systems can be applied almost without change to the problem of determining the worst-case latency of a given message queued for transmission on CAN.

In [69], Tindell, Burns and Wellings developed a CAN analysis based on RMA showing how to find the response time for messages being transmitted in a CAN bus. Since

CAN is primarily a priority-based bus, much of the analysis for systems where activities are dispatched according to fixed priorities can be applied directly.

## 4.7 TTCAN - Time Triggered Communication on CAN

A new development in CAN technology is the TTCAN protocol [72], a higher-layer protocol above the unchanged standard CAN protocol that synchronizes the communication schedules of all CAN nodes in a network and that provides a global system time. When the nodes are synchronized, any message can be transmitted at a specific time slot, without competing with other messages for the bus. Thus the loss of arbitration is avoided, the latency time becomes predictable.

The goal of time triggered operation on CAN is to avoid the usual latency jitters and to guarantee a deterministic communication pattern on the bus. One advantage of TTCAN compared to classic scheduled systems is the possibility to transmit event-triggered messages in certain “arbitrating” time windows as well.

TTCAN protocol specifies a time-slot based communication mechanism avoiding the transmission collisions commonly found in standard CAN networks [57]. In TTCAN, all the message instances are transmitted only on previously allocated time-slots, just like the TTP/C protocol. Before transmitting over a CAN bus, it is necessary to create a valid scheduling table that specifies how the time is discretized into time-slots and how the messages are allocated into those time-slots. The communication is based on the periodic transmission of a reference message by a time master. Based on this time the different messages are assigned to time windows within a basic cycle. However, it is necessary to use some quality criterion to select among distinct scheduling tables that may be built for the same message set.

TTCAN can be implemented using a regular CAN microcontroller, although it is also necessary an extra hardware for the time-triggered portion of the protocol. Synchronizing nodes is not a simple task, and brings a new complexity to the CAN bus. The cyclic message transfer of TTCAN used in the synchronization process may be implemented in software but, depending on the CAN bit rate and on the number of messages in the system matrix, it may result in a high CPU load. A hardware approach is usually chosen, but it introduces an extra cost and also increases the system complexity. The development of proprietary modules implementing TTCAN helps optimizing the system structure.

# Chapter 5

## The Thesis Goals and Approaches

### 5.1 Overview

Whenever a fault-tolerant system is designed, a redundancy type must be incorporated into the equivalent system, which does not incorporate fault tolerance requisites. Since time is the central resource of any fault-tolerant hard real-time system, task schedulability and time redundancy become the basic tools to guarantee fault tolerance requisites for a given hard real-time system. With processor costs dropping off and the emergent development of real-time communication protocols, such as CAN [29], TTP/C [26], TTP/A [25] and FlexRay [27], a great need has consequently arisen to research multiprocessor hard real-time systems and fault tolerance, where the task schedulability needs to be guaranteed for a certain specified fault model.

### 5.2 Summary of Goals

The main goal of this work is to guarantee fault-tolerance requisites for multiprocessor hard real-time systems. The resources used to optimize these fault-tolerance requisites are task schedulability and time redundancy.

This work presents techniques to enhance the fault-tolerance capability of hard real-time systems by incorporating time redundancy, processor redundancy and fault tolerant real-time communication protocols.

Time redundancy is essential in ultrareliable real-time systems where correlated faults must be tolerated. It can also be used to detect and tolerate transient faults, which comprise the majority of faults in computing systems. This work shows how time redundancy can be used in conjunction with hardware and software redundancy to tolerate faults in hard real-time systems.

It was uncertain whether this work would research a schedulability test, a utilization boundary or a set of conditions. Working with this approach guaranteed that all tasks in the system would satisfy their timing constraints even in the presence of faults.

This work, aside from theoretical concerns, researched the implementation of a DSP-based multiprocessor system interconnected by a CAN bus. The research considered the overhead introduced by the real-time fault tolerant communication protocol. For those cases in which the implementation was not appropriate, simulation tests were done.

## **5.3 Goals and Approaches**

### **5.3.1 Enhance the fault tolerance capabilities of multiprocessor hard real-time systems**

The basic approach that we are going to use in this work for providing fault tolerance through scheduling in multiprocessor hard real-time systems is to add slack or backup slots into the schedule. If a fault is detected by the operating system during the execution of a task, that task is either re-executed or a backup for that task is activated as part of the fault recovery. For the uniprocessor models, a certain amount of slack is added to the schedule. If a task does not generate correct results due to transient faults, then this slack is used to re-execute it.

For multiprocessor systems, permanent processor faults must also be tolerated. Therefore, if a task is scheduled on one processor, the slack needed for re-execution is scheduled on a different processor. The scheduling algorithm could ensure that the task can complete its re-execution using the slack on a different processor before its deadline.

The general approach of scheduling two copies of each task in the system is called the primary/backup approach, and the two copies are called the primary and the backup. The backup is executed only if the primary fails. Note that this scheme of adding time redundancy also allows different versions of the task to be executed as primary and backup, thus facilitating the provision of software fault tolerance. In Chapter 6, a case study consisting of a Uninterruptible Power Supply (UPS) is presented. In this example, there are two different processors executing the Inverter and the Static Switch algorithms. The schemes of running a backup version of these processors and algorithms was used. When a fault is detected, a recovery scheme is needed to determine the steps to tolerate the fault. The recovery scheme to

be used in this work is simply the re-execution of faulty tasks. The task is re-executed on the same or other processor to tolerate transient faults, and on a different processor to tolerate permanent faults.

There are four classifications relevant to this work: preemptive versus non-preemptive tasks, periodic versus aperiodic tasks, transient versus permanent processor faults, and uniprocessor versus multiprocessors systems. A fault-tolerant real-time scheduling algorithm might have to deal with any combination of tasks, fault models and processor sets. This work will focus on models for multiprocessor, transient and permanent faults, and periodic preemptive tasks. Figure 5.1 shows the classification of real-time scheduling algorithms based on the task, fault models and processor set.

## Real-time FT Task Scheduling Problems

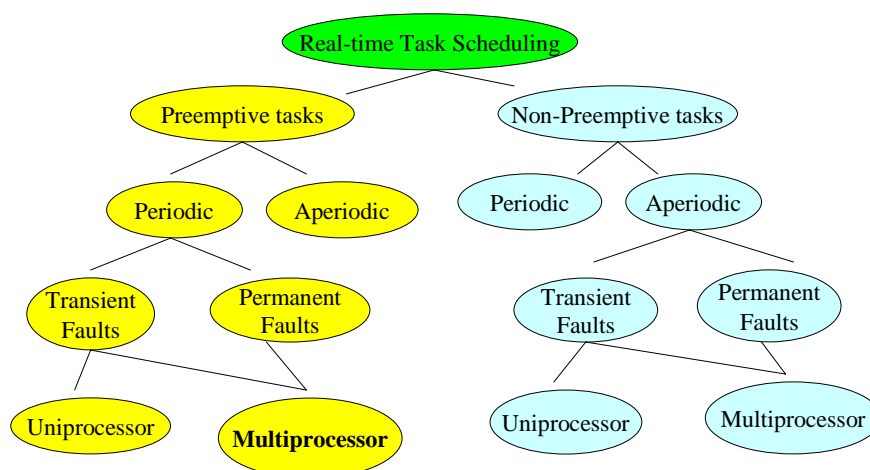


FIGURE 5.1: SCHEDULING PROBLEMS

### 5.3.2 Guarantees required for fault-tolerant execution in real-time systems

Given a task and fault model, it is possible to prove properties about the fault tolerance capabilities of the system. For example, it may be possible to prove that if certain assumptions or conditions hold, then one fault can be tolerated within a specific interval of time. When a new task is being considered for addition into the system, a set of conditions is tested to check whether the fault tolerance guarantees can be provided. If the conditions are met, then the task is accepted, otherwise it is rejected. Thus, these conditions constitute the schedulability tests for new tasks.

In the preemptive task model, the total task utilization is compared with fault-tolerant utilization bounds, as described in Section 3.3. If the total task utilization is lower than the bound, then the task set is schedulable. The number of transient faults tolerated per task is a function of the amount of slack reserved. The type and frequency of faults that can be tolerated depends on the system and fault model. Figure 5.2 shows the classification of permanent, transient and correlated faults.

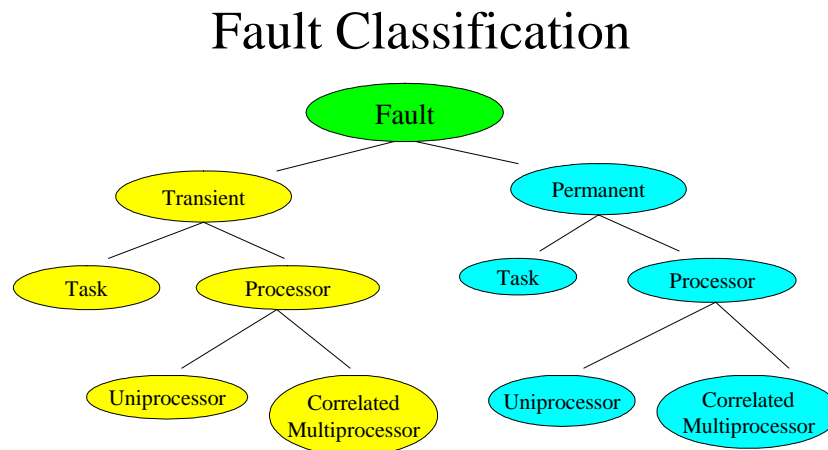


FIGURE 5.2: FAULT CLASSIFICATION

### 5.3.3 Study the tradeoff between fault tolerance capability and resource utilization of hard real-time systems

The capability of a system to tolerate faults is based on the number and frequency of faults it can tolerate. If a system includes more redundancy to tolerate a larger number of faults, its fault tolerance capability increases. However, the fault tolerance capability is achieved at the cost of system utilization, which is the percentage of system resources used for actual operations. If there is a large amount of redundancy in the system, the percentage of resources being used for actual computing purposes is small, and thus the system utilization is low. The goal of a hard real-time system is to be as fault-tolerant as possible, while providing high utilization and thus being cost effective.

The utilization of a preemptive system can be measured by the total utilization of the scheduled task set because that determines the percentage of available processing time used for task execution.

To determine how the fault-tolerant schemes perform in terms of system utilization, they can be compared with an equivalent scheme where no fault tolerance is provided. Such a

scheme is called the No-FT scheme. For example, in the preemptive task model, the rate monotonic scheme introduced by Liu and Layland [5] is used as the No-FT scheme.

To measure the fault tolerance capability of the FT scheme, faults are injected into the system, and the number of lost tasks is measured. A task that cannot meet its deadline because of a fault is called a lost task. This happens only if there are more faults in the system than the provisions made by the fault tolerance approach, i.e., if the fault assumptions are violated. To avoid losing tasks because of faults, the time redundancy of the system must be increased. This increase lowers the total number of tasks scheduled in the system.

### 5.3.4 Use techniques to improve system utilization

To make sure that the backup slots or backup capacity does not cause a large drop in processor utilization, some techniques, like overloading and deallocation can be used. Overloading is the provision of a certain amount of time redundancy for a set of tasks which is less than that required to re-execute all the tasks in that set. This can be done if it is assumed that faults will not occur during the execution of each task in the set, and thus only a subset of the tasks will need to be re-executed. Figure 5.3 [17] shows one example of overloading. The processor  $P_2$  can run the Backup tasks  $Bk_1$  or  $Bk_3$ .

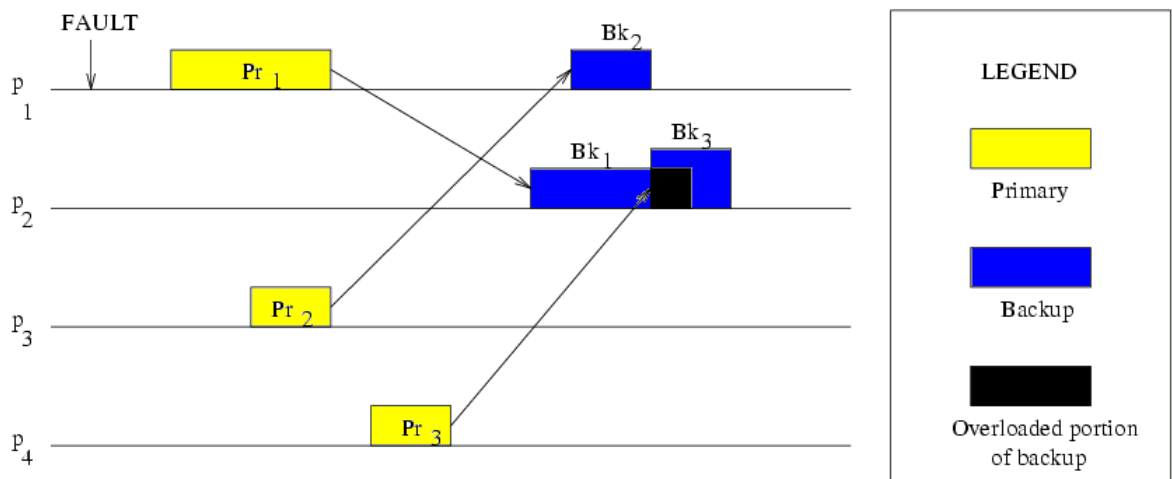


FIGURE 5.3 OVERLOADING TECHNIQUES

In a preemptive system, overloading is the provision of an amount of slack smaller than that required to re-execute all the task instances scheduled at any given time. Overloading effectiveness increases by overlapping more backups or by reducing the amount of slack available for a set of tasks. The system utilization increases if more overloading is achieved, at

the cost of reducing the fault tolerance capability. Thus, overloading can be increased or decreased according to the frequency of faults in the system. If faults occur less frequently, more overloading should be used, and vice versa.

Another technique that can be used is deallocation, which is the reallocation of resources reserved for backup tasks when the corresponding primaries complete successfully. Deallocation is not useful if all tasks are periodic since the deallocated backup of a single task instance cannot be used to guarantee all instances of another periodic task. However, a deallocated backup can be used to guarantee new aperiodic tasks.

### **5.3.5 Define resiliency of the systems**

A recovery scheme determines the time taken by the system to recover from a fault. Resiliency is the ability of a system to tolerate a second fault after recovering from the first one.

The resiliency can be measured in terms of the minimum separation between two successive faults that can be tolerated. The smaller this separation is, the higher the resiliency of the system. In static systems, the resiliency of the system can be measured exactly, because the static schedule determines how far apart two successive faults can be tolerated. On the other hand, in dynamic systems, the average resiliency of the system can be determined by fault injection and simulations.

Since we are working with a preemptive model, the schedulability tests can be executed statically, and thus the resiliency can be measured accurately.

### **5.3.6 Estimate the overhead introduced by the real-time fault-tolerant communication protocol**

This work, aside from theoretical concerns, researched the implementation of a DSP-based multiprocessor system interconnected by a CAN bus. Figure 5.4 shows the real implementation of an architecture developed based on Texas Instruments TMS320LF2407A microcontroller. This implementation is sponsored by Engetron Ltda and has been used in the current work and also on other projects developed at the Electrical Department of the Federal University of Minas Gerais. Other microprocessors than DSPs can also be used interconnected by a CAN bus.



The TMS320 family consists of fixed-point, floating-point, multiprocessor digital signal processors, and fixed-point DSP controllers. TMS320 DSPs have an architecture designed specifically for real-time signal processing. The '240x series of DSP controllers combines this real-time processing capability with controller peripherals to create solutions for control system applications.

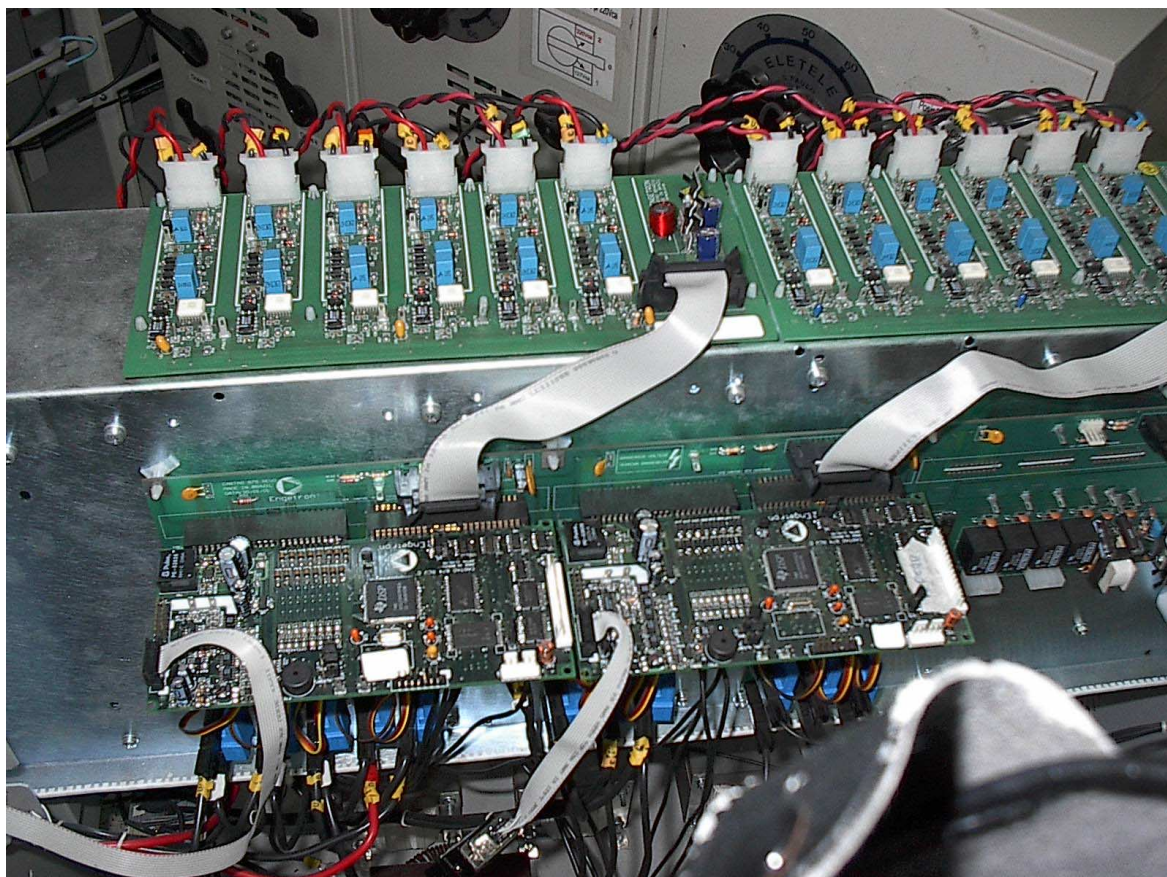


FIGURE 5.4: HARDWARE IMPLEMENTATION

In 1982, Texas Instruments introduced the TMS32010, the first fixed-point DSP in the TMS320 family. Today, the TMS320 family consists of these generations: 'C1 x, 'C2x, 'C20x, 'C24x, 'C5x, 'C54x, and 'C6x fixed-point DSPs; 'C3x and 'C4x floating-point DSPs; and 'C8x multiprocessor DSPs. The '240x devices are considered part of the '24x generation of fixed-point DSPs, and members of the 'C2000 platform.

Devices within a generation of a TMS320 platform have the same CPU structure but different on-chip memory and peripheral configurations. Spin-off devices use new combinations of on-chip memory and peripherals to satisfy a wide range of needs in the worldwide electronics market. By integrating memory and peripherals onto a single chip,

TMS320 devices reduce system costs and save circuit board space. Figure 5.5 shows the block diagram of TMS2407 used in this work.

## TMS320LF2407 Block Diagram

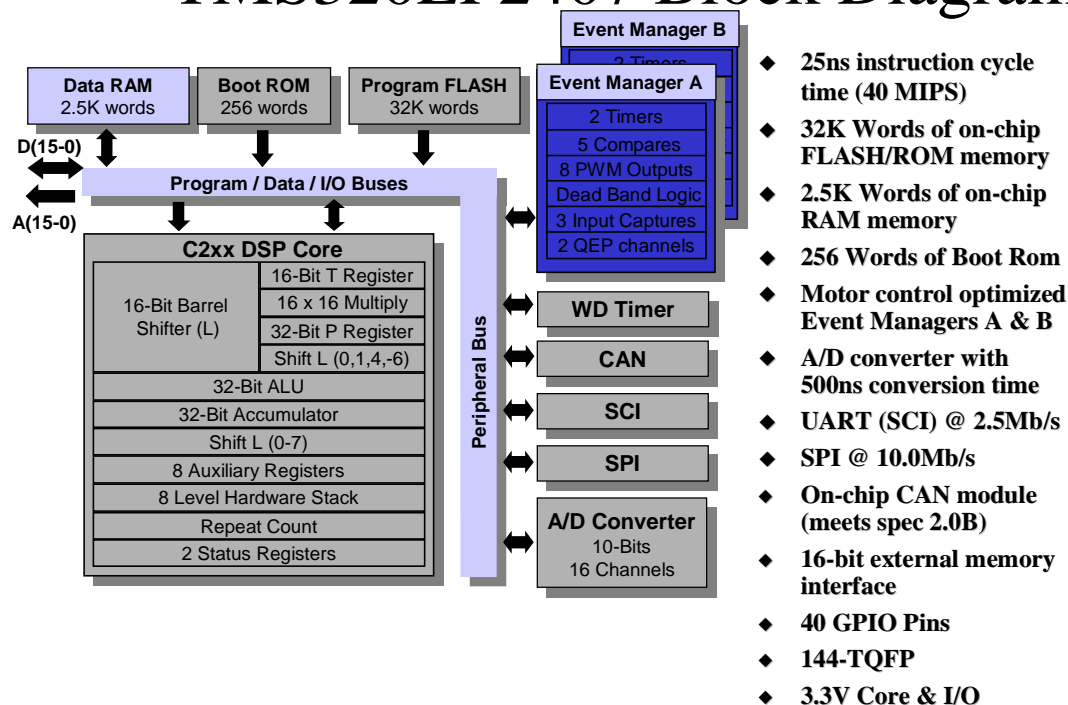


FIGURE 5.5: TMS320LF2407 BLOCK DIAGRAM

The Controller Area Network (CAN) is a well-designed communications bus for sending and receiving short real-time control messages. The bus is designed to connect control systems over a small area (such as automobiles), operating in a noise environment at speeds of up to 1 Mbit/sec. One of the perceived problems of CAN is the inability to bound the response times of messages. To show how this problem can in fact be easily solved, the analysis developed for fixed priority preemptive real-time processor scheduling applied to a CAN bus is shown in the next Chapter.

## 5.4 Summary

This work researched techniques to enhance the fault-tolerance capability of hard real-time systems by incorporating time redundancy, processor redundancy and fault tolerant real-time communication protocols. The main goal was to guarantee fault-tolerance requisites for multiprocessor hard real-time systems. The resources used to optimize these fault-tolerance

requisites were task schedulability and time redundancy. The implementation of a DSP-based multiprocessor system interconnected by a CAN bus was also presented.

# Chapter 6

## Improving Fault Tolerance in HRTSs

### 6.1 Overview

Scheduling policies in real-time systems need to ensure that tasks will meet their deadlines under all circumstances, even in the presence of faults. These real-time scheduling schemata may be used for the processor, as well as for tasks, communication and other resources that are used by real-time systems, including I/O in general.

Hard real-time systems (HRTSs) have stringent timing constraints, and the consequence of missing task deadlines may be catastrophic. Many real-time systems are embedded in sensors and actuators and function as digital controllers. HRTSs are mostly used in the development of control applications due to their characteristics, and task deadlines are typically derived from the required responsiveness of the sensors and actuators, which are monitored and controlled by it.

Whenever a fault-tolerant system is designed, a redundancy type must be incorporated into the equivalent system, which does not incorporate fault tolerance requisites. If the system belongs to real-time, one of the most important resources of such systems is time. Since time is the central resource of any fault-tolerant hard real-time system, task schedulability and time redundancy become the basic tools to guarantee fault tolerance requisites for a given hard real-time system. Most real-time systems are developed to attend to a specific and complex need, requiring a high degree of fault tolerance. Fault tolerance is an especially vital requirement for HRTS development.

A fault in a system can assume several forms, what makes a system test process for fault detection extremely complicated. To reduce this problem, a system is usually designed in a way that the faults that occur in a system follow a certain standard, or model. Thus, the problem of how to treat the faults becomes much more simplified. Although the faults that can occur in a real system are not necessarily modeled faults, there are fault models that are able to cover a very large percentage of the faults that do occur.

All designers rely on an abstract fault model of the system when building a fault-tolerant system. An abstract fault model may be defined as:

- The set of component types.
- The interfaces for each component.
- The interactions between components.
- The set of possible faults.
- A set of symptoms, or observable fault-detection events.
- The component behavior during faults.

In a higher abstraction level, these components may be grouped in a set of states, as presented in Figure 6.1.

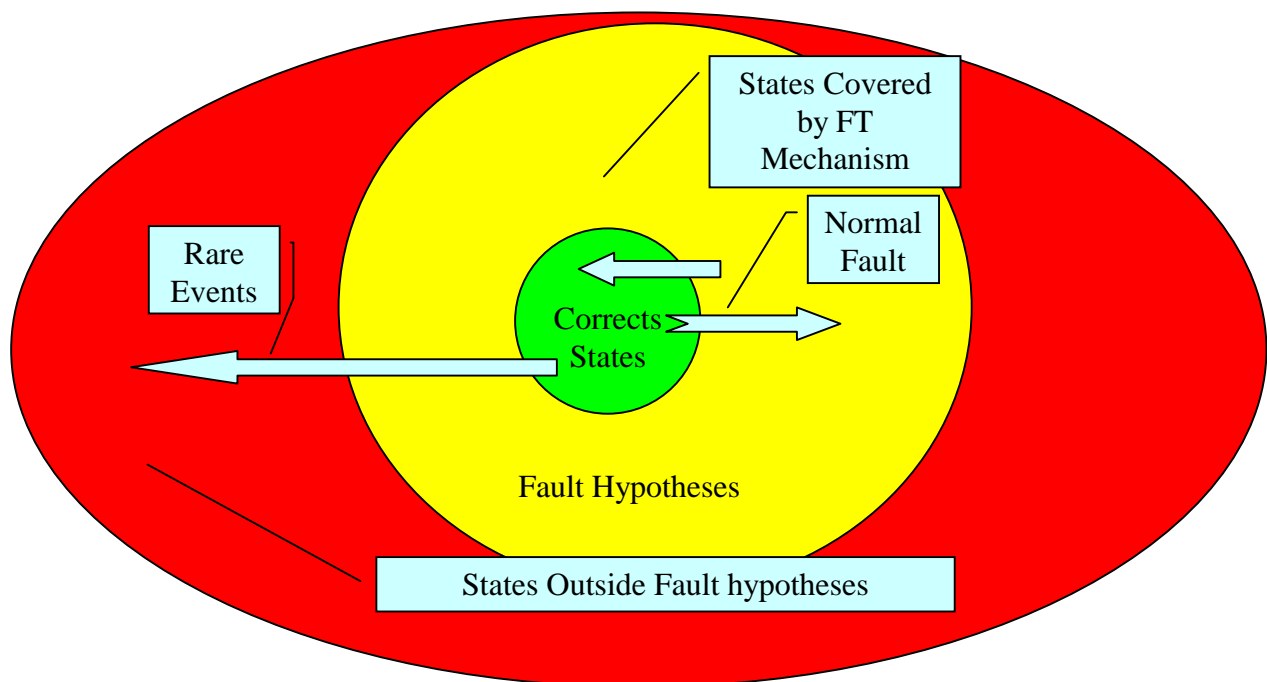


FIGURE 6.1: FAULT MODEL DEFINITION

In general, it is expected that the system works mostly in a set of states called correct states. Moreover, it is also expected that a set of faults may occur, and will be treated by the system without causing a failure. A set of unexpected faults can also occur, but once these rare events are unexpected, they will take the system to an unrecoverable state, where a more serious action may be taken.

## 6.2 System, Task and Fault Models

In the present work, it is assumed that the processors belong to a distributed system and are connected through a CAN bus, as shown in Figure 6.2. Each processor executes applications consisting of real-time periodic control tasks scheduled through RMA, within well-known maximum execution times and frequency rates. An intrinsic characteristic of control tasks is the independency, which means that requests for a certain task do not depend on the initiation or the completion of requests or other tasks and synchronizing tasks execution is not necessary. Another characteristic of control tasks is the fact that they converge to a stable control state.

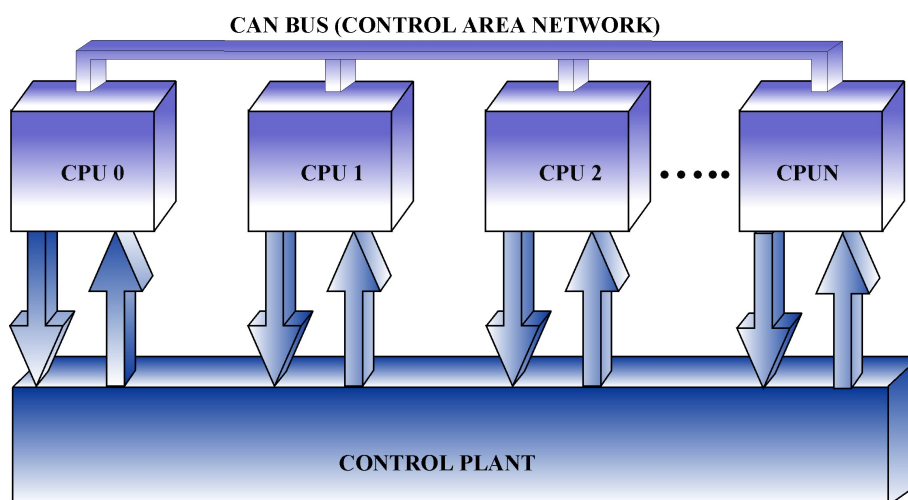


FIGURE 6.2: CAN BUS IN A CONTROL PLANT

The failure characteristics of the hardware and software are the following:

(F1) Transient and fail-stop faults may occur in a processor or even, more specifically, in a task execution process;

(F2) Only transient faults may occur in the bus;

(F3) All nonfaulty processors can communicate with each other;

(F4) Hardware provides fault isolation in the sense that a faulty processor cannot cause incorrect behaviour in a nonfaulty processor; in other words, processors are independent in regard to failures;

(F5) The failure of a processor  $P_f$  is detected by the remaining nonfaulty processors as the absence of a message right after the failure, but within the instant corresponding to the closest task completion time of a task scheduled on  $P_f$ .

## 6.3 Fault Tolerance and Rate Monotonic Scheduling in multiprocessor systems

Any critical real-time system must tolerate permanent faults in addition to transient faults, and transient faults need to be detected before they can be tolerated. A combination of time and hardware redundancy is used to detect and tolerate both transient and permanent faults.

A simple approach to avoid the occurrence of faults in a processor consists of using time redundancy, which allows repeated execution of tasks inside their execution period. The fault will not be perceived by the backup processor since it is processed before the deadline is reached.

Transients and fail-stop faults that may occur in a processor are tolerated in such system. Transient faults are resultant from temporary conditions, such as electromagnetic interferences in hardware or processes abnormal termination. In our fault model, if a transient fault has occurred in a task, the faulty task is re-executed using a slack that was reserved when the tasks were scheduled.

The occurrence of a transient fault in a processor may be caused by a fault in a task that could not be recovered or even by a fault in the operating system execution. In both cases, the processor triggers the execution of a backup task in another processor, while the faulty processor reestablishes its state to a nonfaulty condition. If the fault is transient, then the re-execution allows a correct result to be generated before the deadline of the task.

Fail-stop faults always cause a stop in hardware functioning and end processes execution where they occur. In order to achieve fault tolerance, two or more copies of each task set are used, called primary and backup copies. A backup copy is not necessarily an identical copy of its respective primary copy, once an alternative copy may be used as backup to obtain the desired result using even a different hardware in its execution. All backup copies are passive copies, what means that a backup copy will be executed only if a fault prevents the corresponding primary copy from completing its execution.

Tolerating fail-stop faults is done through hardware reconfiguration, which is also used on recovering from hardware and software permanent faults. Software permanent faults are usually a result of errors in project. In our fault model, if a fault in hardware prevents a task from executing, a backup copy of the task will be executed on another processor or even in the

same processor, assuming it can continue its operation despite the hardware fault. In the second case, an alternative hardware must be used by the processor to execute the backup task. From a processor point of view, a permanent fault is signaled if the primary processor does not generate a result, and the backup processor starts executing. Reconfiguration implies in one CPU assuming the failed CPU's tasks while the latest is taken out of the bus.

It was also included in the fault model errors that may occur in the bus/protocol, once they influence the whole system functioning. In the CAN bus, more specifically, we need to guarantee that a deterministic packet response time can be achieved. A message must be retransmitted a finite number of times in the case of a transmission fault, and still meet its deadline. The occurrence of a transient fault is signaled by the absence of any message, i.e., if a message is not received by the other processors by a certain due time, a fault on the primary processor is assumed and the fault is tolerated through the execution of a backup copy. Any message corruption or further errors indicate that a fault occurred, and an alternative process is executed in another processor to prevent a global failure. Permanent faults in the CAN bus/protocol are not tolerated, causing an interruption in the communication between the processors.

The most important aspect of integrating processors and communications schedulability analysis is to bound the overheads due to packet handling on a given processor. The worst-case number of packets arriving at a given processor in a given time window can be used to bound the worst-case response time of the delivery task.

As shown by Tindell in [9], the scheduling equations for worst-case response time of a processor and worst-case transmission time of a message are mutually dependent. The release jitter of a receiver task depends on the arrival time of a message, which in turn depends on the interference from higher priority messages, which in turn depends on the release jitter of the sender tasks. This way, it is possible to form a recurrence relation where in the first iteration of the scheduling equations we set the inherited release jitter for all tasks to zero. On the  $n$ th iteration the inherited release jitter values can be set according to the results of solving the scheduling equations in the  $(n-1)$ th iteration and so on.

## **6.4 Rate Monotonic Scheduling applied to the CAN bus**

The Rate-Monotonic analysis provides a schedulability test by giving a utilization bound. The Rate-Monotonic Algorithm (RMA), introduced by Liu and Layland [5], is an



algorithm for preemptively scheduling periodic tasks that designates the highest priorities to the tasks with shorter periods, and the RMA is optimum when the tasks are independent.

In [69], Tindell *et. al.* developed a CAN analysis based on RMA showing how to find the response time for messages being transmitted in a CAN bus. In fact, the dynamic scheduling algorithm used by the CAN protocol is virtually identical to scheduling algorithms commonly used in real-time systems to schedule computation on processors [70], and the analysis of the timing behavior of such systems can be applied almost without change to the problem of determining the worst-case latency of a given message queued for transmission on CAN. Since CAN is primarily a priority-based bus, much of the analysis for systems where activities are dispatched according to fixed priorities can be applied directly.

A CAN message consists of between 1 and 8 bytes of data and is uniquely addressed to a CAN node. A given message is assumed to be queued cyclically (i.e. at intervals, the source of the message queues messages of the same size and with the same identifier). A given message is queued at a station within a queuing window, with a minimum interval between subsequent queuing windows (messages do not have to be strictly periodic: a message can be sporadic, but there must be a minimum time between the queuing of the message). The period of a given message  $m$  is denoted as  $T_m$ . The worst-case response time of a given message  $m$  is the longest time between the queuing of a message and the time the message arrives at destination stations, and is denoted  $R_m$ . A message is said to be schedulable if and only if its worst-case response time, given by the longest time between the queuing of the message and the time it arrives at destination stations, is less or equal to the deadline of the message.

As defined by Tindell, the worst-case response time ( $R_m$ ) is composed of two delays: the queuing delay and the transmission delay. The queuing delay, denoted by  $t_m$ , is the longest time that a message can be queued in a station and be delayed because other higher and lower priority messages are being sent on the bus. The transmission delay, denoted by  $C_m$ , is the time taken to actually send the message on the bus.

The queuing delay is itself composed of two times: the longest time that any lower priority message can occupy the bus, known as blocking time  $B$ , and the longest time that all higher priority messages can be queued and occupy the bus before the message  $m$  is finally transmitted, termed the interference. From earlier scheduling theory [1], the interference from higher priority messages over an interval of duration  $t$  is:

$$\sum_{\forall j \in hp(m)} \left\lceil \frac{t + J_j + \tau_{bit}}{T_j} \right\rceil C_j \quad (6.1)$$

Where  $T_j$  is the period of a given message  $m$ ;  $J_j$  is the jitter on the queuing of message  $m$ ; and  $C_j$  is the worst-case time taken to physically transmit the message on the bus.

The set  $hp(m)$  is composed of all the messages in the system of higher priority than message  $m$ . The term  $\tau_{bit}$  is the time taken to transmit a bit on CAN. Note that the set  $hp(m)$  defines a priority ordering. In fact, in the presence of queuing jitter, the optimal ordering is to select priorities on the basis of:  $D_m - J_m$ , where  $D_m$  represents the deadline of message  $m$ .

That is, the smaller the value of  $D - J$  the higher the message priority. From the above description we can see that the queuing delay is given by:

$$t_m = B + \sum_{\forall j \in hp(m)} \left\lceil \frac{t_m + J_j + \tau_{bit}}{T_j} \right\rceil C_j \quad (6.2)$$

A recurrence relation can be formed:

$$t_m^{n+1} = B + \sum_{\forall j \in hp(m)} \left\lceil \frac{t_m^n + J_j + \tau_{bit}}{T_j} \right\rceil C_j \quad (6.3)$$

A model for error handling must also be included, once it is important in a fault tolerant scenario.

In a CAN bus, an error detected by either the sender of a message or a receiver station is signaled to the sender station, which must re-transmit that message. The costs of error handling  $E(t)$  are given as the most probable bound on the overheads due to errors in an interval of duration  $t$ , and it includes the cost of re-transmission. Given a configuration of CAN in a given environment, the cost of error handling can be defined using statistical analysis based on the error characteristics. In general, a given message  $m$  would be delayed by lower priority messages for up to time  $(n + 1)B$ , where  $n$  is the number of re-transmissions of message  $m$ . A probable bound on the error recovery overheads before a message  $m$  arrives at the destination is clearly given by  $E(R_m)$ , which can be re-written as  $E(t_m + C_m)$ . Including the overheads due to error handling for the transmission of a given message  $m$  leads to:

$$t_m = E(t_m + C_m) + B + \sum_{\forall j \in hp(m)} \left\lceil \frac{t_m + J_j + \tau_{bit}}{T_j} \right\rceil C_j \quad (6.4)$$

Tindell [69] has shown how to find the worst-case response time of a given message queued for transmission across a CAN bus. An extended analysis can also be found in [69], which also includes remote transmission request messages.

## 6.5 Improving Reliability in a CAN bus

Scheduling policies in real-time systems need to ensure that tasks will meet their deadlines under all circumstances, even in the presence of faults. These real-time scheduling schemata may be used for the processor, as well as for tasks, communication and other resources that are used by real-time systems, including I/O in general.

Whenever a fault-tolerant system is designed, a redundancy type must be incorporated into the equivalent system, which does not incorporate fault tolerance requisites. Since time is the central resource of any fault-tolerant hard real-time system, task schedulability and time redundancy become the basic tools to guarantee fault tolerance requisites for a given hard real-time system.

In the CAN bus, more specifically, the delivery of low priority messages may be compromised if the bus is flooded with higher priority messages. In this sense, we need to guarantee that an overload in the bus will not occur and a bounded packet response time can be achieved.

### 6.5.1 Applying Fault tolerance requisites to a CAN bus

Fault tolerance requirements make a real-time system even more complicated because faults must be detected and tolerated within the timing constraints of the tasks. If faults trigger backup tasks for recovery purposes, the backup tasks must also be executed before the task deadlines.

Thanks to the critical nature of the tasks in hard real-time systems, it is essential that faults be tolerated. Transient faults in real-time systems are generally tolerated using time redundancy, which involves the re-execution of any task running during the occurrence of a transient fault [53]. In this context, the approach is to maintain enough slack (backup time) in the schedule so that any message instance can be re-transmitted if a fault occurs during its transmission. If no faults occur, messages are transmitted just following the usual RMS scheme and the slack is not used. If a fault occurs in the transmission process of a message, a recovery scheme is used to re-transmit that message. The ratio of slack  $S$  available over an interval of time  $L$  is thus constant and can be imagined to be the utilization of a backup message  $B$ . If the backup utilization is  $U_B$ , then the slack available during an interval  $L$ , denoted by  $B_L$ , is  $B_L = U_B L$ .

Ghosh [17] showed a recovery scheme for single and multiple faults that ensures the re-execution of any task after a fault has been detected. Once the dynamic scheduling algorithm used by the CAN protocol is virtually identical to scheduling algorithms commonly used in real-time systems to schedule tasks, this recovery scheme may be also used to ensure the re-transmission of any message in a CAN bus, and the following conditions must be satisfied:

[S1]: There should be sufficient slack for every instance of each message to be re-transmitted. That is, the slack between  $kT_i$  and  $(k + 1) T_i$  should be at least  $C_i$  for any value of  $k$  and  $i$ , what ensures the availability of sufficient slack for a message to be re-transmitted.

[S2]: When any instance of  $\tau_i$  finishes executing, all the slack available within its period (at least  $C_i$  if [S1] holds) should be available for the re-transmission of  $\tau_i$ . This slack can be used after a message finishes transmitting to re-transmit that message before its deadline, if a fault is detected.

[S3]: When a message re-transmits, it should not cause any other message transmission to miss its deadline, allowing all tasks to meet their deadlines even when a high priority task needs to re-execute.

If these three conditions are met, then it is possible to re-transmit a faulty message and meet its deadline. However, a recovery scheme must define also how the slack should be used and a very straightforward scheme consists on the faulty message simply being re-transmitted at its own priority.

This approach to distribute slack in the schedule can be applied to any non-fault-tolerant scheduling scheme for preemptive, periodic tasks where the RMS assumptions hold. As shown by Ghosh [17], any transmission time  $C_i$  in the non-fault-tolerant scheme can be split into two parts for the fault-tolerant scheme: a new transmission time  $C_i' = C_i(1 - U_B)$  (where  $U_B$  is the backup utilization) and a slack equal to  $C_i U_B$ . To guarantee the re-transmission of a message before its deadline, its *critical instance* is considered, which is defined as the time at which the message's transmission is maximized - it happens when the message starts its transmission process simultaneously with all higher priority messages [5].

The total slack available for any message at its critical instance is equal to the total slack available within a period boundary, which is defined as the beginning of a period.

By splitting up each transmission time  $C_i$  into a new transmission time  $C_i'$  and slack, as described above, the utilization of each task  $\tau_i$  is reduced to  $U_i(1 - U_B)$ , and thus the following general fault tolerance boundary for an RMS ( $U_{G-FT-RMS}$ ) is obtained:

$$U_{G-FT-RMS} = n(2^{1/n} - 1) (1 - U_B) = U_{LL-RMS}(1 - U_B) \quad (6.5)$$

The above equation is a general one applicable to an RMS for any value of  $U_B$ . If  $U_B = \max\{U_i\}$ ,  $i = 1, \dots, n$ , then any message transmission in the system can tolerate a single fault. Any number of faults can be tolerated if [S1] holds.

Multiple faults within two consecutive period boundaries are also guaranteed to be tolerated using the scheme described above. If several backups are provided in the system, and the total backup utilization is  $U_{BT}$ , then a general boundary for the message set can be derived by replacing  $U_B$  with  $U_{BT}$  in  $U_{G-FT-RMS}$ ; that is, the new boundary is  $U_{LL-RMS}(1 - U_{BT})$ .

### 6.5.2 Limiting the maximum transfer rate of the CAN bus

Considering the CAN protocol, the absence of a message is identified by the CPUs connected to the bus as a fault. In this case, a fault model is applied, implying in the re-transmission of the message that failed in its transmission or even in the execution of an alternative action such as the reconfiguration of the bus.

In this scenario, we can say that RMA applies since we are dealing with periodic control tasks and the following conditions hold:

- Each message has a maximum transmission time, as presented in Section 3.1, and deterministic period.
- The messages are independent, which means in other words that they are asynchronous to each other.
- The messages will have their priority ascertained by RMA.

In order to apply the RMS scheme proposed by Ghosh in the CAN bus, we must guarantee that [S1], [S2] and [S3] are satisfied.

With the worst-case transmission time presented above, it is possible to define the backup slack size and ensure the availability of sufficient slack for a message to be re-

transmitted. This slack can be used after a message finishes transmitting to re-transmit that message before its deadline, if a fault is detected. This means that [S1] and [S2] hold for the CAN protocol.

The third condition [S3] may not hold for a CAN bus, once a higher priority message re-transmission may prevent the transmission of a lower priority message, causing the latest to miss its deadline. In truth, a higher priority message re-transmission can flood the bus compromising the delivery of low priority messages.

Thus, to guarantee that a CAN bus can tolerate faults according to Section 6.5.1, we define a maximum transmission rate for each message instance; instead of pre-defining a time-driven slot as is the main idea of TTP/C. With this idea in mind, we will not limit a node transmission to its slot, but allow it to transmit at any time if its transmission rate allows. We will call this extension of the CAN bus as RMCAN, which means Rate Monotonic CAN.

From the protocol point of view, the retransmission of a finite number of messages in the case of a transmission failure may not compromise the bus bandwidth and the RMS may be applied deterministically.

This way, any message corruption or further errors indicate that a fault occurred, and the message must be re-transmitted and also meet its deadline. Moreover, if the fault persists, a failure is detected and an alternative process must be executed in another processor to prevent a global failure.

The following example shows how RMCAN can be used to determine whether a message can be re-transmitted before its deadline with guarantees in a CAN bus. Consider a set of 10 processors,  $N = 10$  - sending 5 periodic messages with utilizations  $U_i = 1\%$  for each message  $i$ . Each processor is also limited to a maximum data transmission volume of 10% of the bus bandwidth. Once this limit is reached, the processor that is sending messages stops sending messages and higher layers of the processor that should be receiving the messages will tolerate the fault taking the appropriate action in the context of the specific application.

If we assume that a message fault needs to be tolerated and re-transmitted up to 5 times, then  $U_B = 5\%$ . Equation (6.6) gives us a bound of 66% while the sum of utilizations of the task is 50%.

$$100 (2^{1/100} - 1) * (1 - 0,05) = 0,66 \quad (6.6)$$

Since  $\sum U_i < UG\text{-}FT\text{-}RMS$ , the messages are schedulable.

## 6.6 Formal Verification of RMCAN

Techniques of formal verification represent the functioning of a circuit or the execution of an algorithm through mathematical models. Mathematical operations on these models are accomplished in a precise and non-ambiguous form, in order to verify if a specific property is satisfied or not.

Diverse research projects are based on the development of automatic verification tools, which are frequently based on techniques of symbolic model checking [73][74].

Tools for formal verification of real time systems must include constructions for the time representation. An example of a tool developed for this purpose is VERUS [75], which was used for the formal verification of RMCAN. A brief explanation of the RMCAN formal verification process is shown below. The detailed description can be found in [72]

### 6.6.1 Modeling Premises

RMCAN assumes, but does not implement, a clock synchronization mechanism between the nodes that make use of the CAN bus. It assumes that the time constants in the control domain are orders of magnitude bigger than the differences of clock between the fault tolerant nodes. Thus, the devices must be built from similar microcontrollers, so that the smallest difference between the view of time passage each one have can be achieved.

The formal verification of RMCAN was evaluated over two aspects: evaluation of its capability to provide guarantees over the delivery of messages and evaluation of its capability on detecting, recovering and tolerating faults.

### 6.6.2 Formal verification of the capability to guarantee message delivery and of the recovery and fault tolerance capability of the protocol

All the possible combinations of message transmission had been modeled in VERUS, in order to guarantee that all messages will reach their destination inside a maximum time despite the collisions that may occur in a CAN bus.

### 6.6.2.1 Modeling Characteristics - capability to guarantee message delivery

The message transmission of an implementation instance of RMCAN consisting of 3 nodes was modeled (Figure 6.3). Consider nodes A, B and C, with transmission rates of two messages per time unit for nodes A and B and one message per time unit for node C. Thus, node C has the highest priority among all nodes. According to RMCAN, the transmission rate of the CAN bus must be therefore of five messages per time unit.

The basic time unit in the modeling was a NTU – Network Time Unit – which it is the latency for the delivery of a message over the CAN bus.

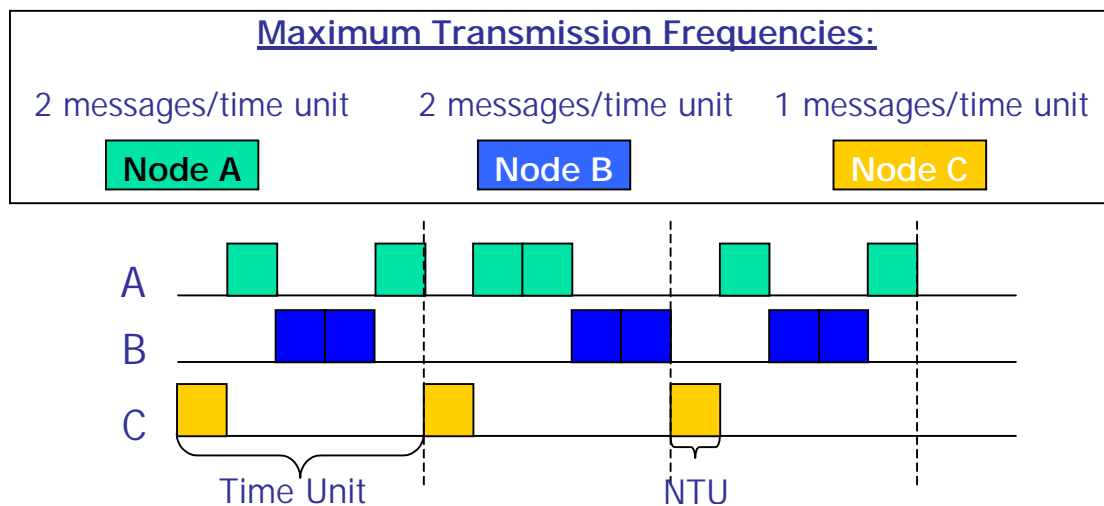


FIGURE 6.3: RMCAN REPRESENTATION

The messages sent by the system inside a period must follow some temporal restrictions. There must be an order for the messages sent by a node, for example, for node A, message a1 must be sent before message a2.

The generation and control of all possible executions of the protocol had been implemented from a VERUS command (SELECT).

The control of the priorities of each message was implemented during the verification of a message collision. A comparison between the possible message collisions was implemented to guarantee that the message of higher priority will be delivered while the one with lower priority will be re-sent in the next NTU.



Messages from node C have the highest priority. Once the nodes A and B have the same number of messages per period, it was defined arbitrarily that the messages of node B are of higher priority than the messages of the node A.

In case of a concurrent transmission of message a1 and any higher priority message, it will be re-sent in the next NTU, otherwise, it will not be sent before the next 5 NTUs.

The protocol verification aims to guarantee that the messages sent during a period will be delivered inside of the same period. All the possible executions of the protocol were implemented and verified, in order to guarantee that the protocol premises follow.

Example of properties that were verified are: (1) a property that guarantees that the higher priority message will be retransmitted only in a new protocol execution period; (2) a property that guarantees that the messages will be sent in at most 5 NTUs; (3) a property that guarantees that, if node A starts sending messages in instant 1, an instant where no collisions will occur happens before the fifth NTU.

#### ***6.6.2.2 Modeling Characteristics - recovery and fault tolerance capability of the protocol***

An instance of a fault tolerant cluster composed by three nodes (A, B and C) was modeled. The three nodes are responsible for executing periodically one specifically task. Each node must execute its task and send a message to the other nodes of the cluster, informing its execution up to a time unit before the expiration of the period.

The execution period of node A, for example, is of five time units, which means that task A is executed by the original node in, at most, four time units. If the confirmation of the fulfillment of the task is not sent by node A, the backup task is executed in the time unit left.

In order to simplify the modeling process, we assume that the execution of the backup task will take only one time unit. This premise can be extended for any number of time units. All nodes of the cluster have the same notion of what a time unit is for RMCAN, which is the time limit for the delivery of the messages of a node.

As a consequence of the execution flexibility of RMCAN, the execution of a task can occur in any instant of time before the deadline, and it must be modeled by the tool. VERUS generates all the possible executions of the protocol for all instants of a task completion and for the occurrence or not of a failure in each of the original tasks.

To guarantee the fault tolerance requisites, a backup task for each relevant task of the system must be implemented. In case of a missing confirmation message (indicating that a

task execution was completed sent by the original node) before the pre-defined time limit, another node will detect the fault and re-execute the task. Figure 6.4 shows properties that aim to define the maximum time for the execution of a task, which can be carried through by the original node or even by the backup task.

```

spec
MAX(!pA.tarefa, (pA.tarefa || pABackup.tarefa));
MAX(!pB.tarefa, (pB.tarefa || pBBackup.tarefa));
MAX(!pC.tarefa, (pC.tarefa || pCBackup.tarefa));

```

FIGURE 6.4: PROPERTIES CHECKED FOR THE RECOVERY AND FAULT TOLERANCE CAPABILITY OF THE PROTOCOL

From the verification of RMCAN it is possible to guarantee that all the messages of a node will be delivered inside the protocol execution period, which will be used by the fault recovery mechanism as a time unit. In a fault scenario, the receiving nodes can, after this time unit, detect the failure in the message transmission and initiate the system recovery process. This time unit is known by all cluster nodes and it sets the time-triggered portion of the protocol for a fault recovery.

## 6.7 Comparison of Real-Time Fault-Tolerant Communication Protocols – advantages of the proposed method

This comparison does not reiterate the common design decisions, but focuses on the differences between TTP/C, CAN, TTCAN and FlexRay protocols and shows the advantages of the adaptation proposed over the CAN protocol in this work (RMCAN).

From the buses considered previously, only TTP/C is solely time-triggered while the CAN bus is event-triggered. TTCAN and FlexRay combine time-triggered and event-triggered operation aiming to be more flexible than TTP/C and safer than CAN protocol. This time-triggered versus event-triggered decision is a fundamental design choice that influences many aspects of their architectures and mechanisms. The mechanism adopted by each protocol to resolve transmission concurrency between nodes is decisive to indicate if collisions or concurrency occur during runtime.

Analyzing the performance of these protocols, we may see that latency is constant and known at design time for TTP/C, while it may increase with load in a CAN bus. The main problem with the CAN bus is that it cannot prevent an overload of the communication system, which may cause a disastrous result when the delivery of low priority messages is prejudiced by higher priority messages re-transmission. RMCAN solves this problem by limiting the transmission rate of a node to a maximum value. In this case, the worst-case and latency are precisely known.

In this sense, little is known about the FlexRay protocol, which has not been released yet. All that is known is that FlexRay provides no services to its applications beyond best efforts message delivery. A never give up strategy inside FlexRay leaves the control of the communication system with the application, and latency will be constant and precisely known at design time for the TDMA window.

Resuming, we may say that TTP/C provides an off-line communication design yielding guaranteed latency for all messages in the system, but presents low flexibility once bandwidth is distributed at design time by assigning frames of specific length to each node. In a CAN bus, otherwise, priorities are distributed at design time by assigning unique identifiers and a full control by the application over the bandwidth distribution. The CAN protocol is highly flexible and widely available, although some extensions must be done to guarantee a reliable mechanism to build fault-tolerant safe-critical systems. TTCAN is a compromise and represents the necessary evolution of CAN for dealing with heavier loads on the bus. However, synchronizing nodes is not a simple task, and brings a new complexity to the CAN bus. RMCAN was developed to be as efficient as TTCAN without incorporating extra hardware or difficulties. Both protocols can be implemented using a regular CAN microcontroller, although for TTCAN it is also necessary an extra hardware for the time-triggered portion of the protocol. Implementing RMCAN is much easier, and does not require any extra hardware. The transmission rate of each message set can be controlled through software.

FlexRay can be considered the state-of-art in the real-time fault-tolerant communication protocols area, although it has not been released yet. It promises a higher bit rate than TTCAN an increase in flexibility when compared to TTP/C.

TABLE 6.1: COMPARISON BETWEEN TTP/C, CAN, TTCAN, RMCAN AND FLEXRAY PROTOCOLS

	TTP/C	CAN	TTCAN	RMCAN	FlexRay
Media access strategy	Time-triggered	Event-triggered	Time and Event-triggered	Event-triggered and transmission rate directed	Time and Event-triggered
Amount of application data transmitted per packet	1-16 bytes per frame	1-8 bytes per message	1-8 bytes per message	1-8 bytes per message	1-12 bytes per frame
Dynamic bandwidth sharing among nodes	No	Yes	Yes	Yes	Yes
Market presence	< 1%	> 99%	May explore the CAN protocol market presence		Not available
Data Efficiency vs. Latency	Constant and known at design time	Increases with load	Constant	Constant	Constant and known at design time
Response Time	Bounded	Unbounded	Bounded	Bounded	Bounded

The use of the CAN protocol in the development of applications is favored by the high availability of microcontrollers incorporating the bus. Today, the biggest advantages of CAN compared to other networks are the costs and the price/performance ratio. The enhancements proposed by TTCAN and RMCAN are examples of how CAN problems can be circumvented and its spread presence in the market can be explored.

## 6.8 Target Application: Design of an Uninterruptible Power Supply

### 6.8.1 UPS Functionality

An example where RMCAN was applied consists of an 80KVA Three-Phase Double Way UPS system, which must provide uninterruptible power supply to the loads connected to it even when there is a fault in the commercial energy supply. It can be used with either redundant or parallel configuration, including small units (10kVA), amplifying their reliability and increasing their capacity.

An UPS system is an example of a hard real-time system where fault tolerance is essential. The primary function of an UPS (Uninterruptible Power Supply) is to ensure continuity of an alternating power source, especially during a fault or disturbance in the

commercial electric energy supply. It may also serve to improve the quality of the power source by keeping it within defined limits.

The input/output voltages have to be monitored continuously (or at very small discrete time intervals) and reaction to variations has to be done within a few nanoseconds; this way an UPS can be classified as a hard real-time system. Figure 6.5 shows the functionality of an UPS.

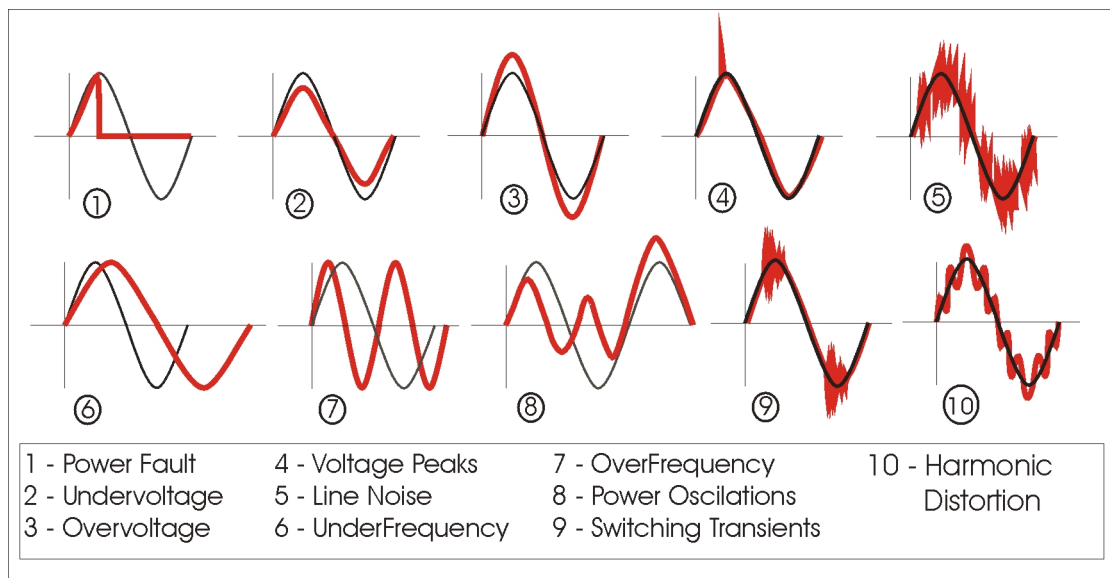


FIGURE 6.5: UPS FUNCTIONALITY – INPUT AND OUTPUT VOLTAGES

## 6.8.2 UPS Architecture and Operating Modes

The full UPS operation is provided through the use of microprocessor controlled logic. All processing tasks are executed by three independent CPUs (rectifier CPU, inverter CPU and static switch CPU), each one consisting of a Texas Instruments DSP model TMS320LF2407A [36]. More CPUs could be used as a hardware redundancy, while the use of less CPUs is also possible if we have two functions executed by the same processor. Although these CPUs can operate autonomously, they communicate through a CAN interface, which allows monitoring and control information exchange; each DSP incorporates an on-chip CAN module.

Figure 6.6 shows a piece of the UPS architecture where the rectifier and the inverter CPUs execute a data synch operation through a CAN bus. Algorithm 1 executes continuously based on the parameters exchanged in the data synchronization between the rectifier and the inverter CPUs. This data synch operation is important to provide a high quality output power.

If a power fault occurs, for example, the output power level remains in its optimal level, as shown in Figure 6.7.

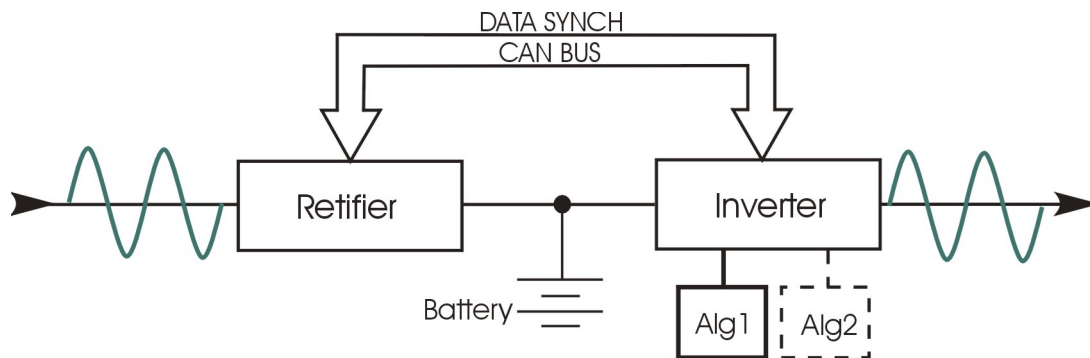


FIGURE 6.6: UPS ARCHITECTURE - DATA SYNCH THROUGH A CAN BUS

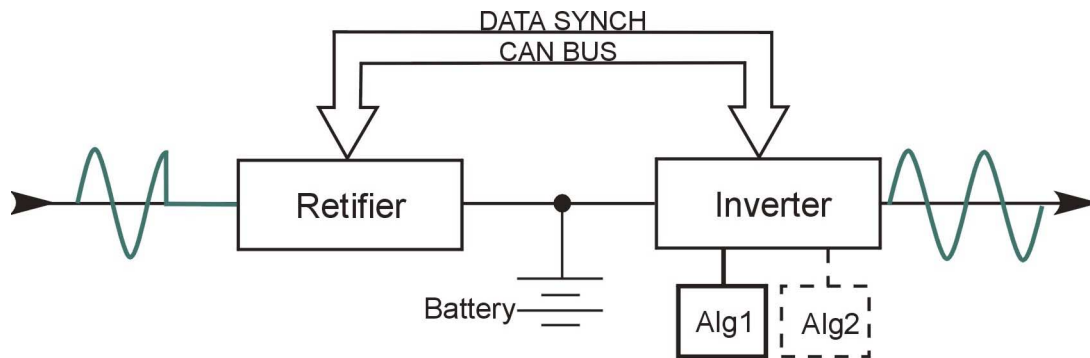


FIGURE 6.7: UPS ARCHITECTURE – OPERATION IN A POWER FAULT SCENARIO

A detailed description of the UPS architecture and operating modes is presented in Appendix A.

### 6.8.3 Fault tolerant support

An UPS system is subject to diverse failures in its components, which may not cause a failure in the whole system. Additional care was taken in order to incorporate fault tolerance concepts in the hardware development process, such as use of redundant power supply, protection against energy source short-circuits that may occur by a component failure and distributed control.

The UPS control software must also circumvent failures such as consumer overload, inverter failure or output voltage variation beyond limits ( $\pm 1\%$ ), in order to guarantee fault-

tolerance requisites and provide an uninterruptible power supply with quality. In case of one of these failures, the UPS switches from the double-conversion mode to the automatic bypass mode. When in double conversion mode, the UPS inverter shall continuously supply power to the critical load. The rectifier/battery charger shall derive power from the utility AC source and supply DC power to the inverter while simultaneously charges the batteries. In the Bypass mode, the inverter or the power interface is operating to provide output voltage conditioning and/or battery charging. The static bypass transfer switch automatically transfer power from the bypass back to the rectifier/inverter, once the load current returns to the UPS nominal rating or less.

Fault tolerance requisites must be incorporated in the CAN bus since it will be applied to an UPS system, which uses the bus to exchange controlling and monitoring information. In such systems, it is essential that the messages reach their destinations according to deadlines, and actions should be taken if any fault occurs.

Figure 6.8 shows how the UPS acts if a fault in the CAN bus occurs concurrently with a power fault. If a fault occurs in the CAN bus, the data synch operation is interrupted and an action may be taken to provide the best output power possible. Algorithm 2 (Alg.2) is thus executed instead of Algorithm 1 (Alg.1), guaranteeing that, in case of a power fault, the output quality will not be affected.

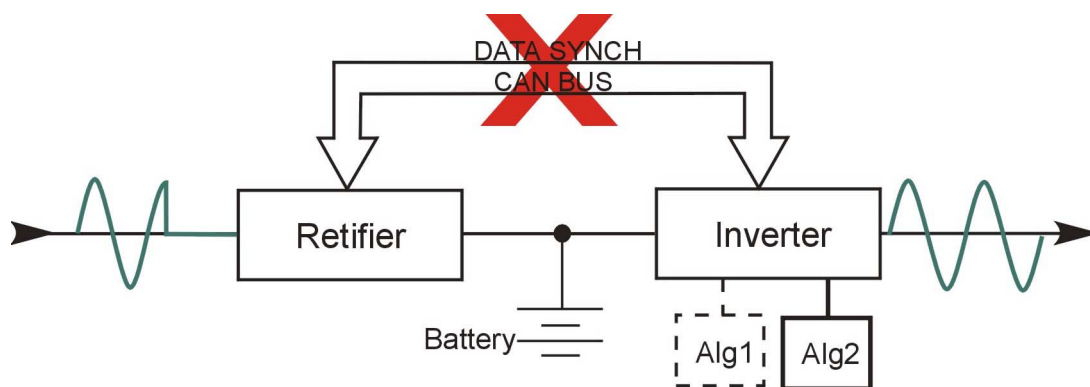


FIGURE 6.8: UPS ARCHITECTURE – OCCURRENCE OF A CAN FAULT IN A POWER FAULT SCENARIO

The results obtained in the execution of Algorithm 2 remain optimal when the loads do not change abruptly, and they are near the optimal value if a transitory occurs.

### 6.8.3.1 Tests and Results

An 80KVA three-phase double way UPS was implemented. It can be used with either redundant or parallel configuration, amplifying its reliability and increasing capacity. The UPS operating modes follows the operating modes presented in Appendix A. It can be used with either redundant or parallel configuration, including small units (10kVA), amplifying their reliability and increasing their capacity. Perturbations to the electrical system are thus avoided, once it filters the disturbances that may occur in the input voltage.

As a test, the UPS was set to provide stabilized output voltage to a load of around 70KVA. We had also connected to the UPS output a 4 ton rolling bridge, which causes a transitory current of 300 A. The overload is twice the nominal power of the UPS when it activates its motors, and the biggest motor consumes 15KVA. In this situation, if the UPS is operating in the double-conversion mode, the inverter will not be able to provide all the power needed, and the UPS reacts switching to the automatic bypass mode, when the power will be supplied by the commercial source. This situation must be considered a fault, which is tolerated by the UPS and registered as an alarm in a report. When the transitory is over, the UPS goes back to the double-conversion mode, and this event is also registered.

Statistics were collected during a week and the UPS tolerated all the faults. Table 6.2 shows the number of mode switchings and their causes. We also show the average quality level, which indicates the percentage of faults tolerated.

TABLE 6.2: RESULTS OBTAINED FROM AN 80KVA UPS

Reason / Mode	To Double Conversion Mode	To Line-Interactive Mode	Average Quality Level
Consumer Overload	924	924	100%
Inverter failure	0	0	100%
Undervoltage	224	224	100%
Overvoltage	35	35	100%

With the results shown above one can notice that the UPS is responding accordingly expected and faults are being tolerated avoiding failures in the power supplied by the UPS.

### 6.8.4 UPSs' parallelism

For expanded system capacity, either a large UPS unit or a multi-unit modular system operating in parallel is conceivable. The former approach may not be practical because of high



initial cost, site installation difficulties, among other problems that may be present in such implementation. By contrast, the latter approach facilitates system expansion and redundancy.

Putting two or more power supply units in parallel provides an improved fault tolerance capacity to the whole system and increases the system reliability. However, each unit may be supplied by different currents, since the power modules are not identical and the resultant circulating current can result in malfunction and destruction of the UPS system. Some factors that contribute to this unbalance are the components tolerance and the circuitry impedance of the output current distribution of each module. The power supply control circuit must be able to calibrate the output voltage of the power supplies and concomitantly distribute uniformly the charge current between the diverse power modules.

In the last few years, different algorithms on converters parallelism have been developed. From the different methods that make use of a communication system we can mention the following:

- Master-Slave Control (MSC);
- Central Limit Control (CLC);
- Circular Chain Control (3C).

Converters parallelism control using these techniques is done through two feedback control systems. The most external system controls the output voltage while the most internal controls the current supplied by the converter. A communication system is also present between the two power modules.

In a master-slave control [76][77], the output current of the master module is used as a current reference by the slave modules, i.e., it is the current that each slave should supply as a result of the correct current partition between modules. The master module is also responsible for controlling the output voltage control system. A variant of the master-slave control [79] proposes that the reference current should be the largest magnitude current among all units, i.e., the unit that supplies the largest current should be the master and the other units would be slave units.

In a Central Limit control [77], the reference current of the most internal system is the pondered sum of all parallel output currents divided by the number of converters. A variant of

this method [78] proposes that just the units that are necessary to supply the load should be turned on, and the others could stay off.

In a Circular Chain control [80], the reference current of the most internal system is the current supplied by the precedent converter in a circular link.

An Output Voltage Droop control [81] doesn't need a communication system between the different control systems. In this method, the frequency and the output voltage are both controlled by the converter active and reactive power flows in a similar way to the one that is used in the control of electric power systems.

An application of the converters parallelism method using a master-slave's control technique with communication through a CAN interface is described in the following session.

#### 6.8.4.1 Applying RMCAN to the UPSs' parallelism problem

Two or more UPSs can be used in either a redundant or parallel configuration, including small units (10kVA), amplifying their reliability and increasing their capacity.

An application of the converters parallelism method using a master-slave's control technique with communication through a CAN interface was used. One UPS is elected the master, which will be responsible for informing a reference current to the slave UPSs. The master and slaves UPSs inverters parallelism is obtained starting from the communication of the load current from the master to all the slaves. Figure 6.9 shows how the slave's reference current is obtained from the master's load current. An additional term is obtained in the proportional controller's output, which guarantees the imposition of the current in the slave's inverter capacitor. This control system reduces the master's current controller effort.

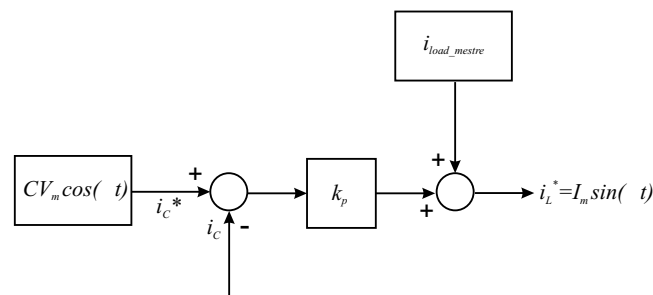


FIGURE 6.9: DETERMINING THE REFERENCE CURRENT OF THE SLAVE UPSs

Figures 6.10 and 6.11 show the results obtained from the simulation of the parallelism of 3 UPS of 4 KVA each. Figure 6.10 shows the output voltage and figure 6.11 shows the master's and slaves' currents, used as a source to a non linear load. Figures 6.12 and 6.13 show the active and reactive power divisions between the UPS units. A perfect division of the

active power can be observed. The error in the division of the reactive power was considered acceptable.

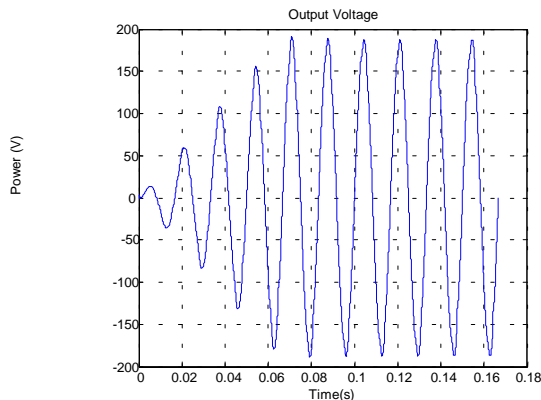


FIGURE 6.10: OUTPUT VOLTAGE

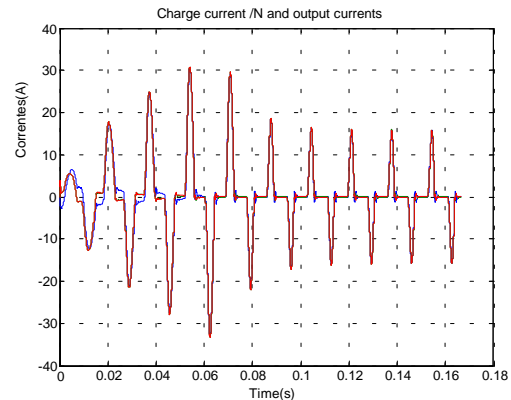


FIGURE 6.11: DISTRIBUTION OF THE CURRENT ON THE INVERTERS

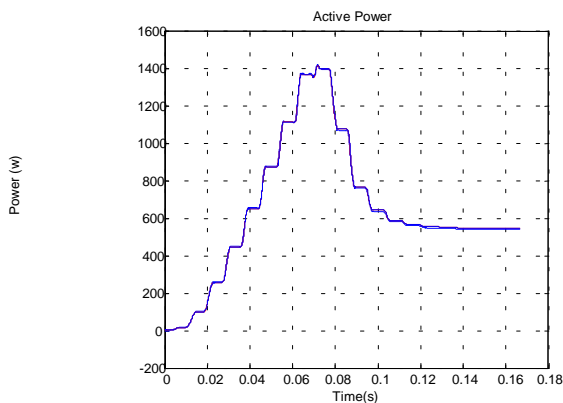


FIGURE 6.12: DISTRIBUTION OF THE ACTIVE POWERS

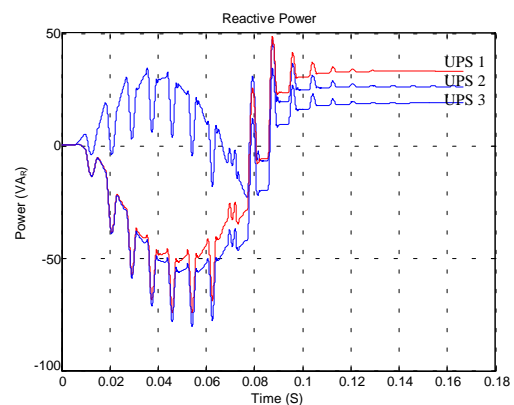


FIGURE 6.13: DISTRIBUTION OF THE REACTIVE POWERS

#### 6.8.4.2 Algorithm for the case of Failure in the Master Controller

One of the advantages of UPSs parallelism is the increase in reliability through redundancy. For example, in the case of a slave UPS failure, the load current is automatically redistributed among the remaining UPSs, without compromising the quality of the energy supplied to the load. However, in case of a master UPS failure or a communication failure between the master and the slave units, it is necessary that one of the slave UPSs assumes the master's function quickly so that a voltage charge interruption does not occur.

In this work, the communication among the UPSs is accomplished through a CAN network. This communication is accomplished in a frequency of 5120Hz, three times slower than the sampling and PWM frequencies. An order is established a priori, among the slaves for the

master unit substitution. In the occurrence of a communication failure with the master, all the slave units maintain their reference current the same as the last value received from the master unit (for the first failure period in the communication). If the communication error persists in the next communication period, the first slave assumes the master's role control.

Figures 6.14 and 6.15 show the current and voltage wave forms in the load, obtained in simulation, in the instant that a failure in the UPS master happens. The failure happens in the positive peak of the current load. In the communication period subsequent to the failure, the master does not supply any current to the load and the UPS slave units maintain the same current value of the previous period. Soon after, the first slave assumes the master's functions. The total voltage harmonic distortion observed was less than 5%, which is considered acceptable to the UPS market specifications.

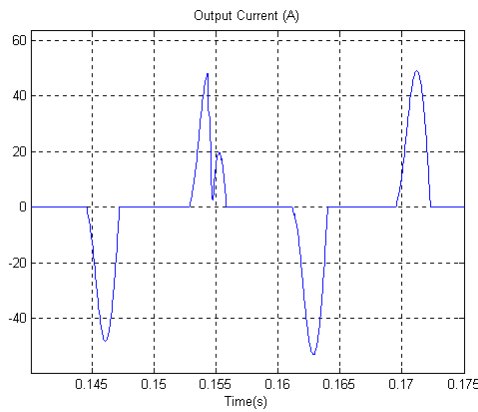


FIGURE 6.14: TOTAL CURRENT IN THE LOAD DURING A FAILURE IN THE UPS MASTER

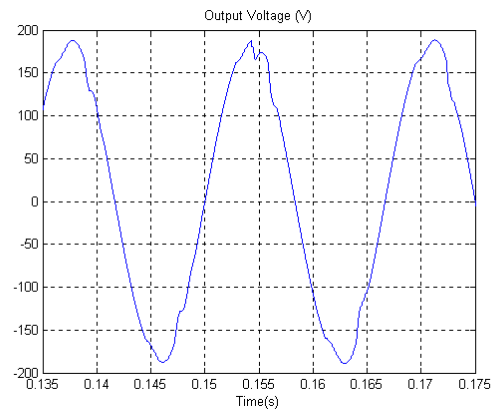


FIGURE 6.15: VOLTAGE IN THE LOAD DURING A FAILURE IN THE UPS MASTER

## 6.9 Summary

The inclusion of fault tolerance in the design of real-time systems must consider timing constraints imposed by the application. One of the essential services provided by real-time fault tolerant distributed architecture is communication of information from one distributed component to another. In this sense, the CAN protocol was created and it is nowadays present in 90% of the microcontrollers and DSPs incorporating real-time protocols. One problem that might be found in a CAN bus is related with the delivery of low priority messages if the bus is flood with higher priority messages.

In this work, we applied the well-known RMS techniques for uniprocessor systems to the CAN protocol and used these techniques to schedule messages on the bus. The approach of adding slack to a schedule was used to tolerate transient faults in the bus. We also restricted the

maximum packet transmission rate for each node to a maximum value and using time redundancy we ensured the re-transmission of any message, before its deadline, after a fault has been detected. Thus, a bounded response time was achieved making the CAN bus more reliable for dealing with heavier loads on the bus.

This theory is being applied to an UPS as an additional fault tolerance support to be used in conjunction with other redundancies already implemented.

# Chapter 7

## Conclusions

### 7.1 Summary

Tasks in real-time systems must meet their deadlines under all circumstances, even in the presence of transient or permanent faults. This work has shown that time redundancy through scheduling is a powerful tool to deal with faults in real-time systems. The harmonious integration of the available techniques enhance the fault tolerance capability of multiprocessor hard real-time systems.

Relating to real-time communication protocols, the time-triggered and event-triggered approaches find favor in different application areas, and each has strong advocates [59]. The CAN protocol may have a unbounded response time for an arbitrary low priority message. Researchers sometimes say that the CAN protocol is more appropriate for soft real-time systems (flexible requirements), while appropriate protocols for hard real-time systems include TTP/C. RMCAN, the extension proposed to the CAN protocol, shows that it is possible to bound the message transmission time, thus making possible its use on HRTSs.

In RMCAN there is a limit on the node transmission rate, making the transmission time deterministic, even for low priority messages. We do not limit a node transmission to its slot, but allow it to transmit at any time if its transmission frequency allows. This way one can guarantee that a message will arrive at its deadline or it will not arrive anymore, in which case a backup action is taken.

One advantage of the CAN protocol over time-triggered protocols is the extensibility aspect. New nodes can be added to the bus, while in the TTP/C protocol, for example, a slot for a new node has to be reserved at design time. Other advantages are the high availability of microcontrollers incorporating the CAN bus, and the price/performance ratio. The enhancements proposed to the CAN protocol show that its reliability can be increased and its spread presence in the market can be further explored.

Time is the central resource of any fault-tolerant hard real-time system, task schedulability and time redundancy become the basic tools to guarantee fault-tolerant requisites for a given hard

real-time system. The main approach to fault tolerance is the use of time redundancy in addition to software (such as in N version programming) and hardware (processor watchdog timer) redundancy to tolerate both transient hardware and software faults. Permanent, transient and correlated faults can be tolerated supported by time and processor redundancy when scheduling backup tasks in a multiprocessor system.

The schedulability tests for new tasks provide the guarantees required for fault-tolerant execution in real-time systems. When a new task is being considered for addition into the system, a set of conditions is tested to check whether the fault tolerance guarantees can be provided. If the conditions are met, then the task is accepted, otherwise a backup action can be taken.

Another aspect in multiprocessor real-time systems relates to the interdependence of task execution time and message transmission time. The release jitter of a receiver task depends on the arrival time of a message, which in turn depends on the interference from higher priority messages, which in turn depends on the release jitter of the sender tasks.

In RMCAN protocol, which extends the CAN protocol, a fault is identified by the CPUs connected to the bus as the absence of a message. In this case, a fault model is applied, implying in the re-transmission of the message that failed in its transmission or even in the execution of an alternative action such as the reconfiguration of the bus.

With the worst-case transmission time determined in Chapter 6, it is possible to define the backup slack size and ensure the availability of sufficient slack for a message to be re-transmitted. This slack can be used after a message finishes transmitting to re-transmit that message before its deadline is reached if a fault is detected.

This way, any message corruption or further errors indicate that a fault occurred, and the message must be re-transmitted and also meet its deadline. Moreover, if the fault persists, a failure in that system component is detected and an alternative process must be executed in another processor to prevent a global failure.

A holistic schedulability and fault tolerance analysis for distributed hard real-time systems conforming to a particular architecture – simple fixed priority scheduling of processors communicating through messages on a shared broadcast bus such as CAN – is presented. Periodic control tasks are executed in the processors, and fault tolerance is implemented according to a pre-defined fault-model. Single processor schedulability analysis is extended to include timing analysis for hard real-time messages on a communications system and address the delivery costs of messages.

## 7.2 Future Work

Real-time systems will be, in the future, bigger and more complex than those we have nowadays and they are going to be part of distributed systems in dynamic environments. These environments will include specialized systems and will integrate complex time characteristics in different granularities. Moreover, ecological, human and economic catastrophes will be the result of a failure in such systems.

If the tasks require resources other than the CPU, then the scheduler needs to verify that all the resources are available to the primary and backup copies of a task when needed. Each resource can be treated in a manner similar to the CPU while reserving it for a task. For example, overloading and deallocation can be used for each resource. However, when multiple resources need to be reserved simultaneously, then the scheduling is more complicated, and timeline driven dispatching techniques are preferred over priority driven dispatching. A timeline can be used to ensure that each resource is reserved for the tasks.

A natural evolution of this work is its applicability in the networks field, where complex communication architectures and protocols designed for efficient transactions based on temporized messages could be incorporated to the system. Another evolution would be to include support for tasks synchronization, not limiting the system's capabilities to only independent real-time tasks and non-real-time tasks.

In the fault-tolerant real-time systems field, other topics are important research topics, such as:

- **Scheduling Algorithms:** the ability to manipulate complex task structures with precedence characteristics, resources and time characteristics in a dynamic and integrated environment must be incorporated in scheduling algorithms.
- **Operating Systems:** OSs must be designed incorporating negotiating functions, allowing time characteristics management in highly integrated and cooperative environments, in a quick and predictive way.
- **Error Manipulation:** project and analysis in a context that includes performance and safety considering error manipulation, which is composed by error detection, fault localization, system reconfiguration, and recovery.

Real-time systems constitute an inter-disciplinary research area that includes aspects of Control Engineering and Computer Science. A successful implementation of a computer-



controlled system requires a good understanding of both control theory and real-time systems. Ensuring continuity of power supply to critical loads such as life support medical systems, Internet nodes, bank transaction systems, computer systems, flight control systems, etc, accentuates the need for high reliability uninterruptible power supplies (UPSs).

For expanding system capacity, either a large UPS unit or a multi-unit modular system operated in parallel connection is conceivable. The former approach may not be practical because of high initial cost, site installation difficulties, among other problems that may be present in such implementation. By contrast, the latter approach facilitates system expansion and redundancy. The method of parallel operation to increase the UPS rated capacity is implemented by synchronizing each of the units of output voltage frequency and utilizing their voltage magnitude and phase angle through a series inductor to control the power distribution of each unit.

Putting two or more power supply units in parallel provides to the whole system an improved capacity of fault tolerance and increases the system reliability. However, each unit may be supplied by different currents, since the power modules are not identical. When the parallel units are different in output voltage, resultant circulating current can result in malfunction and destruction of the UPS system. Some factors that contribute to this unbalance are the components tolerance and the circuitry impedance of the output current distribution of each module. The power supply control circuit must be able to calibrate the output voltage of the power supplies and concomitantly distribute uniformly the charge current between the diverse power modules.

One of the advantages of UPSs parallelism is the increase in reliability through redundancy. For example, in the case of a slave UPS failure, the load current is automatically redistributed among the remaining UPSs, without compromising the quality of the energy supplied to the load. However, in case of a master UPS failure or a communication failure between the master and the slave units, it is necessary that one of the slave UPSs assumes the master's function quickly so that a voltage charge interruption does not occur.

The work developed in this thesis may bring a great support to the development of such application and others that may follow.

# Appendix A

## UPS Architecture and Operation Modes

Figure A1 presents the UPS architecture and its operating principle, with a detailed view of its control structure.

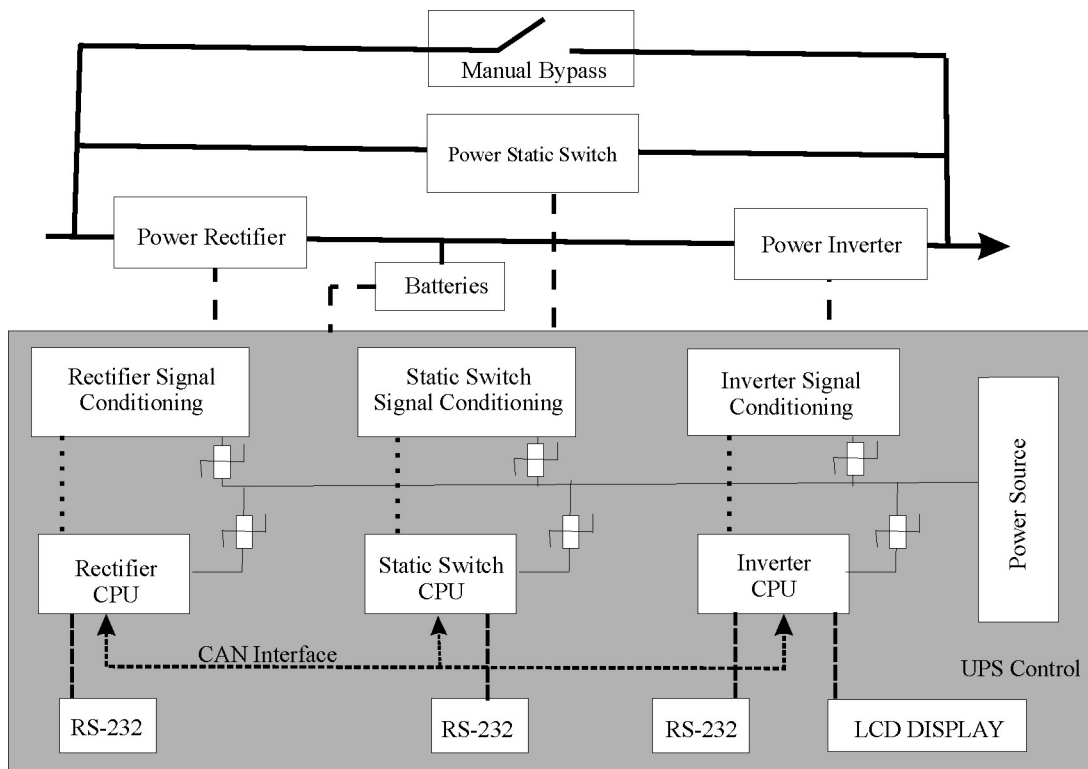


FIGURE A1: UPS ARCHITECTURE CONTROL DETAILED VIEW

An UPS is composed by the following basic blocks:

- **Batteries:** the batteries are responsible for supplying energy during a fault in the commercial energy supply;
- **Rectifier (AC/DC) / Battery Charger:** incoming AC power shall be converted to a regulated DC output by a rectifier. The rectifier shall provide high quality DC power to the power inverter and also to charge the batteries;
- **Inverter DC/AC:** the power deriving from the rectifier or from the batteries shall be converted back to AC power, and must be free of the disturbances that are usually present in the commercial energy supply. One source of energy can be

selected in case of a failure on the other source, allowing uninterruptible power supply.

- **Static Switch:** the static switch is used to select the energy source, which can be the inverter or the commercial energy supply. One source of energy can be selected in case of a failure on the other source, allowing uninterruptible power supply.
- **Manual (static) Bypass:** the bypass is a power path alternative to the indirect AC converter. The bypass transfer switch shall be used to transfer the load to the bypass without interruption to the critical power load. This shall be accomplished by turning the inverter off. This procedure is commonly used in case of an UPS failure or for its maintenance.

The UPS module shall be designed to operate as a double-conversion, on-line reverse transfer system working in the following modes:

- **Normal Mode:** in this mode, the inverter shall continuously supply power to the critical load. The rectifier/battery charger shall derive power from the utility AC source and supply DC power to the inverter while simultaneously float charging the batteries.
- **Stored Energy Mode (on batteries):** the UPS goes to this mode upon failure of the utility AC power source, the critical load shall be supplied by the inverter, which, without any switching, shall obtain its power from the batteries.
- **Automatic Bypass Mode:** the bypass transfer switch shall be used to transfer the load to the bypass without the interruption to the critical load. This shall be accomplished by turning the inverter off. Automatic re-transfer of the load shall be accomplished by turning the inverter back on.
- **Manual Bypass Mode:** also called maintenance bypass, in this mode a power path that allows isolation of a section or sections of a UPS is used for safety during maintenance and/or to maintain continuity of load power.
- **Line Interactive Mode:** in this mode, the inverter or the power interface is operating to provide output voltage conditioning and/or battery charging. The supply frequency is dependent upon the AC input frequency. When the AC input supply voltage is out of UPS preset tolerances, the inverter and the battery maintain

continuity of load power in stored energy mode and the power interface disconnects the AC input supply to prevent back feed from the inverter. The unit runs in stored energy mode for the duration of the stored energy time until the AC input supply returns with UPS design tolerances.

# References

- [1] Dhall, S.K.; Liu, C.L.. “On a Real-Time Scheduling Problem”. *Operations Research*, vol.26, no. 1, pp. 127-140, Jan./Feb. 1978.
- [2] Lawler, E.L.; Lenstra, J.K.; Rinnooy Kan, A.H.G.; Shmoys, D. B.. “Sequencing and Scheduling: Algorithms and Complexity”. *Handbooks in Operations Research and Management Science*, vol. 4, *Logistic of Production and Inventory.*, Amsterdam: North Holland, 1993.
- [3] Parker, R. G.. *Deterministic Scheduling Theory*, Capman & Hall, 1995.
- [4] Leung, J.Y.-T.; Merrill, M.L.. “A Note on Preemptive Scheduling of Periodic Real-Time Tasks”. “*Information Processing Letters*”, vol.11 no.3, pp. 115-118, 18 Nov.1980.
- [5] Liu, C.L.; J.W. Layland. “Scheduling Algorithms for multiprogramming in a Hard-Real-Time Environment”. *ACM Journal*, vol. 20, pp.46-61, 1973.
- [6] Sha, L.; Rajkumar, R.; Lehoczky, J. P.; Ramamritham, K.. “Mode Change Protocols for Priority-Driven Preemptive Scheduling”. *Real-Time Systems Journal*, Vol. 1, 1989, pp. 243-264. Kluwer Academic Publishers.
- [7] Lawler, E.L.; Martel, C.U.. “Scheduling Periodically Occurring Task on Multiple Processors”. “*Information Processing Letters*”, vol. 12, no. 1, pp. 9-12, 13 Feb. 1981.
- [8] Tindell, K. W.. “An extendible approach for analysing fixed priority hard real-time tasks”. *Journal of Real-Time Systems* 6 (2), pp. 133-151, 1994.
- [9] Tindell, K.; Clark, J.. "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems". *Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems) - Microprocessing & Microprogramming*, Vol. 50, Nos.2-3, pp. 117-134, 1994.
- [10] Joseph, M.; Pandya, P. K.. “Finding Response Times in a Real-Time Systems”. *The Computer J.*, vol.29, pp.390-395, Oct.1986.
- [11] Oliveira, M. P.; Fernandes, A. O.. “Deadline: A Multi-Tasks, Fault-Tolerant Core for Real Time Systems.” *Mastership Thesis DCC/UFMG* 1997.

- [12] Oliveira, M. P.; Fernandes, A. O.; Coelho, C. J. N.. “Deadline: A Multi-Task, Fault-Tolerant Core for Real Time Systems”. VIII Fault-Tolerant Computers Symposium. pp 49-53. Computers Institute. “Universidade Estadual de Campinas” 1999.
- [13] Shin, K. G.; Ramanathan, P.. “Real-Time Computing: A New Discipline of Computer Science and Engineering”. Special Issue on Real-Time Systems, Proceedings of the IEEE, Vol. 82 , N° 1, january 1994, pp 6 to 24.
- [14] Pandya, M.; Malek, M.. “Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks”. IEEE Transactions on Computers, Vol. 47, No. 10, pp 1102-1112, October 1998.
- [15] Bertossi, A. A.; Mancini, L. V.; Rossini, F.. “Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems”. IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 9, pp.934-945 September 1999.
- [16] Tennenhouse , D. L. (Intel Director of Research), Keynote Speech at the 20th IEEE Real-Time Systems Symposium (RTSS’99), Phoenix, Arizona, December 1999.
- [17] Ghosh, S.. “Guaranteeing Fault Tolerance Throgh Scheduling in Real-Time Systems”. Ph.D. Thesis, University of Pittsburgh 1996.
- [18] Stankovic, J. A.; Ramamritham, K.. “Advances in Real-Time Systems”. IEEE 1993.
- [19] Stankovic, J. A.; Ramamritham, K.. “Hard Real Time Systems”. IEEE 1988.
- [20] McClean, W.J.. (editor). Status 1993, A Report on the Integrated Circuit Industry. Integrated Circuit Engineering Corporation ICE , 1993.
- [21] Johnson, B. W.. Design and Analysis of Faut-Tolerant Digital Systems. Addison-Wesley Publishing Company, Inc. 1989.
- [22] Rajkumar, R.. IMAGES Project. Integrated Modeling for Analysis and Generation of Embedded Software. <http://www.ece.cmu.edu/~webk/images/> CMU 2001.
- [23] MoBIES Model-Based Integration of Embedded Systems. DARPA 2000. <http://www.rl.af.mil/tech/programs/MoBIES/> .
- [24] Wing, J.. Rare Glitch Project. <http://www-2.cs.cmu.edu/~rareglitch/> CMU 2001.

- [25] TTP/A Protocol. Specification of TTP/A Protocol. <http://www.tttech.com> TTTech Computertechnik AG 2000.
- [26] TTP/C Protocol. Specification of TTP/C Protocol. <http://www.tttech.com> TTTech Computertechnik AG 1999.
- [27] Kopetz, H.. “A Comparison of TTP/C and FlexRay”. Research Report. Institut fur Tevhnische Informatik. Technische Universitat Wien, Austria. 2001.
- [28] Comparison CAN Vs. Byteflight vs. TTP/C. <http://www.tttech.com> TTTech Computertechnik AG 2001.
- [29] [ISO 11898:1993](#) Road vehicles -- Interchange of digital information -- Controller area network (CAN) for high-speed communication.
- [30] Stankovic, J. A.. “Misconceptions About Real-Time Computing”. IEEE Computer, Vol. 21, N° 10, Oct. 1988, pp 10-19.
- [31] Patterson, D. A.; Hennessy, J. L.. Computer Arquitecture a Quantitive Approach. Morgan Kaufmann Publisher, inc. 1990.
- [32] Holub, A. I.. Compiler Design in C. Prentice Hall International, Inc. 1990.
- [33] HPC C Optimizing C Compiler User’s Manual. National Semiconductor, 1994.
- [34] HPC Assembler/Linker/Librarian User’s Manual. National Semiconductor, 1994.
- [35] National Semiconductor. Embedded Controller Databook, 1992.
- [36] Texas Instruments. TMS320LF/LC240x DSP Controllers System and Peripherals, 2000.
- [37] Nelson, V. P.; Carroll, B.D.. “Tutorial: Fault-Tolerant Computing”. IEEE Computer Society Press, Washington, D.C., 1986.
- [38] Tanenbaum, A. S.. Computer Networks. Printice Hall Inc. 1989.
- [39] Tanenbaum, A. S.. Modern Operating Systems. Printice Hall Inc. 1992.

- [40] Krishna, C. M.; Singh, A.D.. “Reliability of Checkpointed Real-Time Systems Using Time Redundancy”. IEEE Trans. on Reliability, 42(3): 427-435, Sept 1993.
- [41] Wensley, J.H.; Green, M. W; Levitt, K. N. Shostak, R. E.. “SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control”. Proceedings of the IEEE, 66(11): 1240-1255, Oct 1978.
- [42] Ramos-Thuel, S.. “Enhancing Fault Tolerance of Real-Time Systems through Time Redundancy”. Ph.D. Thesis, Carnegie Mellon University, 1993.
- [43] Associação Brasileira de Normas Técnicas. NBR ISO 9001, Sistemas da Qualidade - Modelo para garantia da qualidade em projeto, desenvolvimento, produção, instalação e serviços associados. ABNT 1994.
- [44] Garey, M. R.; Johnson, D. S.. “Computers and Intractability, a Guide to the Theory of NP-Completeness”. w. H. Freeman Company, San Francisco, 1979.
- [45] Lehoczky, J. P.. “Fixed Priority Scheduling of Periodic Task Sets with arbitrary deadlines”, Proceedings 11th IEEE Real-Time Systems Symposium (5-7 December 1990) pp. 201-209
- [46] Lehoczky, J. P.; Sha, L.; Ding, Y.. “The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour”. Proceedings of Real-Time Systems Symposium 1989.
- [47] Obenza, R.; Mendal, G. O.. “Guaranteeing Real Time Performance Using RMA”. The Embedded Systems Conference, San Jose, CA, 1998.
- [48] Ghosh, S.; Melhem, R. G.; Mossé, D.. “Enhancing Real-Time Schedules to Tolerate Transient Faults”. IEEE Real-Time Systems Symposium, pp.120-129, 1995.
- [49] Mossé, D.; Melhem, R. G.; Ghosh, S.. “Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm”. FTCS-24: 24<sup>th</sup> Symposium on Fault-Tolerant Computing, IEEE Computer Society Press, pp.16-25, 1004.
- [50] Sha, L.; Goodenough, J. B.. “Real-Time Scheduling Theory and Ada”. IEEE Computer, Vol 23, No. 4, Abril 1990, pp. 53-62 1990.



- [51] Audsley, N. C.; Burns, A.; Davis, R. I.; Tindell, K.; Wellings, A. J. "Fixed Priority Pre-emptive Scheduling: An Historical Perspective." *Real Time Systems* 8, 2-3 (March - May 1995): 173-98.
- [52] Sha, L.; Lehoczky, J. P.; Rajkumar, R.. 1987. "Priority Inheritance Protocols: An Approach To Real-Time Synchronisation". Computer Science Department, Carnegie-Mellon University, USA. CMU-CS-87-181.
- [53] Kopetz, H.; Kantz, H.; Grunsteidl, G.; Puschner, P. P.; Reisinger, J.. "Tolerating Transient Faults in MARS". In *Symp. On Fault Tolerant Computing(FTCS-20)*, pages 466-473. IEEE, 1990.
- [54] Hoyme, K.; Driscoll, K.. "SAFEbus™". In *1 Ith AIAA/IEEE Digital Avionics Systems Conference*, pages 68-73, Seattle, W A, October 1992.
- [55] Muller, B.; Fuhrer, T.; Hatwisch, F.; Hugel, R.; Weiler, H.. "Fault Tolerant TTCAN Networks". *Proceedings 8<sup>th</sup> International CAN Conference; 2002; Las Vegas.*
- [56] Hartwisch, F.; Futhrer, T.; Hugel, R.; Muller, B.. "Timing in the TTCAN Network". *Proceedings 8<sup>th</sup> Intenational CAN Conference; 2002, Las Vegas.*
- [57] Fuhrer, T.; Muller, B.; Dieterle, W.; Hatwisch, F.; Hugel, R.; Weiler, H.; Walther, M.. "Time Triggered Communication on CAN". *Proceedings 7<sup>th</sup> International CAN Conference; 2000.*
- [58] Rushby, J.. "Bus Architectures for Safety-Critical Embedded Systems". SRI International Computer Science Laboratory. 2001 Menlo Park USA.
- [59] Rushby, J.. "A Comparison of Bus Architectures for Safety-Critical Embedded Systems". SRI International Computer Science Laboratory. CSL Technical Report. 2001 Menlo Park USA.
- [60] Kopetz, H.. "Communication Protocols for Fault-Tolerant Distributed Real-Time Systems". 1994. Nortic Seminar on Dependable Computing, Technical University of Denmark, Lyngby, Denmark.
- [61] Multi-Transmitter Data Bus-Par 1: Technical Description, ARINC Specification 629-2, Aeronautical Radio, Inc, 2551 Riva Road, Annapolis, Maryland, 21401, October 1991

- [62] Controller Area Network CAN, an IN-Vehicle Serial Communication Protocol-SAE 11583, March 1990, 1992 SAE Handbook, pp.20341-20355
- [63] The PIP Protocol, Technical Report, World PIP Europe, 3 bis, rue de la Salpetriere, 54000 Nancy, France, 1994
- [64] Kopetz, H.; Griinsteidl, G.. "TTP- A Protocol for Fault-Tolerant Real-Time Systems". IEEE Computer, January 1994, pp. 14-23
- [65] LON Protocol Overview, Echelon Systems Corporation, 727 University Avenue, Los Gatos, California 95030
- [66] The Profibus Standard, Profibus Nutzerorganisation e.V., Hersler Strasse 31, D -50389 Wesseling, August 1992
- [67] Bauer, G.; Paulitsch, M.. "An investigation of membership and clique avoidance in TTP/C". In 19th Symposium on Reliable Distributed Systems, Nuremberg, Germany, October 2000.
- [68] Kopetz, H.. "A Solution to an Automotive Control System Benchmark". Institut fur Technische Informatik, Technische Universitat Wien, research report 4/1994 (April 1994)
- [69] Tindell, K.; Burns, A.; Wellings, A.. "Calculating Controller Area Network (CAN) Message Response Times". University of York, Department of Computer Science, York, England.
- [70] Tindell, K.; Burns, A.. "Guaranteeing Message Latencies on Control Area Network (CAN)". University of York, Department of Computer Science, York, England.
- [71] Tindell, K.. "Fixed Priority Scheduling of hard real-time systems". Ph. D. Thesis. University of York, Department of Computer Science, York, England. 1994.
- [72] Beckler, A. R.; Campos, S. V.. "Verificação formal de protocolos para sistemas de tempo real tolerantes a falhas". Universidade Federal de Minas Gerais, Brazil, 2003.
- [73] Clarke, E. M.; Grumberg, O.; Peled, D. A.. "Model Checking". MIT Press, Cambridge, MA, USA. 1999
- [74] Burch, J. R.; Clarke, E. M.; Long, D. E.. "Symbolic model checking with partitioned transition relations." VLSI 91, Edinburgh, Scotland, 1990.

- [75] Campos, S. V.; Clarke, E.; Minea, M.. "The Verus tool: a quantitative approach to the formal verification of real-time systems". Conference on Computer Aided Verification, 1997.
- [76] Chen, J. F.; C. L. Chu. "Combination Voltage-controlled and Current-controlled PWM Inverters for UPS Parallel Operation". IEEE Transactions on Power Electronics – September 1995, vol. 10, pp. 547-558.
- [77] Siri, K.; Lee, C. Q.. "Current Distribution Control of Converters Connected in Parallel". Proceedings of Industry Applications Society – IAS 1990, october 1990, vol. 2, pp. 1274-1280.
- [78] Lee, C. Q.; Siri, K.; Wu, T. F.. "Dynamic Current Distribution Controls of a Parallel Connected Converter System". IEEE – PESC 1991, june 1991, pp. 875-881.
- [79] Lee, C. S.; Kim, S.; Kim, C. B.; Hong, S. C.; Yoo, J. S.; Kim, S. W.; Kim, C. H.; Woo, S. H.; Sun, S. Y.. "Parallel UPS with a Instantaneous Current Shraring Control". IECON 1998, August 1998, vol1, pp. 568-573.
- [80] Wu, T. F.; Huang, Y. H.; Chen, Y. K.; Liu, R. Z.. "A 3C Strategy for Multi-module Inverters in Parallel Operation to Achieve na equal Current Distribution", May 1998, vol. 1, pp. 186-192.
- [81] Kawabata, T. et al.. "Large Capacity Parallel Redundant Transistor UPS". Int. Power Electronics Conference Rec – IPEC, vol. 1, pp. 660-671 – IEE of Japan.
- [82] Pfeiffer, O.. "Betting on CAN and CANopen". Embedded systems academy. <http://www.esacademy.com/faq/docs/bettingcan>.
- [83] Colin, A.; Puaut, I. A modular and retargetable framework for tree-based WCET analysis Real-Time Systems, 13th Euromicro Conference on, 2001., Vol., Iss., 2001 pp.37-44
- [84] Heckmann, R.; Langenbach, M.; Thesing, S.; Wilhelm, R. The influence of processor architecture on the design and the results of WCET tools. Proceedings of the IEEE, Vol.91, Iss.7, July 2003 pp. 1038- 1054.
- [85] Puschner, P.; Bernat, G. WCET analysis of reusable portable code Real-Time Systems, 13th Euromicro Conference on, 2001., Vol., Iss., 2001 pp.45-52.