Universidade do Minho

Escola de Engenharia

Francisco Carlos Afonso

Operating System Fault Tolerance Support for Real-Time Embedded Applications

Tese de Doutoramento em Electrónica Industrial

Área de Conhecimento: Informática Industrial

Trabalho efectuado sob a orientação de:

Professor Doutor Adriano José da Conceição Tavares

Professor Doutor Carlos Alberto Batista da Silva

Doutor Sergio Montenegro-Retana

Janeiro de 2009

Acknowledgements

I would like to thank my supervisors at University of Minho, Prof. Dr. Adriano Tavares and Prof. Dr. Carlos Silva, for their continuous support and guidance during the four years of this work. I am grateful for their constant availability to discuss the thesis direction, for helping me with difficult topics, and for providing every resource I needed to carry on this work.

To Dr. Sergio Montenegro, my supervisor at FIRST, and now at DLR, I would like to thank the kindness and hospitality regarding my internship at FIRST. I am also grateful for the several emails he answered me explaining the BOSS mechanisms and proposing solutions to the problems I faced.

I also would like to thank my PhD colleagues at the Department of Industrial Electronics, José Carlos Metrôlho, Sergio Lopes e Paulo Cardoso, for their advice and support to my research. I wish we could have worked closely during our research time.

I am also grateful for the financial support provided by *Fundação para a Ciência e a Tecnologia*, which has sponsored my scholarship, my internship at FIRST, and the presentation of my work in several conferences and workshops. I also thank University of Minho and the Department of Industrial Electronics, for providing and adequate work environment and for financing the acquisition of the required hardware.

To my brothers João Luiz and José Augusto, I would like to thank for their encouragement and assistance from the thesis proposal to the thesis review.

Everything I did would be much more difficult if it weren't for my parents, Francisco e Cândida, who received us home for more than six months and gave us lots of care and love during these four years.

Last but not least, I wish to thank to my beloved wife Helena and my dear kids Mariana and Carlos, for being always beside me during this hard endeavor. A PhD

research can be a very lonely task, and I am sure I would not have finished it if I did not have my caring family at home. To them I dedicate this thesis.

Operating System Fault Tolerance Support for Real-Time Embedded Applications

Abstract

Fault tolerance is a means of achieving high dependability for critical and high-availability systems. Despite the efforts to prevent and remove faults during the development of these systems, the application of fault tolerance is usually required because the hardware may fail during system operation and software faults are very hard to eliminate completely.

One of the difficulties in implementing fault tolerance techniques is the lack of support from operating systems and middleware. In most fault tolerant projects, the programmer has to develop a fault tolerance implementation for each application. This strong customization makes the fault-tolerant software costly and difficult to implement and maintain. In particular, for small-scale embedded systems, the introduction of fault tolerance techniques may also have impact on their restricted resources, such as processing power and memory size.

The purpose of this research is to provide fault tolerance support for real-time applications in small-scale embedded systems. The main approach of this thesis is to develop and integrate a customizable and extendable fault tolerance framework into a real-time operating system, in order to fulfill the needs of a large range of dependable applications. Special attention is taken to allow the coexistence of fault tolerance with real-time constraints. The utilization of the proposed framework features several advantages over ad-hoc implementations, such as simplifying application-level programming and improving the system configurability and maintainability.

In addition, this thesis also investigates the application of aspect-oriented techniques to the development of real-time embedded fault-tolerant software. Aspect-Oriented Programming (AOP) is employed to modularize all fault tolerant source

code, following the principle of separation of concerns, and to integrate the proposed framework into the operating system.

Two case studies are used to evaluate the proposed implementation in terms of performance and resource costs. The results show that the overheads related to the framework application are acceptable and the ones related to the AOP implementation are negligible.

Suporte do Sistema Operativo à Tolerância a Falhas em Aplicações Embebidas de Tempo-Real

Resumo

Tolerância a falhas é um meio de obter-se alta confiabilidade para sistemas críticos e de elevada disponibilidade. Apesar dos esforços para prevenir e remover falhas durante o desenvolvimento destes sistemas, a aplicação de tolerância a falhas é normalmente necessária, já que o hardware pode falhar durante a operação do sistema e falhas de software são muito difíceis de eliminar completamente.

Uma das dificuldades na implementação de técnicas de tolerância a falhas é a falta de suporte por parte dos sistemas operativos e *middleware*. Na maioria dos projectos tolerantes a falhas, o programador deve desenvolver uma implementação de tolerância a falhas para cada aplicação. Esta elevada adaptação torna o software tolerante a falhas dispendioso e difícil de implementar e manter. Em particular, para sistemas embebidos de pequena escala, a introdução de técnicas de tolerância a falhas pode também ter impacto nos seus restritos recursos, tais como capacidade de processamento e tamanho da memória.

O propósito desta tese é prover suporte à tolerância a falhas para aplicações de tempo real em sistemas embebidos de pequena escala. A principal abordagem utilizada nesta tese foi desenvolver e integrar uma *framework* tolerante a falhas, customizável e extensível, a um sistema operativo de tempo real, a fim de satisfazer às necessidades de uma larga gama de aplicações confiáveis. Especial atenção foi dada para permitir a coexistência de tolerância a falhas com restrições de tempo real. A utilização da *framework* proposta apresenta diversas vantagens sobre implementações ad-hoc, tais como simplificar a programação a nível da aplicação e melhorar a configurabilidade e a facilidade de manutenção do sistema.

Além disto, esta tese também investiga a aplicação de técnicas orientadas a aspectos no desenvolvimento de software tolerante a falhas, embebido e de tempo real. A Programação Orientada a Aspectos (POA) é empregada para segregar em

módulos isolados todo o código fonte tolerante a falhas, seguindo o princípio da separação de interesses, e para integrar a *framework* proposta com o sistema operativo.

Dois casos de estudo são utilizados para avaliar a implementação proposta em termos de desempenho e utilização de recursos. Os resultados mostram que os acréscimos de recursos relativos à aplicação da *framework* são aceitáveis e os relativos à implementação POA são insignificantes.

Contents

Acknowledgements ..... **iii**

Abstract ..... **v**

Resumo ..... **vii**

Contents ..... **ix**

Figures ..... **xv**

Tables ..... **xix**

List of Abbreviations ..... **xxi**

Chapter 1 ..... **1**

Introduction **1**

1.1 Motivation 2

1.2 Problem statement 4

1.3 Approach and contributions 5

1.4 Thesis organization 7

Chapter 2 ..... **9**

Fault tolerance **9**

2.1 Faults, errors and failures 10

2.2 Dependability and fault tolerance 12

2.3 Basic techniques in fault tolerance 14

2.4 Redundancy 16

2.5 Design diversity 16

2.6 Hardware fault tolerance 17

2.7 Software fault tolerance 20

2.8 Fault tolerance strategies 21

2.8.1 Checkpoint and Restart 21
2.8.2 Recovery Blocks 22
2.8.3 Distributed Recovery Blocks 23
2.8.4 N-Version Programming 26
2.9 Fault-tolerant communication 27
2.10 Fault tolerance software structures 29
2.11 Fault tolerance application support 33
2.11.1 FT-RT-Mach and DEOS 33
2.11.2 Delta-4 34
2.11.3 TMOSM and ROAFTS 35
2.11.4 Adaptive Fault Tolerance for Spacecraft 37
2.11.5 Fault Tolerant CORBA 38
2.12 Summary 39
Chapter 3 41
Aspect-Oriented Programming 41
3.1 Separation of concerns 42
3.1.1 Meta-level Programming 43
3.1.2 Composition Filters 43
3.1.3 Aspect-Oriented Programming 44
3.2 AspectC++ 45
3.2.1 Weaving 46
3.2.2 Join points, pointcuts and advices 47
3.2.3 The JoinPoint API 49
3.2.4 Performance and memory footprint 51
3.3 AOP for the operating system 52

3.4 AOP for the middleware 53
3.5 Fault tolerance using AOP 55
3.6 Summary 58
Chapter 4 59
BOSS operating system 59
4.1 Introduction 60
4.2 Kernel services 62
4.2.1 Task processing 62
4.2.2 Synchronization 64
4.2.3 Communication 66
4.2.4 Utility classes 68
4.2.5 Hardware interface and management 68
4.3 Middleware services 70
4.3.1 Message to message communication 71
4.3.2 Message to thread communication 73
4.4 Middleware extensions 75
4.4.1 Message identification and discarding 76
4.4.2 External messages handling 77
4.5 Summary 80
Chapter 5 81
Fault tolerance framework 81
5.1 Introduction 82
5.2 Fault tolerance thread model 83
5.3 Framework general description 84
5.3.1 Framework structure 85

5.3.2 Fault tolerance introduction 86
5.3.3 Application-specific entities 87
5.4 Framework general implementation 90
5.4.1 Timing behavior 90
5.4.2 Class structure 92
5.5 FT strategies implementation 95
5.5.1 Recovery Blocks strategy 95
5.5.2 Distributed Recovery Blocks strategy 99
5.5.3 N-Version Programming strategy 105
5.6 Discussion 114
5.7 Summary 115
Chapter 6 117
Applying AOP for fault tolerance 117
6.1 Application-level fault tolerance 118
6.1.1 Code generation 119
6.1.2 AspectC++ restriction 120
6.1.3 AOP implementation 122
6.1.4 Discussion 127
6.2 FT framework integration 128
6.2.1 Code generation 129
6.2.2 AOP implementation 131
6.2.3 Discussion 132
6.3 Operating system fault tolerance 133
6.3.1 Semaphore error detection 134
6.3.2 Code generation 135

6.3.3 AOP implementation 135
6.3.4 Discussion 139
6.4 Project configuration using AOP 139
6.5 Summary 141
Chapter 7 143
Case studies and evaluation 143
7.1 Development and test environment 144
7.1.1 Target systems 144
7.1.2 Host system 145
7.1.3 Porting BOSS to the target board 146
7.2 Case study I: sorting application 146
7.2.1 Testing configurations 146
7.2.2 Execution time measurements 149
7.2.3 Runtime costs 153
7.2.4 Memory costs 157
7.2.5 FT scheduling tests 159
7.3 Case study II: radar filtering application 161
7.3.1 Testing configurations 162
7.3.2 Fault tolerance implementations and testing 165
7.3.3 AOP implementations 165
7.3.4 Runtime costs 166
7.4 Evaluation 167
7.4.1 FT framework 167
7.4.2 Aspect-oriented implementation 168
7.5 Summary 169

Chapter 8 ..... **171**

Conclusions **171**

8.1 Conclusions 172

8.2 Future work 174

Bibliography **177**

Figures

Figure 2.1: Faults, errors and failures. 10

Figure 2.2: Means to achieve dependable systems. 13

Figure 2.3: Triple Modular Redundancy. 18

Figure 2.4: Self-checking modules. 19

Figure 2.5: RB execution. 22

Figure 2.6: DRB execution. 24

Figure 2.7: Xu,Randell, Rubira-Calsavara and Stroud´s framework example. 29

Figure 2.8: Tso, Shokri, Tai and Dziegiel´s class diagram for the RB technique. 30

Figure 2.9: Reliable Hybrid pattern class diagram. 31

Figure 2.10: The Generic Software Fault Tolerance pattern class diagram. 32

Figure 3.1: Code tangling and code scattering. 42

Figure 3.2: AspectC++ weaving process. 46

Figure 3.3: AspectC++ program example. 48

Figure 3.4: Example of source code transformation by AspectC++. 50

Figure 4.1: Task processing related classes. 62

Figure 4.2: Thread states. 63

Figure 4.3: Synchronization related classes. 65

Figure 4.4: Communication related classes. 67

Figure 4.5: BOSS utility classes. 68

Figure 4.6: BOSS basic architecture. 69

Figure 4.7: Kernel/HDL interface. 70

Figure 4.8: *Message* class diagram. 71

Figure 4.9: *NameServer* data structure. 72

Figure 4.10: Middleware message distribution. 72

Figure 4.11: *IncommingMessageAdministrator* class diagram. 74

Figure 4.12: *IncommingMessageAdministrator* sequence diagram. 74

Figure 4.13: TMR configuration. 75

Figure 4.14: *NameServer* extension for discarding duplicate messages. 76

Figure 4.15: Middleware extensions class diagram. 77

Figure 4.16: External messages processing. 79

Figure 4.17: External message packet description. 79

Figure 5.1: Fault tolerance thread model. 83

Figure 5.2: Example of candidate thread for FT implementation. 84

Figure 5.3: Simplified FT framework class diagram. 85

Figure 5.4: Example of FT application thread. 87

Figure 5.5: RB execution timing example. 91

Figure 5.6: RB execution activity diagram. 92

Figure 5.7: FT strategy execution class diagram. 92

Figure 5.8: *MiddewareScheduler* thread sequence diagram. 94

Figure 5.9: RB strategy class diagram. 96

Figure 5.10: RB strategy execution. 97

Figure 5.11: RB timing example. 98

Figure 5.12: Stateless RB threads configuration example. 99

Figure 5.13: DRB strategy class diagram. 100

Figure 5.14: DRB strategy execution. 101

Figure 5.15: DRB strategy configuration example. 104

Figure 5.16: DRB timing diagram. 104

Figure 5.17: NVP strategy class diagram. 106

Figure 5.18: NVP strategy execution. 107
Figure 5.19: NVP state initialization example. 108
Figure 5.20: Voting configurations. 109
Figure 5.21: *VoterThread* class diagram. 110
Figure 5.22: Voting execution diagram. 112
Figure 5.23: Master election state diagram. 113
Figure 5.24: Master voter failure worst scenario. 114
Figure 6.1: Example of thread source code before fault tolerance introduction. 118
Figure 6.2: Example of thread source code after fault tolerance introduction. 119
Figure 6.3: Code generation process using AOP at the application level. 120
Figure 6.4: AspectC++ base class introduction example. 121
Figure 6.5: FT framework modified for AOP application. 122
Figure 6.6: DRB strategy abstract aspect. 123
Figure 6.7: DRB strategy concrete aspect example. 124
Figure 6.8: NVP strategy with StdVoter abstract aspect. 125
Figure 6.9: NVP strategy with *StdVoter* concrete aspect example. 126
Figure 6.10: Code generation process when using AOP at the OS level. 129
Figure 6.11: Alternative code generation process with double weaving. 130
Figure 6.12: MiddlewareScheduler activation aspect. 131
Figure 6.13: Aspect for introducing FT attributes and methods in the *Thread* class. 132
Figure 6.14: Semaphore error detection aspect for the first predicate. 135
Figure 6.15: Semaphore error detection aspect for the second predicate. 137
Figure 6.16: Synchronization aspect applied to the *Semaphore* class. 138
Figure 6.17: *Semaphore* enter method sequence diagram. 139
Figure 7.1: Test environment. 145

Figure 7.2: Case study I - non-FT or RB configuration. 147

Figure 7.3: Case study I - DRB configuration. 147

Figure 7.4: Case study I - NVP configuration. 148

Figure 7.5: Case study I - no-failure condition. 151

Figure 7.6: Case study I – comparison between no-failure and variant 1 failure conditions. 152

Figure 7.7: Case study I - comparison between no-failure and one node failure conditions. 153

Figure 7.8: Case study I - CPU utilization configuration. 154

Figure 7.9: Case study I - CPU utilization results. 155

Figure 7.10: CPU utilization comparison for AOP and FT scheduling versions..... 156

Figure 7.11: Comparison of FT and non-FT memory footprints. 158

Figure 7.12: Case study I - detailed comparison of memory footprint. 158

Figure 7.13: Case study II - non-FT configuration. 162

Figure 7.14: Case study II - PSP configurtion. 162

Figure 7.15: Case study II - TMR configuration. 163

Figure 7.16: Case study II - display output example. 164

Figure 7.17: Case study II – CPU utilization results. 166

Tables

Table 3.1: AspectC++ pointcut functions. 48

Table 5.1: Multiple version strategies requirements. 88

Table 5.2: Single version strategies requirements. 88

Table 5.3: Voter requirements. 89

Table 7.1: Case study I - local execution time. 149

Table 7.2: Case study I - total execution time. 150

Table 7.3: Case study I - time settings. 150

Table 7.4: Memory footprint results. 157

Table 7.5: Additional memory costs for AOP and FT scheduling versions. 159

Table 7.6: FT scheduling test configurations. 160

Table 7.7: FT scheduling test results. 161

List of Abbreviations

AFT	Adaptive Fault Tolerance
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
API	Application Programming Interface
AT	Acceptance Test
CORBA	Common Object Request Broker Architecture
CPM	Communication Processor Module
DRB	Distributed Recovery Blocks
EJB	Enterprise Java Beans
FIFO	First In First Out
FT	Fault Tolerance or Fault-Tolerant
FTRMS	Fault-tolerant RMS
GSFT	Generic Software Fault Tolerance
HDL	Hardware Dependent Layer
I^2C	Inter-Integrated Circuit
ISR	Interrupt Service Routine
MIPS	Million Instructions per Second
MOP	Meta-object Protocol
MS	MiddlewareScheduler
NMR	N-Modular Redundancy
NVP	N-Version Programming
NAC	Network Attachment Controller
OOP	Object-Oriented Programming

OS	Operating System
OSI	Open Systems Interconnection
PC	Personal Computer
PCMCIA	Personal Computer Memory Card International Association
PSP	Pair of Self Checking Processors
RB	Recovery Blocks
RISC	Reduced Instruction Set Computing
RMI	Remote Method Invocation
RMS	Rate Monotonic Scheduling
ROAFTS	Real-Time Object-Oriented Adaptive Fault Tolerance Support
RPC	Remote Procedure Call
RPM	Revolutions per Minute
SIU	System Interface Unit
SPI	Serial Peripheral Interface
STU	Single Translation Unit
TMO	Time-triggered Message-triggered Object
TMOSM	TMO Support Middleware
TMR	Triple Modular Redundancy
TTP	Time-triggered Protocol
WPT	Whole Program Transformation
WTST	Watchdog Timer and Scheduler Thread

Chapter 1

Introduction

This chapter initially describes the thesis motivation and the main topics related to this work. The definition of the research problem and the formulation of the research questions are addressed next. Finally, the approach and contributions of this work are stated.

1.1 Motivation

Embedded systems have a widespread use in several domains, such as consumer electronics, home/office automation, and the automotive industry. A precise definition of the term *embedded system* does not exist. In general, embedded systems are defined as hardware-software systems that perform a specific function, usually being part of a larger system, which explains the "embedded" denomination. Besides being designed to execute a predefined function, as opposed to a general purpose computing system (mainframe, desktop, notebook, and so on), embedded systems usually have a particular method of software development called cross-platform development [96], in which the software is generated in other platform and then it is transferred to the embedded device.

Most embedded systems have to react to the system environment in a timely fashion. Real-time systems must satisfy timing constraints, and therefore the correct response depends also on the time that it is produced. Examples of real-time embedded systems include portable media players and control systems. The consequences of not satisfying a timing constraint are severe in hard real time systems, in contrast with soft real time systems, in which there is some degree of tolerance to timing violations.

Some embedded systems demand high reliability, availability or safety, as a system failure may endanger human lives or compromise the success of the entire system operation. These are classified as safety-critical and mission-critical systems, respectively. Examples of these critical systems include drive-by-wire systems in automobiles, fly-by-wire systems in avionics, missile control systems and autonomous space systems.

Critical systems are also termed high-dependability systems. Dependability is a wider concept that includes several attributes, such as reliability, safety, maintainability and security. The reliability of high-dependability systems can be several orders of magnitude higher than for commercial systems. For instance, civil transport airplane critical equipments are designed to have less than 10^{-9} catastrophic failures per hour of operation (a failure in 114 thousand years) [71]. Similar requirements are applied in railway control systems. High-dependability systems are

also needed in satellites and space missions because most of these systems must operate without any maintenance at all.

As critical embedded systems are composed by hardware and software, there is a strong need to reduce the number of failures related to these two domains. Hardware reliability has been constantly increasing over time. However, transient and permanent hardware faults may still occur, especially in environments subjected to high energy particles and radiation, such as space systems. In relation to software faults, the ever increasing functionality of the computer systems has a direct impact in the software complexity, which is the main cause of design faults in software. Despite the efforts taken at the several phases of software development, including the testing phase, various software faults are likely to remain unpredicted and undetected. Therefore, fault tolerance (FT) techniques are needed in order to maintain the system operational in the presence of hardware and software faults.

Several fault tolerance techniques have been proposed in the last 30 years. However, the application of these techniques is expensive, in terms of resources and costs, and therefore they are normally only used in safety or mission-critical systems.

Fault tolerance is usually applied by means of redundancy and diversity. Redundant hardware implies the establishment of a distributed system executing a set of fault tolerance strategies by software, and may also employ some form of diversity, by using different variants or versions for the same processing. Redundant hardware involves extra software coordination, which makes the software system more complex and error-prone. Software fault tolerance may be implemented by software re-execution or multiple versions techniques, which also requires the application of additional control mechanisms.

In many fault tolerant projects, the programmer has to address both application-dependent and fault tolerance related concerns. This strong customization requires highly specialized design teams, thus making realistic fault-tolerant software costly and difficulty to implement and maintain. Therefore, there is an urgent need to provide a flexible support for fault-tolerant applications that is able to deliver some degree of transparency for the application developer and at the same time that

facilitates customizability across a broad range of applications, as well as diverse reliability requirements.

One of the difficulties in implementing fault tolerance techniques is the lack of support from operating systems and middleware. Operating systems are not designed with fault tolerance support in mind and even those that were extended to include some basic fault tolerance mechanisms did not provide support for a full fault tolerant implementation. The same happens to middleware implementations, such as CORBA [90], which were meant originally to solve the distribution problem, and only a few years ago have specified basic mechanisms of fault tolerance [88].

Another problem regarding the fault tolerance implementation is that it has a huge impact in the real-time behavior of an application. A fault tolerance implementation normally demands additional computations for fault detection, alternative implementations and replica coordination. These mechanisms change the application timing behavior and often violate real-time constraints. As an example of this issue, it can be mentioned the incompatibility of the FT-CORBA [88] and RT-CORBA [89] specifications [48, 85].

In particular, for small-scale embedded systems, the introduction of fault tolerance techniques may have impact on the restricted resources of these systems, such as processing power, memory size, power consumption, physical size and weight. These restrictions are considered in the requirements of many embedded projects, such as satellite systems. Most fault tolerance research developed so far focus on large-scale systems with no resource constraints, such as navy command and control systems and airline reservation systems. Most solutions proposed to that kind of systems are not applicable to small-scale embedded systems.

1.2 Problem statement

The purpose of this research is to provide fault tolerance support for real-time embedded applications by extending a real-time operating system. The focus of this research is on small-scale distributed embedded systems connected by local area networks or field buses. The emphasis of fault tolerance is on the computation (fault-

tolerant computing) and not in the communication between nodes, which is assumed to be reliable.

The main research questions are:

- Is the approach described above feasible and acceptable in terms of performance and resource costs?
- What benefits and drawbacks this approach brings to the embedded software development process?
- Can Aspect-Oriented Programming (AOP) [62], a new technique for advanced separation of concerns [36, 91], be applied at the operating system and application level to support the implementation of embedded fault-tolerant systems? If so, what are the benefits?

The operating system employed in this research was the BOSS operating system [81], developed by Fraunhofer Institute for Computer Architecture and Software Technology (FIRST). This operating system was written in C++, uses object-oriented technology extensively, and it includes a middleware for communication support based on a publish-subscriber protocol. The BOSS operating system targets real-time high-dependability applications, such as satellite and medical systems.

1.3 Approach and contributions

The main approach taken in this research was to develop and integrate a customizable and extendable fault tolerance framework into a real-time operating system, in order to fulfill the needs of a large range of dependable applications. This FT framework defines a set of collaborations between operating system basic classes and fault tolerance support classes in order to implement fault tolerance techniques with maximum transparency the application-level threads. Additionally, AOP was employed to provide a full modularization of the fault tolerance implementation.

The contributions of this research are listed as follows:

- The proposal of a framework for developing real-time embedded fault-tolerant software. In contrast with previous works, we target the application thread level, based on a thread model which allows both state and stateless threads.
- The development of several fault tolerance strategy implementations using the proposed framework in order to cover a wide range of fault tolerance requirements, supporting both hardware and software fault tolerance.
- The development of new mechanisms for the BOSS middleware, namely for message identification, duplicate messages discarding and external messages handling.
- The application of aspect-oriented techniques to the development of real-time embedded fault-tolerant software. In contrast with previous works, we applied AOP in order to provide fault tolerance to application threads. Additionally, we employed AOP to integrate the proposed FT framework into the original operating system and to implement fault tolerance mechanisms at the operating system level.
- The evaluation and comparison of the proposed fault tolerance framework and the AOP implementation in terms of performance and resource costs based on two case studies: a sorting application and a radar filtering system. These case studies were developed using a PowerPC 823 based target board, in a similar configuration employed in a satellite computer system. Performances based on execution time, plus costs related to runtime overhead and memory footprint were measured for several FT configurations and implementations.
- The evaluation of the proposed framework and the AOP implementation in terms of benefits to the embedded software development process, including maintenance and reusability issues.

The approaches and contributions described in this thesis have been succinctly presented in research papers published by international conferences and workshops related to real-time systems, industrial embedded systems and aspect-oriented software development [2-6].

1.4 Thesis organization

This thesis is divided in eight chapters. The remaining chapters are described as follows:

- **Chapter 2** introduces the main definitions and concepts related to fault tolerance. It also presents the fault tolerance techniques applied in this work and reviews the related work about fault tolerance.
- **Chapter 3** presents the main concepts related to Aspect-Oriented Programming, describes the AspectC++ language extension and reviews the research results regarding the application of AOP in operating systems, middleware and fault-tolerant systems.
- **Chapter 4** describes the main features of the BOSS operating system, including its kernel and middleware. A brief introduction about BOSS principles, history and applications is presented, followed by a detailed description of the kernel and the middleware. The middleware extensions developed for handling external messages are also described.
- **Chapter 5** describes the fault tolerance framework developed for supporting application-level fault tolerance, as an extension to the BOSS operating system and its middleware. The framework objectives and constraints are presented, as well as the thread model for FT introduction. The implemented fault tolerance strategies are described in detail. This chapter also discusses the benefits and drawbacks of the proposed FT framework.
- **Chapter 6** presents how AOP was applied to support the implementation of fault tolerance. It covers the application of AOP for three different purposes: (1) modularize the fault tolerance code at the application level; (2) integrate the FT framework into the operating system; and (3) implement fault tolerance at the operating system level. This chapter also discusses the benefits and drawbacks of the AOP application.
- **Chapter 7** presents the development and test environment applied in this work and describes the case studies developed to test the proposed FT framework, comparing performance and costs of several configurations and implementations.

- **Chapter 8** concludes this thesis and indicates possible future directions for this research topic.

Chapter 2

Fault tolerance

This chapter introduces the main definitions and concepts related to fault tolerance. Besides, the main techniques and approaches to build fault-tolerant systems are presented, as well as the related work regarding fault tolerance.

2.1 Faults, errors and failures

In this section, the basic terminology in fault tolerance is introduced by explaining the difference between faults, errors and failures. These terms are frequently combined with others to classify fault tolerance concepts and techniques, and therefore a precise definition of these terms is required[1].

A **failure** is an event that ocurrs when a system's delivered service deviates from the correct service [20]. The correct service is the one described in the system specification. An **error** is a part of the system state that may cause a subsequent failure. A **fault** is the cause of an error.

Figure 2.1 shows the relationship between faults, error and failures in a multi-component system [117]. A fault is active when it produces an error, otherwise it is dormant. A dormant fault may be activated (generates an error) after a system input or computational process. The failure of a component represents a fault for the system, and it can further generate a system error. Errors can propagate within a component or system. An error that has not been detected is a latent error. A system failure occurs when the error propagates to the system interface. In summary, a fault is a defect, an error is a corrupted state, and a failure is the event that we want to avoid.

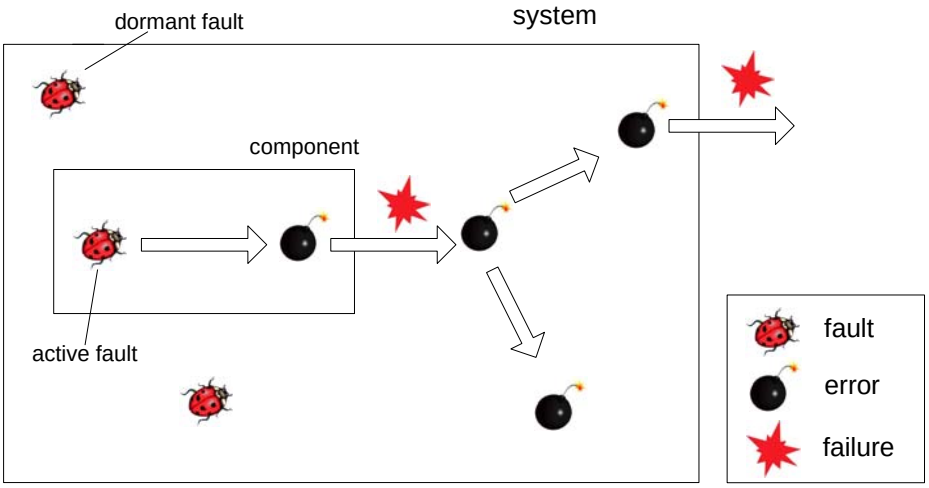

Figure 2.1: Faults, errors and failures.

[1] For instance, error detection and error handling have a completely different meaning than fault detection and fault handling.

Faults can be classified according to many criteria. In relation to the domain, there are hardware or software faults. Design faults occur much more frequently in software than in hardware because of the difference in complexity of these two domains. This difference is explained by the fact that hardware machines have usually a smaller number of internal states than software programs [98].

In relation to persistence, faults can be classified in permanent or transient. Hardware faults can be permanent or transient, but a software fault is always permanent. Apparent transient software faults are in fact permanent software faults with complex activation patterns. The ability to identify the activation pattern of a fault determines the fault activation reproducibility. Faults can be categorized according their activation reproducibility as solid (or hard, or bohrbugs [49]), and elusive (or soft, or heisenbugs [49]). The activation of elusive faults is not systematically reproducible. Elusive faults activation can depend, for instance, on unusual combinations of internal states and external requests, system load, and timing. Most residual design faults in large and complex software are elusive faults. The similarity of the manifestation of elusive development faults and of transient physical faults leads to both classes being grouped together as intermittent faults. Errors produced by intermittent faults are termed soft errors [20].

Failures can be classified in relation to the domain as content failures and timing failures. Content failures, also called value failures, present a deviation in the content of the information delivered by a system in regard to the system specification. In timing failures, the deviation is related to the arrival or duration of the information delivery. A failure can also be consistent or inconsistent. Consistent failures are perceived identically for all system users, while inconsistent failures are perceived differently by one or more users. Inconsistent failures are also called Byzantine failures.

2.2 Dependability and fault tolerance

The dependability of a computer system is the ability to avoid system failures that are more frequent and more severe than acceptable [20]. The concept of dependability is strong connected with the concept of trust, and comprises the following attributes:

- **Reliability**: continuity of the correct service. Reliability is the probability that a system will perform its intended function satisfactorily, for a specified period of time. It is usually expressed in terms of failure rate (λ), or its inverse, the mean time to failure (MTTF) [71]. The system reliability is dependent on the system environment. For instance, the activation of some types of faults can be triggered by specif input sequences [111]. A system can have many faults but still be reliable if the environment does not trigger any fault activation in its normal operation.
- **Availability**: readiness for correct service. Availability is the probability that a system is performing its required function at a given point in time. To calculate the system availability it is necessary to include information about the mean time to repair (MTTR).
- **Safety**: absence of catastrophic consequences on the user and the environment. A fail-safe system is one that cannot cause harm when it fails. A system can be fail-safe but unreliable and vice-versa. For many systems, the fail-safe property cannot be guaranteed as, for instance, in airplane flight control systems [108]. Safety can also be defined as the reliability with respect to catastrophic failures.
- **Confidentiality**: absence of unauthorized disclosure of information.
- **Integrity**: absence of improper system state alterations.
- **Maintainability**: the ability to undergo repairs and modifications.

There are four basic means to achieve dependability: fault prevention, fault removal, fault forecasting and fault tolerance [20]. These techniques are described as follows:

- **Fault prevention**: to avoid or prevent the introduction of faults in the system design. Examples of software fault prevention include software design methods,

modularization and reusability. Many design faults are introduced because of an incorrect or incomplete system specification.

- **Fault removal**: to detect and eliminate faults from system, both at development and operational phases. It includes verification, diagnosis and correction. Verification can be static, using for instance inspections and formal methods, or dynamic, with the application of fault injection and testing.
- **Fault forecasting**: to predict and estimate the presence and activation of faults as well as their consequences. Fault forecasting techniques include failure mode and effects analysis (FMEA), Markov chains and fault-trees. Fault forecasting techniques may indicate the need for modifications in system design and the application of fault tolerance.
- **Fault tolerance**: to preserve the delivery of a correct system service in the presence of active faults. Fault tolerance is intended to prevent active faults from becoming failures. In order to achieve fault tolerance, the system must react to errors before they reach its boundaries.

Figure 2.2 shows the relationship among the four means to achieve dependable systems. As represented in this figure, faults may be still present after system development and validation, when fault prevention and fault removal techiques are applied. The remaining faults must be taken care at operation time, by using fault tolerance techniques. Fault forecasting may be applied in all phases of the system lifecycle, using both prediction and estimation techniques regarding faults and failures.

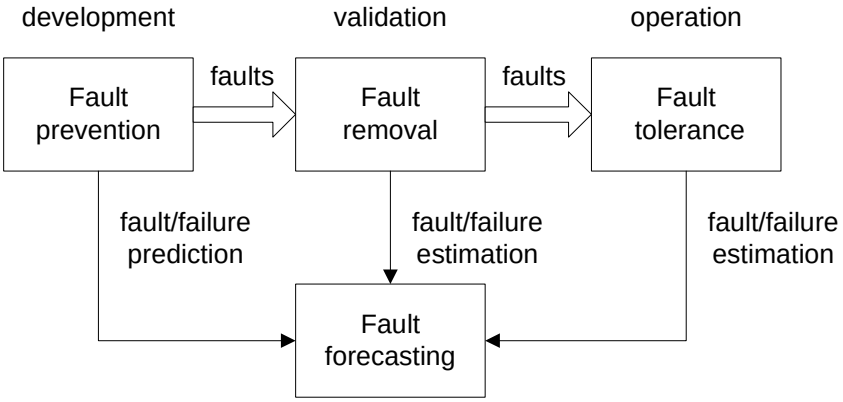

Figure 2.2: Means to achieve dependable systems.

In particular, software design faults are very hard to eliminate completely through fault prevention and removal. Besides, hardware faults, either permanent or transient, may happen during system operation. Therefore, only fault tolerance can cope with software residual faults and hardware operational faults.

Fault tolerance is directly related to system reliability, as its application increases the time between failures. Increasing system reliability will also result in larger system availability and safety.

Fault-tolerant systems may be classified as follows [104, 111]:

- **Critical systems**: require a high degree of reliability and safety. This category includes safety-critical systems, in which a failure can cause loss of lives, and mission-critical systems, in which a failure can cause damage in equipment, or the loss of efforts and the mission failure. Some safety-critical systems examples are flight control systems, nuclear plants and railway control systems. Commercial fly-by-wire systems, for instance, require a probability of failure per hour not greater than 10^{-9}, considered as ultra-high reliability [71].
- **Long life systems**: require that a computer operates as intended when the time between maintenance is large or even without any maintenance at all. This includes, for instance, satellites and space systems.
- **High-availability systems**: demand a very high probability that the system will be ready to provide the intended service when required, such as airline reservation systems.
- **General purpose systems**: are the less demanding in terms of fault tolerance, generally providing only error detection capabilities.

2.3 Basic techniques in fault tolerance

Fault tolerance is implemented by means of error detection and system recovery. Error detection aims to spot errors within the system. Several methods may be applied to detect errors, such as replication checks, timing checks, reasonableness checks and structural checks [57]. System recovery must apply **error handling** for eliminating

the error from the system state and, additionally, may apply **fault handling** for diagnosing the fault and preventing it from being activated again. There are three general techniques for **error handling**: rollback, rollforward and compensation.

In the **rollback** technique, also called backward recovery, the system is restored to a previous assumed error-free state. This technique requires that the system state is stored periodically in predetermined recovery points, in a process called checkpointing. It is effective against transient faults because these faults may have disappeared after restarting from the last checkpoint. For permanent faults, the use of rollback mechanisms must be associated with other techniques as, for instance, changing the algorithm in case of software faults.

In the **rollforward** technique, also called forward recovery, the system is taken to a new state without errors. Using this technique, the system tries to make corrective actions to remove the error from the system state. Therefore, it requires precise information about the error nature and extent. This diagnosis is application and system dependent.

In the **compensation** technique, the erroneous state contains enough redundant information to enable error elimination. Corrections codes such as Hamming code and multiple executions of the same computation are examples of error compensation. rror Compensation does not depend on error detection, and so it can be executed continuously. This form of recovery is called **fault masking**. Alternatively, compensation can be executed only after some error detection.

Error handling techniques eliminate errors from the system state, but they do not prevent new errors from occurring. For this reason, **fault handling** is needed. Fault handling involves four steps:

- **Fault diagnosis**: identifies the fault type and location.
- **Fault isolation**: performs physical or logic exclusion from future participation in service delivery.
- **System reconfiguration**: switches to a spare component or task.
- **System reinitialization**: updates system state and configuration information.

Some components and systems are designed to fail only in specific modes that preserve safety (fail-safe) or that do not produce incorrect results that may affect further processing (fail-silent). Additionally, a system can be designed to provide a degraded functionality in case of failure, returning to full functionality after a system reconfiguration and reinitialization. These systems are termed failure-controlled systems.

2.4 Redundancy

Fault tolerance implementation depends heavily on redundancy. Redundancy is the utilization of additional resources that are not required for normal system operation.

Hardware redundancy includes replicated and supplementary hardware to support fault tolerance, and is the most used form of redundancy in fault-tolerant systems. Software redundancy includes additional programs, modules and objects to support fault tolerance [94]. Information redundancy is the use of additional information with the aim of detecting or tolerating faults. Examples of information redundancy include the use of parity bits and error correcting codes. Temporal redundancy involves additional time for providing fault tolerance as, for instance, using multiple sequential computations, but it is only effective with transient faults.

2.5 Design diversity

Redundancy is not sufficient for tolerating solid design faults. A replicated hardware or software will fail identically for these faults, as they have the same design. In order to tolerate solid design faults, it is necessary to make use of design diversity, which means the redundancy of design.

Design diversity can be used in all forms of redundancy. In hardware systems it would involve using modules of different hardware design, whereas in software it would require different programs to implement the same function. For information

redundancy, diversity can be implemented by using different data structures and not just simple data copies.

Design diversity may be applied in all phases of the software development, such as system requirements, design and implementation. Diverse specifications, programming languages, algorithms and software teams can contribute to increase the design diversity and therefore to reduce failures related to design faults.

2.6 Hardware fault tolerance

Hardware fault tolerance is generally defined as the kind of fault tolerance for dealing with hardware faults. Hardware faults were a main issue in the early ages of computing. Although the reliability of hardware systems has been improving steadily, hardware faults are still a problem for dependable systems.

Hardware faults can be permanent or transient. Transient hardware faults may be produced, for instance, by high energy subatomic particles, electromagnetic radiation and power fluctuations. Bursts of radiation are responsible for permanent and transient failures in satellites.

Hardware fault tolerance can be implemented by using hardware or software mechanisms. The application of extra hardware to detect and correct errors was the first successful method for achieving fault-tolerant systems and it is still applied in memories, disks and microprocessors. The Leon [45] and PPC-750FX boards [54], applied in high-dependability applications as aerospace, use multiple circuits in the processor to recover from hardware failures.

The utilization of software techniques to recover from hardware failures is usually called software-based hardware fault tolerance [117]. In these systems, the system software is modified to implement error detection and handling in single or multiple computing units. Multiple computers are necessary to tolerate permanent hardware faults.

Some software mechanisms designed for handling hardware transient faults, such as backward recovery, are also effective against software elusive faults. The study in

[49] relates an experiment in which only 1 out of 132 elusive software faults have manifested again after a second run.

Hardware redundancy can be implemented in static, dynamic or hybrid configurations. Static redundancy techniques use compensation or masking to avoid system failures. A typical example is Triple Modular Redundancy (TMR), represented in Figure 2.3, in which three output channels (generated by hardware or software) are subjected to majority voting and consequently an error in one channel is tolerated. Static redundant systems are fast and simple to implement, but demand more hardware than other configurations. N-Modular Redundancy (NMR) is an extension of the TMR technique using "n" redundant modules, which are able to tolerate (n-1)/2 faulty modules.

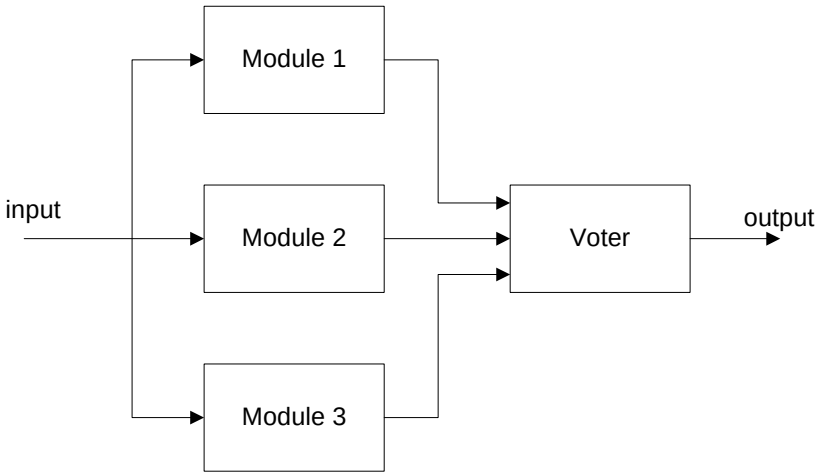

Figure 2.3: Triple Modular Redundancy.

Dynamic redundancy techniques use error detection followed by fault handling to isolate the faulty components. Two examples of dynamic redundancy are shown in Figure 2.4 [87]. In Figure 2.4(a) two self-checking modules are used, and the final output is chosen based on the error signals. In Figure 2.4(b), a self-checking unit is built by two modules that have their results compared. Other example of dynamic redundancy is the usage of standby sparing (hot, warm or cold).

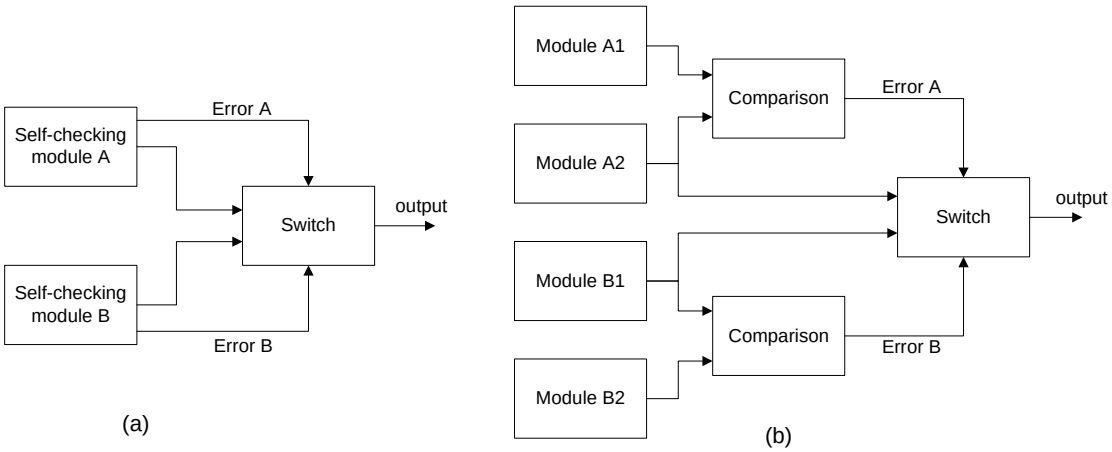

Figure 2.4: Self-checking modules.

Hybrid redundancy techniques combine elements of both static and dynamic redundancy as, for instance, the substitution of the faulty unit by a spare in the TMR technique.

The additional functionality needed to implement static, dynamic or hybrid hardware redundancy (e.g. voters and comparators) can be provided by hardware or software mechanisms. A common architecture based on software mechanisms consists of a multi-computer system connected by a communication network, commonly referred as a distributed system.

In distributed systems terminology, replication means the use of multiple hardware and software. The main replication techniques are:

- **Active replication** (also termed the state machine approach). In this technique all replicas process the inputs and send the results concurrently. This technique assumes that all replicas are deterministic and will reach the same results. For fail-silent nodes, the destination nodes are supposed to discard duplicated messages. The active replication technique may be extended to tolerate value failures [92], as TMR does, and even Byzantine failures [72] .
- **Passive replication** (also termed the primary-backup approach). It is a centralized technique equivalent to standby sparing. In this technique all inputs are sent to a primary replica, which processes them and replies, updating the state of the backup replicas. If the primary replica fails, one of the backup replicas assume as primary. Passive replication can only be applied in fail-silent nodes.

2.7 Software fault tolerance

Complexity is the root cause of software faults in computer systems [81, 111]. Software fault tolerance is needed because of our inability to produce error-free software. Fault tolerance can be applied to different software layers and software elements, such as at the operating system level, the application level, the process level, object level and function/method level.

Software fault tolerance can be divided in two groups: single version or multiple version software techniques. Single version techniques aim to tolerate software faults with a single software implementation, or version. To accomplish this, single version software can use rollback and rollforward techniques, as well as time and information redundancy. Examples of single versions techniques include error detection, checkpoint and restart, exception handling, and input data re-expression. Although single version techniques such as exception handling cannot fully recover from errors, they can be used to produce fail-controlled systems.

In contrast, with multiple version techniques, two or more software versions are executed sequentially or concurrently. These versions are created using some kind of design diversity, such as different programming teams or different algorithms, in order to avoid design faults. Several strategies have been proposed to implement fault tolerance with multiple version software, although most of them use the same architectural principles used in hardware fault tolerance.

Multiple version software is in general very expensive, but it has been used in safe-critical systems such as flight control systems, e.g. Airbus A340 [25], transport systems, e.g. Elektra Railway Signaling System [58], and space systems, e.g. NASA Space Shuttle [104]. The degree of design diversity utilization is variable. Full diverse software may use even different specifications for each software team, while in the other extreme diversity may be implemented by a single programmer, using different algorithms for each software version.

2.8 Fault tolerance strategies

This section presents several fault tolerance strategies, both for hardware and software fault tolerance. A fault tolerance strategy, also termed technique or scheme, is usually a pattern for fault tolerance implementation, using a set of error detection, error handling, fault handling, redundancy and diversity mechanisms.

2.8.1 Checkpoint and Restart

The application of the Checkpoint and Restart technique started with computers in the late 1950´s [50]. As the reliability and availability of these systems were very low, it was common to save the state of a task in stable storage to avoid losing all the work after a system failure.

Checkpoint and Restart is a strategy based on backward recovery [93]. After detecting an error, a system or component tries to reach a previous error-free state and then restarts processing again. Checkpointing can be taken periodically or at previously determined points as, for instance, before executing some operation.

The application of Checkpoint and Restart is effective against transient hardware faults and elusive software faults because they probably will not be activated in a second execution under a slightly different context. Randell [98] states the following about the use of checkpointing mechanisms: “fault tolerance does not necessarily require diagnosing the cause of the fault, or even deciding whether it arises from the hardware or the software.”

A checkpoint can be saved in memory or in stable storage, and is generally discarded after the next checkpoint is executed. Other mechanisms of recovery points include recovery cache and audit trail. In the recovery cache mechanism, only states that will be changed are saved. In contrast, in the audit trail mechanisms, all state changes are saved.

2.8.2 Recovery Blocks

The Recovery Blocks (RB) strategy [55, 98] is an extension of the Checkpoint and Restart strategy for multiple version software. In this technique, two or more software variants are implemented. The main software variant, also called primary alternate, is executed first and then an acceptance test (AT) is performed. The AT is an application-dependent error detection mechanism, such as a reasonableness check. If the acceptance test detects an error, alternate versions are executed sequentially until one of them is successful. If all variants fail, the recovery block strategy ends in a failure condition, and the error must be treated using forward recovery.

The general implementation of Recovery Blocks is shown in Figure 2.5. A checkpoint or other kind of recovery point is taken before starting the execution of alternates. After executing an alternate, an acceptance test is run and, in case of success, the checkpoint is discarded and the recovery block ends normally. If otherwise the acceptance test fails, the checkpoint is restored and a new alternate is executed, unless no alternates are available, which represents a failure.

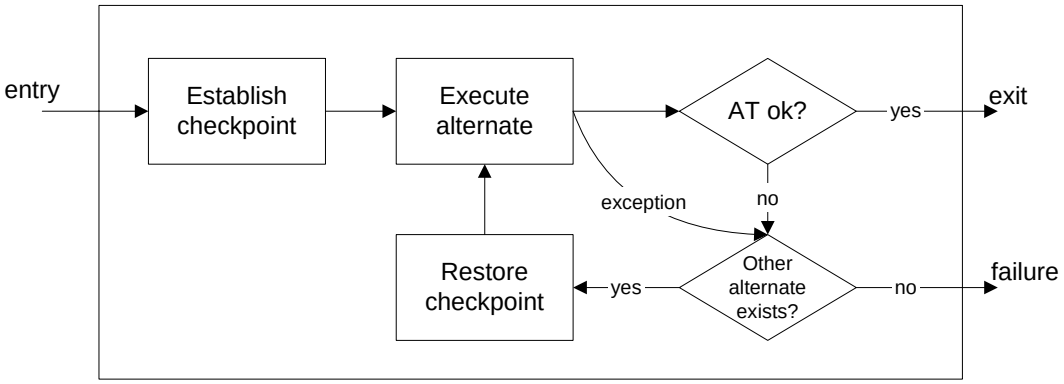

Figure 2.5: RB execution.

Recovery Blocks may optionally use watchdog timers for establishing deadlines for the variants execution, as a way to detect an anormal behavior, such as infinite loops. A watchdog timer may be configured with the worst case execution time of the alternate, before its execution. The watchdog timer activation acts as an exception signal to the execution control of the recovery blocks strategy.

Typically, for primary alternate it is selected the more effective software version. For the second or further alternate versions, a degraded functionality may be

provided, as the primary version is expected to run correctly in future activations. The degree of diversity in recovery blocks is restricted to different algorithms because each alternate is implemented as a function or class method.

The Recovery Blocks construct can be nested. This means that inside one alternate it should be possible to start another RB, and so several levels of recovery block could be running at the same time. However, this feature demands a more complex checkpoint and control implementation.

Most implementations of Recovery Blocks try to make the recovery point mechanism automatic, as for instance using recovery caches, either in hardware or software. Recovery caches save only global data accessed by alternates. However, in order to restore the previous state after an error has been detected by the acceptance test, all the operations taken by the software alternated have to be reverted. If an input or output has occurred after the last checkpoint, as for instance, by sending or receiving a message, this operation has to be reverted. Therefore, the implementation of recovery blocks in concurrent systems must take in account the coordination between recovery points in different processes or nodes to prevent system inconsistencies and the domino effect [98].

The acceptance test is unique for all alternates and it does not include any fault tolerance. Consequently it must be simple, effective and free from design faults. Besides, a complex acceptance test can introduce too much runtime overhead.

An experiment using the RB strategy in a Naval Command and Control System showed a failure coverage of over 70% [97]. The cost of the fault-tolerant software was 60% greater than the original software cost, and the system presented a 40% runtime overhead. These apparent high costs were considered acceptable in face of the improvement in system reliability.

2.8.3 Distributed Recovery Blocks

The Recovery Blocks strategy does not establish any procedure for execution in distributed environments. The Distributed Recovery Blocks (DRB) strategy [64]

combines the Recovery Blocks concept with distributed processing in dual nodes to provide additional fault tolerance for permanent hardware faults.

Figure 2.6 shows a block diagram of a DRB computing station. This scheme uses two computing nodes, two software variants (try blocks), and a common acceptance test. One of the nodes works as a primary node and the other as a shadow node.

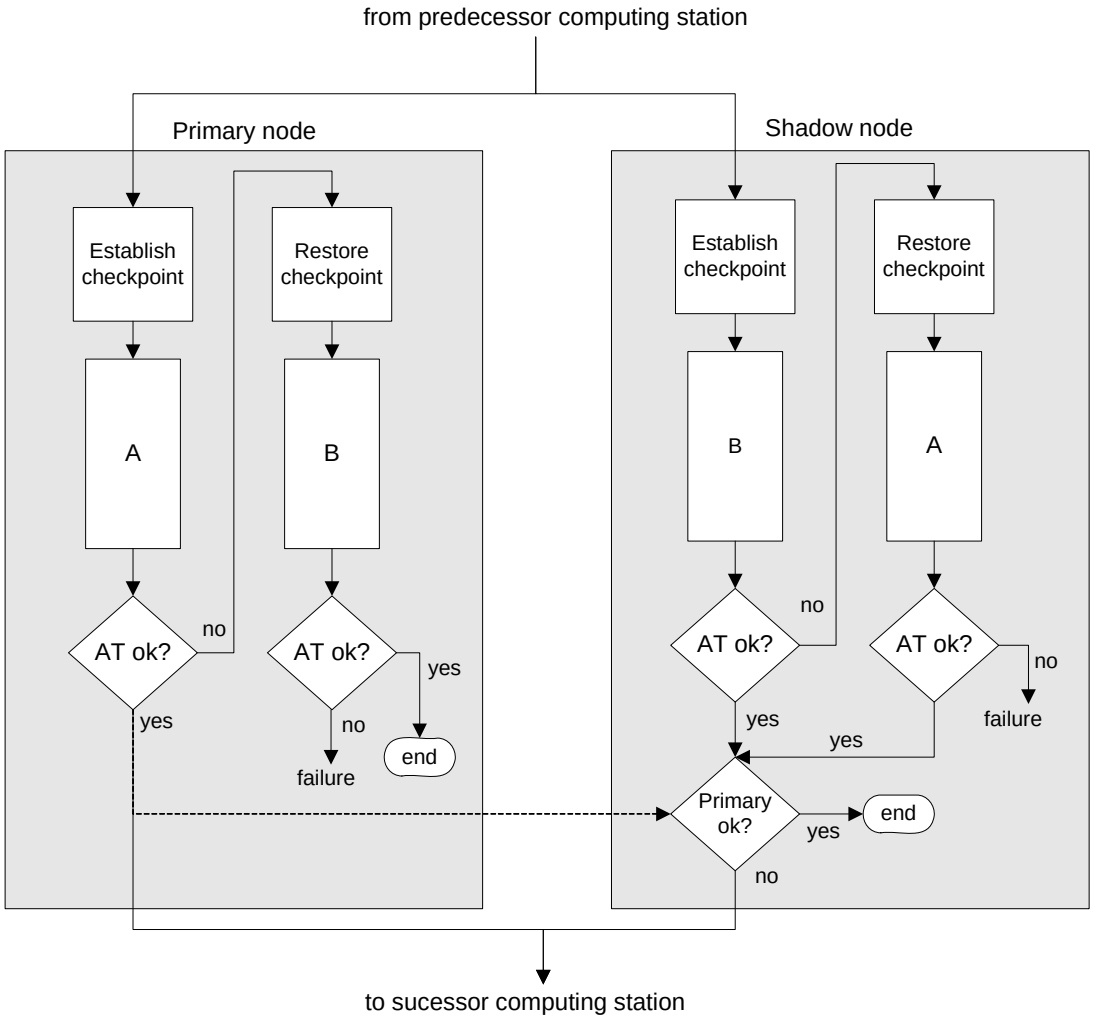

Figure 2.6: DRB execution.

In normal operation, only the primary node sends outputs to other computing stations. The nodes act in a two-phase mechanism. In the first phase, an input is selected for running, and in the second phase, an output is produced. The DRB operation is executed as follows. After an input selection, the two nodes start running different try blocks: the primary node runs try block A, while the shadow node runs try block B. After executing each try block, an acceptance test is performed. If the

primary node succeeds in try block A, it sends a message to the shadow node notifying its success and then outputs its results to the next computing station. However, if the primary node fails and the shadow node passes its test, the shadow node assumes the role of primary and sends its results. If both the primary and the shadow fail in the first try block, they try to execute the remaining try block. A correct execution of the second try block (A) in the shadow node will be a valid output. A correct execution of the second try block (B) in the primary node is necessary to keep state consistency between the nodes.

The DRB strategy also depends on a recovery point mechanism as the RB strategy does, and can also have watchdogs to control the try blocks execution. A failure in keeping the try block deadline is considered a failure in a time acceptance test.

The DRB strategy has the following major useful characteristics [66]:

- Provides a uniform treatment for hardware and software faults.
- The recovery time is reduced because concurrency is exploited between the primary and the shadow nodes. However, the timeout for failure detection of the primary node can affect this recovery time.
- In normal operation (no errors), the increase in processing time for the primary node is minimal because it does not have to wait for any message from the shadow node, although it has to send the AT success notification to the shadow node.
- It is cost effective because only two software variants are needed and the second version can be simpler and provide a degraded functionality.

The drawbacks of DRB are related to node coordination. First, it needs some mechanism for ensuring input data consistency, otherwise the two nodes will work in different computations and their state will become inconsistent. Second, it requires the communication of acceptance test results between the primary and the shadow node. A delay in receiving this result would make the shadow node change its role to primary, presuming it has failed. If that was not the case, both nodes would send their results and two primaries nodes would be active. Finally, a mechanism for detecting

role inconsistencies between nodes is required to avoid two primaries or two shadows at the same time (e.g. when both nodes fail in the try blocks).

The DRB strategy assumes that the communication network is reliable [66]. Fault values messages are negligible by incorporating error correcting schemes. However, acknowledge messages by successor computing stations may be required to assure reliable communication.

Some extensions to the DRB strategy have been proposed. The Extended Distributed Recovery Block (EDRB) [51] includes a supervisor station for confirming node crashes and misjudgments by DRB nodes about their partners. It also defines two networks: one for supervision and the other for working nodes communication.

An approach for extending the DRB strategy for using more than two nodes and more than two try blocks is described in [65]. This approach is called Recursive Shadowing [66] because each additional shadow node interfaces with the previous DRB station which is considered as a primary for the new configuration.

A Pair of Self-checking Processing Nodes (PSP) [70] consists of an implementation of a DRB station using only one software version. It combines the application of the Checkpoint and Restart strategy with two self-checking units. This configuration does not tolerate solid software faults.

2.8.4 N-Version Programming

The N-Version Programming (NVP) strategy [27] combines the use of software design diversity with the compensation technique. It is equivalent to static redundancy (e.g. TMR) in hardware software tolerance. In NVP, two or more functionally equivalent programs are executed either concurrently or sequentially and their outputs are compared by a decision mechanism implemented by software. If only two versions are used, the comparison of results is called matching, and can only detect errors. If more than two versions are used, the comparison of results is called voting, and errors can be detected and corrected by masking.

NVP includes a methodology for developing software versions with a high level of diversity, based on a common specification that should include all necessary

information for independent software teams [19]. It recommends the utilization of different algorithms, programming languages and compilers. The level of NVP application can be the entire program or single modules or functions.

The decision mechanism is the most critical element of the NVP strategy, as only a single version is provided. Differently from the exact voters used in hardware, NVP voters often must deal with inexact values, generated by different algorithms and programming languages. Besides, the voter design is application-specific, similarly to the acceptance test in Recovery Blocks. Several types of voters exist as majority, mean, consensus and dynamic voters [94].

In comparison with Distributed Recovery Blocks, concurrent NVP has the advantage of not requiring checkpoint mechanisms and the acceptance test. However, it demands more hardware and software versions for tolerating the same number of faults.

2.9 Fault-tolerant communication

A distributed fault-tolerant system depends heavily in fault-tolerant communications. Fault tolerance strategies have to rely on network facilities to deliver inputs and outputs to and from software variants, and to allow the coordination in strategy execution. Furthermore, for systems with global state, missing an input message will lead to state inconsistency among distributed variants.

In order to obtain a fault-tolerant communication system, the following methods are used [117]:

- Spatial masking – sending the same message by multiple links.
- Temporal masking – sending the same message multiple times.
- Detection/recovery – using acknowledgements, timeouts and retransmissions.

The detection/recovery method may use positive or negative acknowledgements. In the positive acknowledgement method, if a receiver does not send an acknowledgment after a timeout, the message is retransmitted. This may be repeated for a fixed number of times. In the negative acknowledgement method, the receiver is

responsible to detect that a message was lost (or is corrupted) and to ask for retransmisson. This may be implemented by using sequence numbers, or by using a time-triggered technique.

Multicast message transmission can use broadcast/multicast network facilities, or even point-to-point messages, where the same message is sent individually to all recipients. In that context and regarding sender resilience, multicast can be classified as [117]:

- Unreliable multicast: no effort is made to overcome link failures.
- Best-effort multicast: the sender makes some effort to deliver the message, such as performing retransmissions, but if the sender fails before delivering the message to all recipients no reliability can be guaranteed.
- Reliable multicast: the participants coordinate to ensure that the message is delivered to all recipients, as long as it is delivered to at least one recipient.

Even using broadcast/multicast network facilities, the sender may fail before the message is correctly received by all receivers. Possible reasons are electric noise or the lack of buffering space at the receiving node [61]. However, the implementation of reliable multicast involves several rounds of communication and large use of buffering, in order to guarantee atomicity in worst case scenarios. This high latency makes this method unsuitable for hard real-time systems. Therefore, many real time architectures use the best-effort approach, such as the Time-triggered Protocol (TTP) [71] and the Time-triggered Message-triggered Object Support Middleware (TMOSM) [70].

Besides reliable communication, some distributed fault tolerance strategies such as DRB and NVP also demand input data consistency. Some communication systems are able to guarantee the delivery of messages in the same order for all receivers. If that is not the case, the fault tolerance strategy must include a mechanism for input synchronization.

2.10 Fault tolerance software structures

In order to reduce the complexity of the fault-tolerant software and promote software reuse, several object-oriented patterns and frameworks have been proposed. These software structures generally translate fault tolerance concepts as variants and decision algorithms (also called abjudicators) into abstract classes that define interfaces for the implementation of fault tolerance techniques. A common approach is to separate the fault tolerance functionality from the application software, making it reusable. Additionally, the applications program becomes a user of the fault tolerance software, reducing system complexity.

Xu, Randel, Rubira-Calsavara and Stroud [119] proposed an object-oriented structure for dealing with software fault tolerance. They suggested the application of idealized components with diverse design using classes to implement the control algorithm, the software variants and the abjudicator, as shown in the example of Figure 2.7.

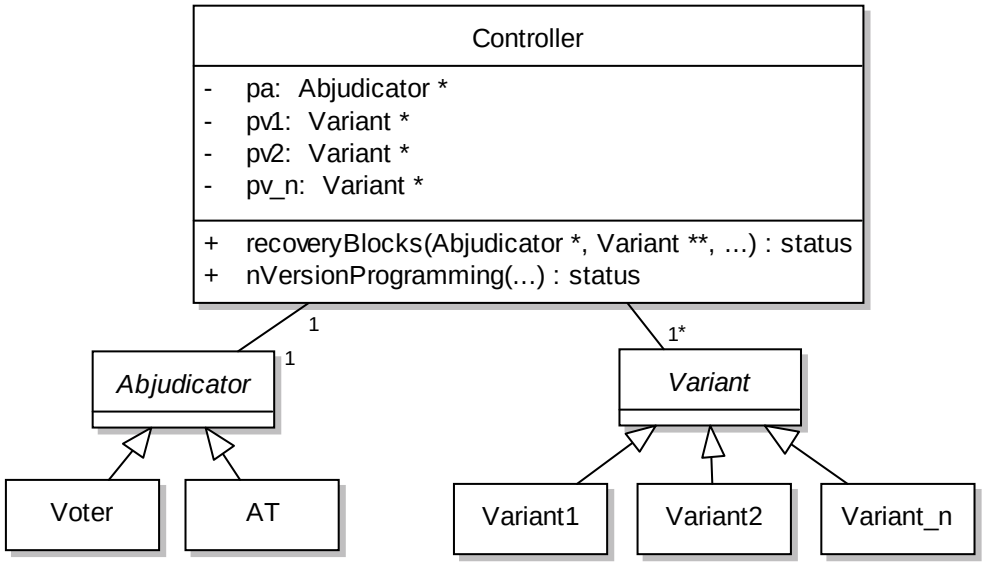

Figure 2.7: Xu,Randell, Rubira-Calsavara and Stroud´s framework example.

Each fault tolerance technique is implemented by a method of the *Controller* class, using one *Abjudicator* and several *Variant* objects passed as arguments by the application program. In this architecture, the inclusion of a new fault tolerance strategy demands the addition of a new method to the *Controller* class. There is no

definition on how input data is passed for the variants and how the results are returned, but a general solution must be adopted, otherwise the *Controller* class would not be reusable. Specialized abjudicators can be defined by deriving the *Voter* and AT classes.

Variant classes can achieve design diversity by using diverse algorithms and internal data structures. This is termed class-level design redundancy. However, some mechanism must be provided for maintaining state consistency among *Variant* objects if they maintain their state between activations. Less general solutions to variant diversity include object-level design redundancy, in which variant objects belong to the same class but are initialized with slightly different data, and operation-level design redundancy, in which variant classes have diverse implementation algorithms but no class data.

Tso, Shroki, Tai and Dziegiel [115] developed and implemented a framework of software tolerance components. Figure 2.8 shows the class diagram for their implementation of the Recovery Blocks technique.

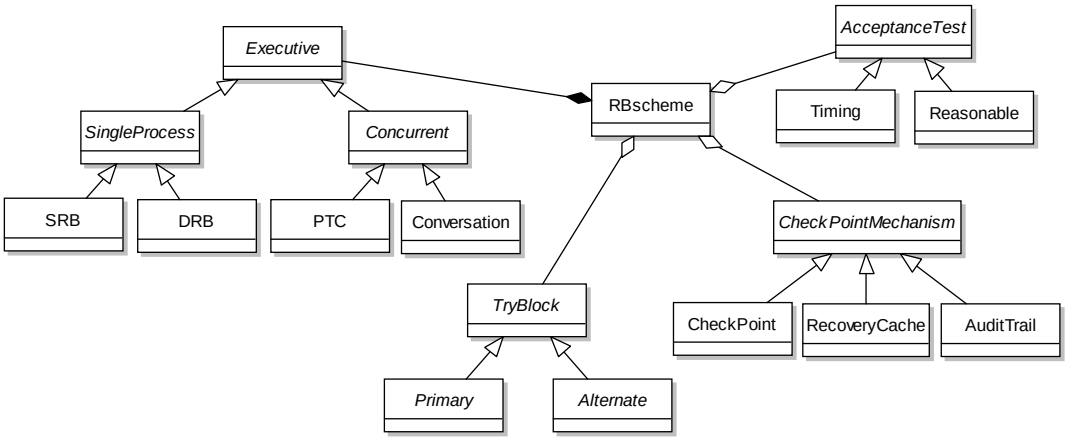

Figure 2.8: Tso, Shokri, Tai and Dziegiel´s class diagram for the RB technique.

The *RBscheme* class is responsible for implementing the Recovery Blocks technique. It delegates the control algorithm to an *Executive* object, which is specialized by inheritance to cover several execution schemes, using single and concurrent processes. Primary and alternate variants are implemented as classes derived from the *TryBlock* class. Acceptance tests algorithms are defined by classes that inherit from the *AcceptanceTest* class. Checkpointing mechanisms, as recovery

caches and audit trails, are implemented by classes derived from the *CheckPointMechanism* class.

The main drawback of this framework, comparing to Xu et al. framework, is the definition of a different class structure for each fault tolerance scheme. For instance, voter classes are added for NVP and data re-expression classes are added for data diversity techniques, such as Retry Block and N-Copy Programming [11].

Daniels, Kim and Vouk [35] proposed the Reliable Hybrid pattern, which targets the design of fault tolerance applications. The focus of this pattern is on the decision mechanism, which can combine acceptance tests and voters in hybrid strategies, such as Concensus Recovery Blocks [102] and Acceptance Voting [18]. Figure 2.9 presents the Reliable Hybrid pattern structure.

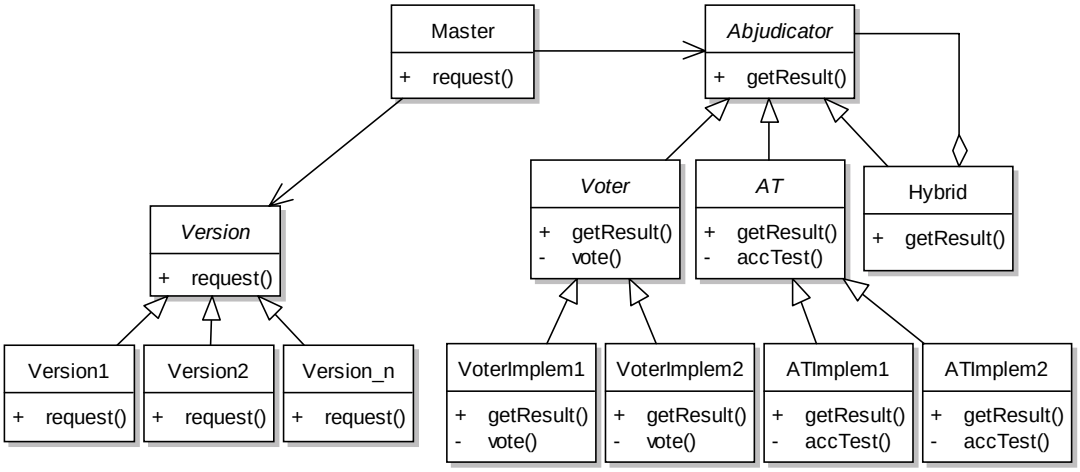

Figure 2.9: Reliable Hybrid pattern class diagram.

The Reliable Hybrid pattern has a class diagram that is similar to Xu et al. framework. The improvement is related to the abjudicator, which includes the *Hybrid* class and implements the Composite pattern [47]. The *Master* class has a single association with one *Abjudicator* object, which may be a *Voter*, an *AT* or a *Hybrid* object. The *Hybrid* class possesses a list of *Abjudicator* objects (*Voters*, *AT* objects and other *Hybrid* objects) and its *getResult* method calls each *Abjudicator* object sequentially until a successful result is obtained.

In this pattern, the fault tolerance strategy is performed by the *Master* class, which calls the several *Version* objects and sends their results to the *Abjudicator* object. However, no specific mechanism is devised to change the control algorithm.

Xu and Randell improved their previous framework and published it as the Generic Software Fault Tolerance (GSFT) pattern [121]. This pattern class diagram is shown in Figure 2.10.

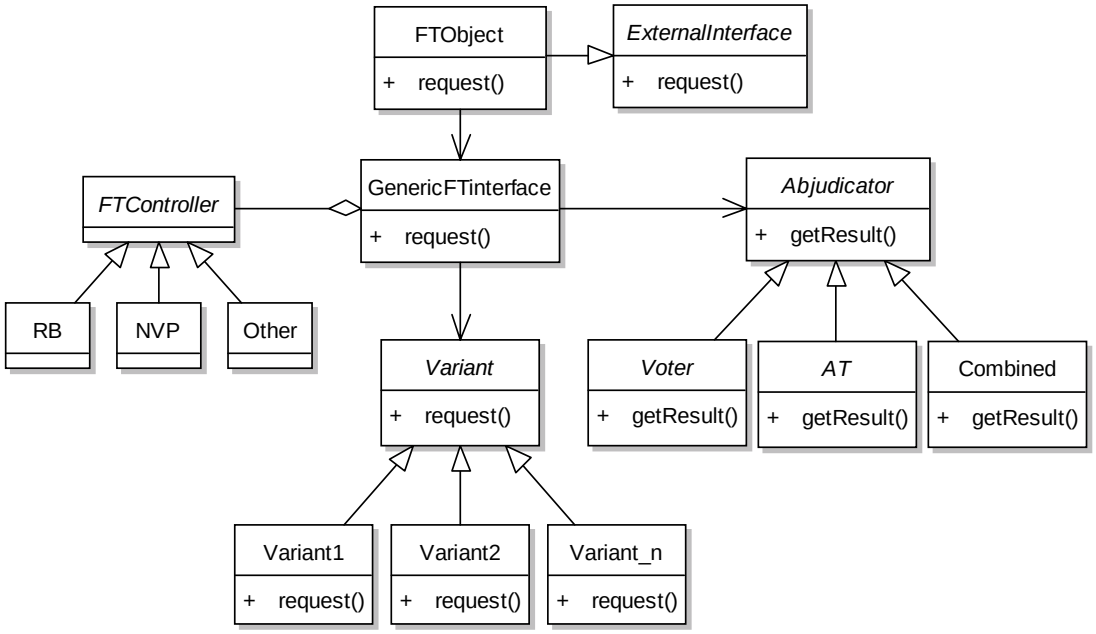

Figure 2.10: The Generic Software Fault Tolerance pattern class diagram.

A fault-tolerant class (*FTObject*) must implement *ExternalInterface* to conform to the interface characteristics of an idealized component. The *FTObject* class passes the user requests to the *GenericFTInterface* class, which actually executes the fault-tolerant processing, using *FTController* subclasses to implement the control algorithm. The abjudicator is implemented similarly to the Reliable Hybrid Pattern, including a *Combined* class that behaves as the *Hybrid* class in that pattern. The main difference of this pattern to the original framework proposed by the authors is the inclusion of the *FTController* hierarchy that implements the control algorithm by applying the Strategy pattern [47], similarly to the Tso et al. framework (Figure 2.8).

The GSFT pattern is, to our knowledge, the most comprehensive framework for fault tolerance ever presented. However, it leaves undefined many issues. One is regarding data passing between variants, abjudicators and the user application.

Another issue is how to implement this pattern using processes or threads as units of fault tolerance. In Chapter 5 we propose a fault tolerance framework that addresses these issues.

2.11 Fault tolerance application support

Fault tolerance can be supported at different software layers, such as the operating system, the middleware and the application level. This section presents the related work in support fault tolerance at the application level.

2.11.1 FT-RT-Mach and DEOS

The FT-RT-Mach project of the FORTS group at University of Pittsburgh [44] consisted of the implementation of fault tolerance support for the RT-Mach, an operating system developed by the Carnegie Mellon University [110]. The project purpose is to tolerate transient faults by thread re-execution in case of error detection without modifying the original Rate Monotonic Scheduling (RMS) of periodic threads, using the Fault-Tolerant RMS (FTRMS) algorithm [40]. The algorithm affects the thread admission control of real-time periodic threads, as it takes into account the time needed for thread recovery [37]. A thread in FT-RT-Mach has its context information cleared at the end of each execution. A fault flag is provided for each thread and it can be set by exception handlers or by application threads. This flag is tested at the end of each thread execution and may trigger error recovery mechanisms, such as Checkpoint and Restart or Recovery Blocks. Checkpointing in FT-RT-Mach is not provided by the operating system; therefore, it must be implemented by the application threads.

The same mechanisms used in FT-RT-Mach were applied in the DEOS operating system, a commercial avionics operating system developed by Honeywell [37]. The FTRMS algorithm and the fault tolerance support were adapted to this operating system, which presents several differences in relation to FT-RT-Mach. Threads in DEOS never have their context information cleared, and they usually run in an infinite

loop, calling a function to suspend itself after each execution. Additionally, periodic threads must be harmonic. Two periodic threads are harmonic if the larger period is an integer multiple of the smaller. The checkpoint mechanism is performed by the operating system, by defining and managing a backup state memory for each thread. However, in order to reduce the time and memory overhead, the application programmer has to define the set of variables that are considered as state information. Only those variables are saved by the operating system.

2.11.2 Delta-4

The Delta-4 [21, 92] was a collaborative project developed by a multinational team of companies and academic researchers. It started in 1986 and terminated in 1992, aiming the definition of dependable distributed system architecture for real-time systems areas, such as computer integration manufacture. The system is meant to be used on local area networks communicating by message-passing between nodes. The architecture separates each node into a host computer and communication hardware called Network Attachment Controller (NAC). NACs use built-in hardware self-checking to be fail-silent and are capable of reliable multi-point communication using an atomic multicast protocol. This protocol, implemented at the data-link layer of the Open System Interconnection (OSI) model, guarantees atomicity and ordering to all messages. As messages can be lost, it uses a message retry mechanism that tolerates a pre-defined number of successive omission failures. Replicas are application objects (processes) that can communicate using synchronous (equivalent to Remote Procedure Call - RPC) or asynchronous messages.

Delta-4 supports the following fault-tolerant strategies:

a) For hardware fault tolerance:

- Active replication.
- Passive replication.
- Semi-active replication.

b) For software fault tolerance:

- Recovery Blocks.
- N-Version Programming.

In the implementation of active replication, a system supplied voting mechanism may be used to compare message signatures and select the correct output. In NVP, the voting algorithm is application-dependent, as different software versions can find different correct answers.

The system has mechanisms for cloning a new replica in order to replace a failed one. The cloning mechanism depends on whether the replica is stateless or not. Stateless replicas only require a standard initialization while state replicas require some form of acquiring the current state from other replicas via a standard interface.

Unfortunately, the architecture proposed by Delta-4 was not applied in many field applications and in research area, probably because it needs a special hardware for the NAC.

2.11.3 TMOSM and ROAFTS

The DREAM laboratory at University of California - Irvine [38] has been working on real-time and fault tolerance computing in object-oriented distributed architectures. Their work is based on the Time-Triggered Message-Triggered Object (TMO) structuring scheme, or model, formerly named RTO.k object scheme [67]. In this model, a real time object has both time-triggered methods, which are activated at predefined times and message-triggered methods, which are asynchronous and non-blocking. Message-triggered methods have lower priority and are not allowed to execute if they can interfere with time-triggered methods. For both kinds of methods deadlines can be established and monitored.

The execution of TMO objects is controlled by the TMO Support Middleware (TMOSM) [69]. This middleware has been ported to several operating systems as Windows NT, Solaris, Windows XP, Windows CE and Linux. The middleware requires clock synchronization between nodes and an operating system clock tick

service for triggering a top priority thread, called Watchdog Timer and Scheduler Thread (WTST). This thread is responsible for scheduling other middleware threads and the pool of threads that effectively run the TMO application methods. WTST reserves one time slice in three for the exclusive execution of application threads, while the other time slices are scheduled for middleware threads. For instance, in the Windows NT implementation described in [69], the time slice is set to 3 ms and, therefore, an application thread is scheduled to run freely for 3 ms in a 9 ms cycle. This mechanism is designed to reserve a minimum amount of CPU utilization for application threads. Additionally, unused time periods attributed to middleware threads are spent by application and operating system threads. In a defense prototype case study it was observed that deadlines of 20 ms were met for about 99.9 % of time. The failure in accomplishing the deadlines is reputed to overheads introduced by operating system threads that could not be disabled.

Communication between TMO objects uses the concept of Data Field Channels that are logical multicast channels based on some ID, called content code. It supports two types of messages: state messages and event messages. State messages carry information to be stored at fixed memory locations and messages can overwrite data before being read by some process. Event based messages are normal messages that are stored in a buffer after being received.

Fault tolerance was introduced by means of the Primary-Shadow RTO.k replication (PSRR) scheme [67], which executes TMO objects replicas using the DRB or PSP fault tolerance techniques. In this scheme, the shadow node is supposed to receive several messages from the primary node, such as the acceptant test result and an output success confirmation. The PSRR scheme has later evolved to implement adaptive fault tolerance by means of the Real-Time Object-Oriented Adaptive Fault Tolerance Support (ROAFTS) middleware [68], which is able to switch between three basic modes: DRB/PSP (or parallel redundant mode), RB (or sequential backward recovery mode) and exception handling (or sequential forward recovery mode). The decision about changing FT modes is based on equipment availability, criticality and recovery time. The middleware configuration includes network surveillance and reconfiguration services in order to detect and confirm failures in working nodes. ROAFTS has been ported to the Solaris operating system and CORBA, using 100 ms

as time slice and satisfying deadlines from 40 to 100 ms [103]. The development of ROAFTS is still in early phases and no implementation is publicly available as occurs with the TMOSM [38].

2.11.4 Adaptive Fault Tolerance for Spacecraft

Adaptive Fault Tolerance (AFT) for Spacecraft [52] is a middleware designed for space applications which can change the application fault tolerance configuration based on the mission phase, the failure history and the environment. It aims to cover hardware physical faults, rare conditions in software and unusual environment effects. It is based on a dual redundant system architecture which runs atop the VxWorks operating system. Tasks in this system are classified into critical and non-critical, periodic and aperiodic. The objective of the adaptive fault tolerance mechanism is to match redundancy and resource consumption with the mission phases and reliability requirements. The system runs in one of 8 possible modes, differing in node processor speeds and in the responsibilities of critical and non-critical tasks execution. Some modes involve replication of critical tasks only, others the replication of all tasks, and others no replication at all. For non-replication modes, transient faults can be detected and tolerated by using acceptance tests and backward recovery mechanisms. Replicated nodes use DRB or a Primary/Backup architecture using active replication. Non-replicated modes use exception handling and Recovery Blocks.

The communication between nodes uses TCP or UDP socket primitives. Messages can be sent to logical channels and multicast. Each task can join or leave a channel dynamically. Both reliable and unreliable communications are provided. For reliable logical channels, the delivery mechanism is based on the concept of negative acknowledgement. Additionally, period cross-check messages circulate among the replicated processes to ensure that any broadcast or multicast message has not been lost.

In redundant strategies, the middleware is responsible for checking the heartbeat of both replicas and to ensure that they have a consistent state. The replicated data management maintains consistent (synchronized) state data among replicated objects using several strategies such as processing input with uniformity (state updates

associated to incoming messages are made to all processes before a response is generated), processing without uniformity (a response is generated without ensuring that all processes have updated their state data) and periodic update (state update is sent by the primary process on a periodic basis).

The middleware has a node state restoration service to restart a failed node or shutdown node and provide the startup configuration. State restoration can be performed as a single event or incrementally. The system does not need clock synchronization between computers.

2.11.5 Fault Tolerant CORBA

The Common Object Request Broker Architecture (CORBA) is a remote method invocation based middleware defined by the Object Management Group (OMG) [90]. It offers transparency in relation to objects location and programming language in which they are implemented, and hides operating systems, platforms, networks and protocols details from application programs. The Fault Tolerant CORBA (FT-CORBA) specification [88] is part of the formal CORBA architecture that aims to provide fault tolerance support for applications that require high level of dependability. The fault tolerance mechanisms provided by FT-CORBA are based on entity redundancy, or the replication of CORBA objects. Besides, the specification defines mechanisms for error detection and recovery. The following replication styles are supported in FT-CORBA:

- **Stateless**: the replicated objects maintain no state data and therefore no state consistency mechanism is performed.
- **Cold Passive Replication**: only the primary replica responds to client requests. If the primary fails, then backup replica is selected and the state of the failed primary is loaded from a logging system.
- **Warm Passive Replication**: similar to Cold Passive Replication, but the state of the primary is transferred periodically to the backup replicas during normal operation. This type of recovery provides faster recovery than Cold Passive Replication.

- **Active Replication**: all replicas execute the request simultaneously but only one reply is sent to the client. Duplicate messages are discarded automatically by the infrastructure. This mechanism provides faster recovery from failures, but requires replica determinism and total order message delivery to maintain state consistency among replicas.
- **Active Replication with Voting**: this is a planned extension to the existing specification and adds a mechanism of exact majority voting before sending a reply to the client.

The present FT-CORBA specification provides only fault tolerance to crash failures. Faulty objects are supposed to stop working without generating incorrect results. The fault detection mechanisms supported by FT-CORBA are based in heartbeats and timeouts only. The implementation of FT-CORBA requires the utilization of objects working as Replication Managers, Fault Detectors and Fault Notifiers. The creation and management of objects and object groups can be implemented by the FT-CORBA infrastructure or by the application program.

The application of FT-CORBA in real-time system is limited because it can spend an unpredictable amount of time detecting faults and recovering from them [48]. In Passive Replication the recovery time needed to switch to a backup replica can be unacceptable for a real-time application and when using Active Replication too much time can be spent providing totally ordered reliable multicast. The Real-Time CORBA specification [89] targets systems with real time requirements, but this specification is not compatible with FT-CORBA [48, 85].

Several projects aimed the implementation of fault tolerance in CORBA, such as Aqua [99], DOORS [86] and MEAD [85].

2.12 Summary

Fault tolerance is a means of achieving high dependability for critical, long life and high-availability systems. Despite the efforts to prevent and remove faults in systems development, the application of fault tolerance is usually required because the hardware may fail during system operation and the software is rarely fault free.

The implementation of fault tolerance involves the application of error detection and system recovery. System recovery aims to eliminate the error from the system state, and additionally may diagnose the fault and preventing it from being activated again. Fault tolerance implementation depends on redundancy, the utilization of additional resources, and in design diversity in order to tolerate design faults.

Several fault tolerance techniques have been described, both for hardware and software fault tolerance, using single or multiple version software. The emphasis was to present FT strategies that are applied in this work, such as RB, DRB and NVP. Fault-tolerant communication concepts have also been introduced.

The related work regarding software structures for fault tolerance have been presented. This includes frameworks and design patterns proposed by the research community in order to reduce the complexity of the fault-tolerant software and promote software reuse.

Finally, the related work regarding application-level fault tolerance support has been presented. Some works introduce fault tolerance support by the operating system, e.g., FT-RT-Mach, while others by the middleware, such as ROAFTS and FT-CORBA.

Chapter 3

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a new programming technique that targets the modularization of crosscutting concerns. This chapter introduces the main concepts related to AOP, describes the AspectC++ language extension and presents the related work regarding the application of AOP in operating systems, middleware and fault-tolerant systems.

3.1 Separation of concerns

Separation of concerns is a concept that has been applied in software engineering for a long time [36, 91] and involves the division of the software application in smaller functionalities or concerns. Separation of concerns leads to the development of systems in modules that could be developed largely independently from each other, reducing the system complexity and improving its reusability. The usage of procedural programming is the initial step into separation of concerns. Later, the concept of information hiding was introduced and contributed to Object-Oriented Programming (OOP) as a new mechanism of separation of concerns.

The lack of separation of concerns in a system can be detected by inspecting its source code and looking for the existence of code tangling and code scattering. Code tangling happens when a module handles multiple concerns. For instance, the same source code can be dealing with business logic, persistence and distribution concerns. Code scattering happens when a concern implementation is spread in multiple modules. Figure 3.1 shows examples of code tangling and code scattering.

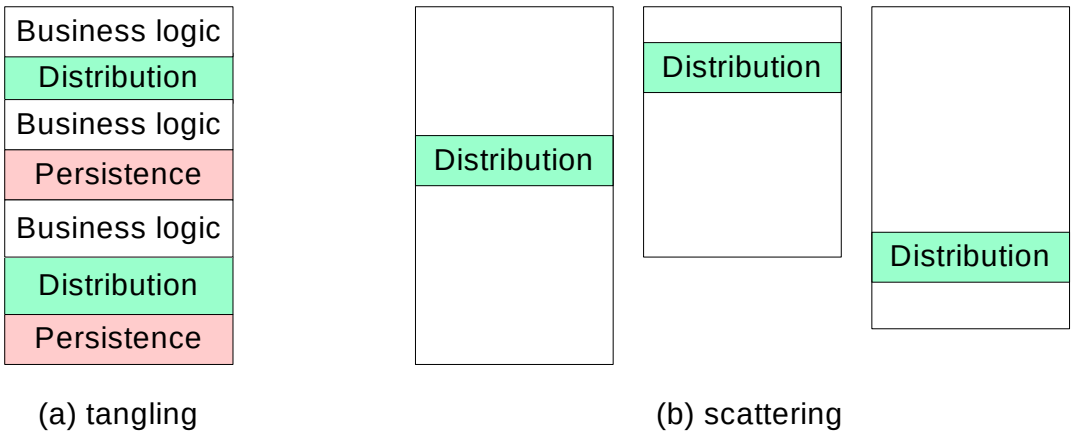

Figure 3.1: Code tangling and code scattering.

Some concerns are very hard to separate from others. The implementation of these concerns is often tangled with other concerns and is scattered throughout the code. Therefore, they are called crosscutting concerns. Examples of crosscutting concerns are distribution, fault tolerance, and security. Crosscutting concerns are also

called non-functional, as opposed to the functional concerns that implement the system's main functionality.

New mechanisms to provide advanced separation of concerns have been proposed. The work in [56] identifies and analyses some mechanisms for modularizing crosscutting concerns, such as Meta-level Programming [77] and Composition Filters [8].

3.1.1 Meta-level Programming

Meta-level Programming is based on meta-object protocols (MOP), which enable the modification of the language semantics and implementation. Meta-object protocols are the interface between the base-level program and the meta-model program. By intercepting the activation of methods in the base-level program, meta-objects have the opportunity to execute other concerns. However, there is no special mechanism to separate crosscutting concerns from each other. An example of MOP for the C++ programming language is OpenC++ [28]. In OpenC++, the complete syntax tree is visible on the meta-level and arbitrary transformations are supported.

The work in [120] evaluated OpenC++ to implement software fault tolerance techniques, such as RB and NVP, in a distributed sorting application. They concluded that the meta-object approach provides a cleaner and simpler interface to applications comparing with standard object-oriented implementations. Moreover, they measured a runtime overhead factor of about two between calling an OpenC++ operation and calling a C++ operation, but this overhead was considered small in comparison with the overhead imposed by the fault tolerance mechanism.

3.1.2 Composition Filters

Composition Filters extend object-oriented programming by adding filter classes. Messages between objects are processed by filters both before and after the normal method execution. More than one filter may be applied to a single message. Separation of concerns is achieved by defining a filter class for each crosscutting

concern. No research work concerning the application of Composition Filters to fault tolerance has been reported yet. The TRESE group at the University of Twente [114] has available implementations of Composition filters for C# and Java.

3.1.3 Aspect-Oriented Programming

Aspect-Oriented Programming is a programming technique proposed in [62]. In AOP, components are defined as properties of a system, for which the implementation can be cleanly encapsulated. In contrast, aspects are properties for which the implementation cannot be cleanly encapsulated in a generalized procedure. Aspects and components crosscut each other in the implementation of a system. The goal of AOP is to support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system.

The process of composing components and aspects is performed by the **aspect weaver**. Essential to the weaver operation is the concept of **join points**, which are the elements of the components' static structure or dynamic behavior that aspect programs are able to coordinate with. Join points can be method calls, variable accesses or any other point in the execution of a program where additional behavior can be attached. The kind of join points allowed for a given AOP implementation defines its join point model. The behavior introduced in a join point is named **advice**. A **pointcut** defines a set of join points.

A difference between AOP and other separation of concern approaches is the definition of different abstraction and composition mechanisms for components and aspects [62]. The work in [42] proposes that the distinguishing properties of AOP are **quantification** and **obliviousness**. Quantification is the capacity of writing unitary and separate statements that have effect in many non-local places in a programming system. A quantification mechanism allows reaching several join points of the code with one declarative statement. Obliviousness means that the component code does not need to be prepared or aware of the additional behavior introduced by aspects. Therefore, programmers in the components side (or base code) do not have to expend any additional efforts to make the AOP mechanisms work.

Another definition of AOP was presented in [42]:

"In program **P**, whenever condition **C** arises, perform action **A".**

In this definition, the program P represents the base component code. The condition C is defined by a pointcut in the aspect code, and the action A is the advice executed at the join points. The condition defined by C can be evaluated at compile-time based on the static structure of the base code (e.g. method execution) or at runtime, based on the dynamic behavior of the program (e.g. calls to method X in the execution context of method Y). A single aspect usually defines a set of pairs (C, A).

Most criticism against AOP is related to the obliviousness property. The base code can evolve and the original join points used by aspects can be modified. So, aspects can miss the desired join points or capture undesired join points. This problem has been called the AOSD (Aspect-Oriented Software Development) Evolution Paradox [112]. The inexistency of an explicit interface between the base code and the aspect code compromises the independent evolvability of the base code. On the other hand, the use of explicit interfaces in the base code (e.g. annotations) reintroduces the scattering that AOP was supposed to avoid [107]. Despite this and other drawbacks [34, 84], the acceptance of AOP by the researchers from academia and industry is high, possibly because AOP is very powerful and can solve real problems related to crosscutting concerns.

The same team that proposed AOP has developed its first and most popular implementation to date: AspectJ [17]. The two existing implementations of AOP for C++ are AspectC++ [16] and XWeaver [122]. The AspectC++ implementation was applied in this work, and it will be described in the next section.

3.2 AspectC++

AspectC++ is a general-purpose aspect-oriented language extension to C++ [16] [106]. It has been strongly influenced by the AspectJ language model, but supports additional concepts that are unique to the C++ domain. A primary design goal of AspectC++ was to keep the low runtime overhead of the C++ programming language,

aiming its application in resource-constrained environments such as embedded systems.

3.2.1 Weaving

The AspectC++ weaver composes the C++ base code and the aspect code in a source-to-source transformation, as shown in Figure 3.2. After the weaving process, the resulting source code can be compiled by any C++ compiler.

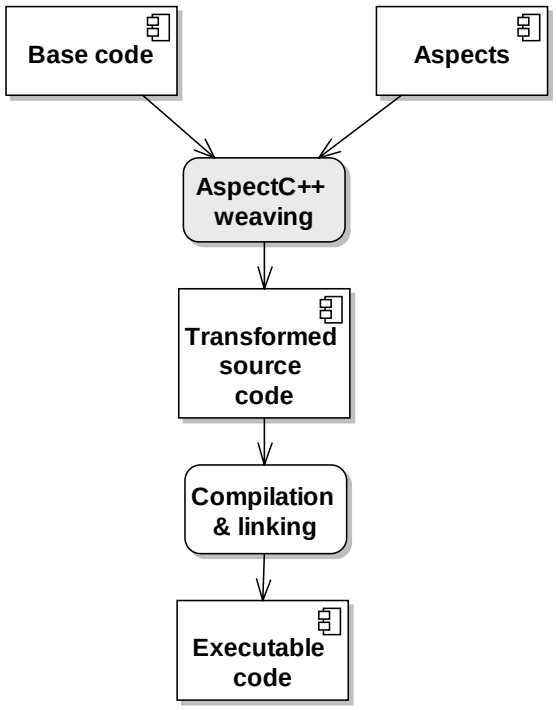

Figure 3.2: AspectC++ weaving process.

Two weaving modes are available: Whole Program Transformation (WPT) and Single Translation Unit (STU). WPT transforms all files (header files and translation units) in the project directory tree and saves them in a new directory tree. The aspect code, normally using the ".ah" file extension, is also transformed and saved in the new directory tree. In the STU mode, the weaver transforms one file at a time, making easier to integrate the weaver with *makefiles* and Integrated Development

Environments (IDE). In this mode, all header files directives are expanded and saved together with the transformed translation unit.

3.2.2 Join points, pointcuts and advices

The following types of advices are supported in AspectC++:

- Code advices: define a computation that can be executed **before**, **after** or **around** (instead of) a given join point.
- Introductions: define new attributes, methods and parent classes to existing classes.
- Order definitions: establish the order of application among aspects.

Two types of join points are supported in AspectC++: name join points and code join points. **Name join points** (or static join points) are named instances in the static program structure, such as a class name, function name or namespace. **Code join points** (or dynamic join points) represent events that happen during program execution, such as the calling or execution of a function. Code join points result from the application of **pointcut functions** to name join points. Four basic types of code join points exist: call, execution, construction and destruction. Call and execution join points are related to methods; construction and destructions join points are related to classes.

Figure 3.3 presents a very simple example program using AspectC++ designed to debug some method activations in the base code. In this program, the aspect *DebugClasses* contains one pointcut (*debug*) and one code advice (a *before* advice). The *debug* pointcut is defined by a **pointcut expression** that combines an *execution* pointcut function with a *call* pointcut function using the algebraic "or" operation. The *execution* pointcut function will select the join points related to the execution of all methods of ClassA. If for instance this class has three methods, then three join points will be selected. The *call* pointcut function will select the join points related to the calling of ClassB methods whose names begin with the "set" string. If, for instance, ClassB has only one method that matches this expression, the number of selected join points will depend on how many places this method is called in the entire application

code. The *before* advice defined in *DebugClasses* will be executed always before the selected join points. In this case, the advice code just prints the method signature (e.g. "void ClassA::methodA(int,short int)"), provided by AspectC++.

```
aspect DebugClasses {

  pointcut debug() = execution("ClassA") ||
                     call("% ClassB::set%(...)");

  advice debug() : before() {
    printf("debug:before %s \n", JoinPoint::signature() );
  }
};
```

Figure 3.3: AspectC++ program example.

In addition to the four pointcut function previously discussed, other pointcut functions are provided in order to filter or select join points with specific properties. A summary of AspectC++ pointcut functions is presented in Table 3.1.º

Table 3.1: AspectC++ pointcut functions.

Pointcut function	Kind	Application
call(pointcut)	function	Selects calls to functions described by the pointcut parameter.
execution(pointcut)	function	Selects functions implementations described by the pointcut parameter.
construction(pointcut)	class	Selects class construction implementations described by the pointcut parameter.
destructor(pointcut)	class	Selects class destruction implementations described by the pointcut parameter.
within (pointcut)	scope	Filters all join points that are within the functions or classes in the pointcut.
cflow(pointcut)	ctrl flow	Filters all join point inside the dynamic context of joint points in the pointcut.
base(pointcut)	type	Returns all base classes of classes defined by the pointcut.
derived(pointcut)	type	Returns all classes in the pointcut and all classes derived from them.
that(type pattern)	context	Returns all join points where the C++ **this** pointer is related to the type pattern.
target(type pattern)	context	Returns all join points where the target object of a call is related to the type pattern.
result(type pattern)	context	Returns all join points where the result object of a call/execution is related to the type pattern.
args(type pattern,…)	context	Returns all joint points which match the provided argument signature.

In Table 3.1 we can see that besides the normal pointcut functions related to functions and class construction/destruction, there are others related to scope, dynamic control flow, base and derived types and context matching. The type patterns defined as parameters of the *that*, *target*, *result* and *args* pointcut functions can be used to convey context information to code advices.

Like C++ classes, aspects can have data members, constructors and member functions, and derive from classes or from other aspects. Pointcuts can be defined as virtual or pure virtual, which allows its redefinition in derived aspects. Aspects that contain pure virtual member functions or pure virtual named pointcuts are called **abstract aspects**. An aspect that inherits from an abstract aspect and defines all pending virtual member functions and virtual pointcuts is called a **concrete aspect**. Aspects can inherit only from ordinary C++ classes and abstract aspects.

3.2.3 The JoinPoint API

The aspect weaver creates a unique class for each join point affected by a code advice that needs context information. This class is called the **JoinPoint structure** or **JoinPoint API** and provides the context information to the code advice, such as method parameters types and values, and method return type and value. It also provides context information about calling and target objects, and other useful information as the method signature and the number of arguments. For each affected join point only the necessary context information is included in the JoinPoint structure. This feature is very important to keep a low memory footprint in embedded systems.

If context information is needed, an object of the JoinPoint structure is created in each affected join point, and a pointer to this object (the **tjp** pointer) is passed to an inline template function which also receives the JoinPoint structure as a template argument (the **JoinPoint** type). This inline function will call the real code advice, implemented as a template method of a class representing the aspect it belongs. In summary, each aspect will be transformed into a C++ class and each code advice related to this aspect will be transformed into a template member function. The aspect class member functions will be called by code inserted at the affected join points,

which will pass a pointer to an unique JoinPoint structure carrying the context information. Each non-abstract (concrete) aspect results in a singleton object.

An example of how AspectC++ weaves is shown in Figure 3.4. This figure presents how the source code resembles if a *before* advice is applied at an execution join point. The source code was simplified and methods/classes created by AspectC++ were renamed and shortened.

```
//----------  base code after weaving

struct TJP_XYZ {
  //...
  inline static const char *signature () {
      return "void ClassA::methodA(int,short int)";
  }
};

void ClassA::methodA ( int  arg0, short  arg1 ){
  TJP_XYZ jp;
  //... here the jp object is initialized

  AC::invoke_myAspect_ABC<TJP__XYZ> (&jp);

  this->__exec_old_methodA(arg0, arg1);
}

inline void ClassA::__exec_old_methodA(int aa,short int zz){
   // original methodA implementation
}
```

```
//----------- aspect code after weaving

class myAspect {
public:
  static myAspect *aspectof () {
    static myAspect __instance;
    return &__instance;
  }

  template<class JoinPoint> void ADVICE_1(JoinPoint *tjp){
     // advice code
     // here the JoinPoint type and the tjp pointer are seen.
  }
};

namespace AC {
  template <class JoinPoint>
  inline void invoke_myAspect_ABC (JoinPoint *tjp) {
    ::myAspect::aspectof()->ADVICE_1(tjp);
  }
}
```

Figure 3.4: Example of source code transformation by AspectC++.

The upper part of Figure 3.4 presents the base code after weaving. The JoinPoint structure *TJP_XYP* is declared closer to the implementation of the affected method (*methodA*). An object of this type is created inside *methodA* and its data fields are initialized, if any. Then, the *before* advice is called, using an inline template function *invoke_myAspect_ABC*. After returning from the advice code, the original implementation of *methodA* is called, now renamed by the weaver to *__exec_old_methodA*.

The aspect code after weaving, shown in the lower part of Figure 3.4, contains the definition of the inline template function called by *methodA*. This function redirects the call to *ADVICE_1*, a template method of the aspect class (*myAspect*), where the advice code is located.

AspectC++ is not able to advice data member accesses as AspectJ does using the *get* and *set* pointcut functions. This design decision was made because of the possibility of accessing variables with pointers in C++, which cannot be captured as a join point by the aspect weaver. However, an extension of AspectC++ described in [78] offers this functionality, but without considering pointer accesses. Another unimplemented feature of AspectC++ is template weaving. The weaver is able to parse C++ templates but weaving is restricted to non-templated code. However, support for template weaving is planned for future versions.

3.2.4 Performance and memory footprint

The work in [74] presents a series of micro-benchmarks for the main AspectC++ features, based on consumed CPU time (clock cycles), and memory (code/data and stack), in a Pentium 3 computer using the GNU g++ 3.3.5 compiler. The work in [106] extends the same experiment for the Intel C++ compiler icc 9.0. The results show that code advices (before, after or around) applied to parameterless functions have a very small runtime overhead (only 2 cycles) and no extra memory consumption. Considering functions with parameters and the application of the *JointPoint* pointer (tjp), there is an increase in stack consumption for the *tjp* pointer and the function parameters. However, the runtime overhead to retrieve join point specific context is quite low (0 to 6 cycles). The overheads for dynamic pointcut

functions as *cflow*, *that* and *target* (see Table 3.1) are relatively high (6 to 10 cycles), as they require the testing of runtime conditions. They also consume more memory, with a maximum of 50 bytes in the worst scenario. All overhead data presented here was obtained using compiler optimization. The same test cases without compiler optimization lead to much worse results.

Another work regarding AspectC++ performance and cost is presented in [75]. This paper reports an experiment comparing three different implementations of a weather station embedded software product line: C-based, OOP-based and AOP-based. The results show that the OOP version requires significantly more memory space than its AOP counterpart (up to 138% more), and that AOP requires at most 10% more memory space than the C version. Moreover, the runtime performance of AOP was the same as the C version, while the OOP version overhead was between 4 and 6.6%.

3.3 AOP for the operating system

Coady et al. [29] reported the application of AOP in the FreeBSD operating system kernel to modularize the prefetching of virtual memory mapped files. The prefetching mechanism implementation was spread in several functions over three different layers of the operating system code. After refactoring, the prefetching modes were implementing by single aspects, using AspectC, a subset of the AspectJ language for the C language, developed by the authors at University of British Columbia. The AOP solution presented several advantages over the tangled implementation, such as configurability, independent development and better comprehensibility. In a follow up work [30], the authors implemented other crosscutting concerns in the FreeBSD code, such as page deamon wake up, disk quota management and device blocking, analyzing the evolvability of AOP implementations in several OS versions. They concluded that AOP brings several benefits, such as localized changeability and explicit configurability. Unfortunately, the AspectC weaver was not officially released.

PURE is an object-oriented (C++) operating system designed for embedded applications by the University of Magdeburg [23]. Several works have been reported with regard to the separation of crosscutting concerns in PURE using AspectC++. Mahrenholz et al. [80] described how aspects can be woven with the operating system kernel for monitoring task switches. The work in [79] presented the implementation of the interrupt synchronization strategy in PURE using AOP. The same concern was implemented in the PURE successor, named CiAO [73], as described in [76]. In this work, several interrupt synchronization strategies (e.g. hard synchronization and two-phase synchronization) were implemented by aspects and could be selected at compile-time. The application of mutual exclusion mechanisms at PURE components using AOP was described in [105].

A quantitative analysis of the application of AOP in the ECOS [39] operating system is reported by Lohmann et al. [74]. In this work, the ECOS kernel was refactored to implement as aspects the following crosscutting concerns: tracing, interrupt synchronization and kernel instrumentation. Additionally, the implementation of configuration options in the OS was changed from conditional compilation (ifdefs) to aspect-oriented mechanisms. Each configuration option was encapsulated into a single aspect that applies introductions or code advices to implement the optional functionality. The AOP version of the OS showed at average a 0.9 % higher code size and a 1% better performance. The authors conclude that the application of AOP (using AspectC++) for the modularization of crosscutting concerns and the implementation of configuration options in operating systems does not induce intrinsic overheads.

In Chapter 6, we describe the utilization of AOP to implement fault tolerance mechanisms such as executable assertions at the operating system level

3.4 AOP for the middleware

Zhang and Jacobson [124] studied the degree of separation of concerns in the internal implementation of CORBA middleware platforms and the advantages of AOP in refactoring these systems. They developed an aspect mining methodology

supported by a software tool, and reported several new concerns to the platforms chosen (JacOrb, ORBacus and OpenOrb), as the dynamic programming interface and the support for portable interceptors. They also measured the degree of scattering related to normal concerns like logging, synchronization, exception handling and pre/post-condition checking. Additionally, they reimplemented some concerns in the ORBacus middleware using AspectJ and applied a set of metrics to the original and the refactored implementation. They concluded that AOP lowers the complexity of the middleware architecture, increases modularity, maintains the performance, and allows a higher level of adaptability and configurability, which is needed to customize platforms for particular domains such as real-time, embedded, and fault-tolerant systems.

The application of AOP to a large-scale middleware product line was reported by Colyer and Clement [31, 32]. In this work, an IBM commercial middleware with more than one million lines of code and hundreds of developers had several of its concerns refactored using AspecJ, as tracing/logging, exception handling and performance monitoring. The general approach was to develop a single abstract aspect for each of these concerns, defining a common policy (when and how), and several concrete aspects for defining the scope of application (where). This approach changes the way the policy team work: instead of delivering policy documents, they can implement the policy by writing the abstract aspects. As a consequence, policy compliance is more accurate and any policy evolution can be implemented with less efforts.

Colyer and Clement [32] also reported an experiment with AOP in an application that uses a middleware support extensively. They modified an application server software in order to separate the usage of Enterprise Java Beans (EJB) from the rest of the application. This problem could only be solved by heterogeneous aspects that impact on multiple places, but with different behavior in each of these places. The base code was refactored by the removal of EJB related code and the creation of hook methods that will be affected by advices. The woven application server software presented significant improvements in startup time and memory footprint. The main problem is that the aspect code is very dependent on the base code and cannot be reused in other projects. However, the refactoring simplifies the base code and allows

selecting or not EJB support at compile-time. The authors argue that a similar solution using plain OOP would be much more complex because of the huge number of variation points.

Ceccato and Tonella [26] described how to migrate an existing non-distributed application in a Java Remote Method Invocation (RMI) distributed application, based on the AOP application. They state that making an application run in a distributed environment involves many modifications that are spread and intertwined with the original code. Their solution is able to keep the original application oblivious to the distribution concern, which is regarded as a clear advantage in code understandability and maintainability. Aspects are created for the several Java RMI implementation issues, as remote interfaces, object factories, method invocation, parameter passing and exception handling. Code generation employs TXL [116] and AspectJ.

An experiment on middleware specialization using AspectC++ was conducted by Kaul and Gokhale [59]. Their motivation was to increase the performance and reduce the memory footprint of middleware platforms by using aspects to include only the needed features and to perform its optimization. They carried out a case study involving different concurrency models in the ACE middleware [1]. AOP was used to define the thread model and to implement part of its functionality, aiming to improve the system performance. Their results show that the AOP version presented smaller latency (3 to 4%) and larger throughput (2 to 3%) than the original OOP implementation.

In Chapter 6, we describe how to integrate a fault tolerance framework, which is considered an additional middleware, into the operating system code, using Aspect-Oriented Programming.

3.5 Fault tolerance using AOP

This section describes the related work in implementing fault tolerance using Aspect-Oriented Programming. Although fault tolerance (fault handling and dependability) is considered a non-functional concern and it is commonly cited as one

of the problems that AOP can address [56, 62, 73], few works combining fault tolerance and AOP have been reported.

Herrero et al. [53] created a replication model based on aspect-oriented techniques. The replication aspect can be defined by the JReplica language or by special UML extensions. A visual tool was being developed to generate JReplica code from UML. In this work, computational reflection is used to separate the functional level from the aspect level, but no information is given about how the final code is generated. Input messages to functional objects are intercepted and redirected to the aspect level. Output messages from objects are also intercepted and adapted to the right middleware (e.g. CORBA or JavaRMI). Only passive replication is supported. Replication mechanisms are implemented by aspects that define when state messages are exchanged, and the behavior for error detection, notification and recovery. The replication aspect can be composed with aspects developed for other concerns, as for instance synchronization.

Gal et al. [46] proposed the use of aspect-orientation in real-time systems for the distribution, timeliness and dependability domains. An example of the application for each domain is given, using CORBA in a logging application as test case. Aspects are implemented in AspectC++. The example for timeliness is based on execution time surveillance using a watchdog timer, which raises an error condition if an execution time budget is exceeded. The example for fault tolerance is based on the replication of the logging messages to several stations, but no fault tolerance strategy is applied.

Kienzle and Guerraoui [63] question if it is suitable to use AOP techniques to separate concurrency control and failure management concerns from the functional code. They conclude that the answer is no, because they feel that this separation is hard and potentially dangerous. They applied AspectJ in a case study based on transactions, analyzing three basic approaches: (1) aspectizing transactions uniformly in the whole program, (2) aspectizing transactions homogeneously in selected objects and methods, and (3) aspectizing transactions heterogeneously in selected objects and methods. They concluded that the first approach is impossible, the second approach yields poor performance and that applying the third, heterogeneous aspects result in functional code semantically coupled with the non-functional part, and consequently any maintenance in the functional code should trigger a modification on the

transaction aspects. A comment on this paper written by Kiczales [12] states that the AOP goal is not making a concern transparent, but instead making its implementation modular. In Kiczales opinion, the performance of critical concerns like distribution, failure handling and concurrency cannot be made totally transparent.

Szentiványi and Nadjm-Tehrani [109] reported a work for improving the performance of a FT-CORBA implementation by applying AOP at the application level. In this work, the logging of method executions needed for passive replication was shifted from the FT middleware to applications using AspectJ. Synchronization and method logging aspects are woven with base code update methods. The capability of advising data field access (*set* join point) in AspectJ allows the synchronization of variable accesses within the update methods. Using AOP, the overhead for passive replication was reduced by around 40%.

Alexandersson et al. [9] address the question of whether AOP can provide a base to implement fault tolerance mechanisms in non-distributed environments, termed "node level fault tolerance". This work presents examples of aspects for single node computing, such as time-redundant execution, assertions and Recovery Blocks, using AspectJ. An AOP recovery cache mechanism, needed for backward error recovery, was implemented using the *set* join point of AspectJ. The time-redundant mechanism applied in this work is a sequential software-implemented TMR. If the first execution results do not agree with the second execution results, a third execution is performed. The computation is defined by a class method and the results are the returning object. Assertions are implemented by application-specific aspects that check inputs and results of the selected methods and raise exceptions in case of failure. Recovery Blocks is implemented using one abstract aspect that defines the FT algorithm and application-specific concrete aspects that define the selected methods and introduce the new methods, such as the acceptance test and the alternative computation. Similarly to the time redundant mechanism, failures are handled by exceptions. The authors conclude that AOP is well suited to implement node level fault tolerance.

The work presented above was later reimplemented using AspectC++ because the authors' research targets embedded safe-critical systems [10]. For using AspectC++ they developed some extensions to the official AspectC++ distribution, such as the inclusion of *set* and *get* join points for primitive data types and their

pointers. This extension does not cover object data types because the assignment operator can be overloaded or advised by an aspect.

In Chapter 6, we propose the utilization of AOP to introduce fault tolerance at the application-level, based on the FT framework described in Chapter 5. This approach differs from Alexandersson' (reference) by defining the thread as the basic unit of fault tolerance and by targeting distributed environments.

3.6 Summary

The separation of concerns concept has been applied in software engineering for a long time. New techniques for dealing with crosscutting concerns have been proposed recently, such as meta-programming, composition filters and AOP.

AOP is a new programming technique to support the programmer in cleanly separating the functional components from crosscutting concerns, which are implemented as aspects, providing a mechanism to compose them and produce the overall system. The key concepts in AOP are join points, pointcuts and advices.

AspectC++ is a language extension to C++ that allows writing aspects and weaving them with the base code using source-code transformation. The main features of AspectC++ have been discussed, including the description on how aspects are composed to the main functionality. Additionally, the research work about AspectC++ performance and memory footprint has been presented.

This chapter has also reviewed the related work regarding the application of AOP to operating systems, middleware and fault-tolerant systems. General purpose and embedded operating systems have been submitted to AOP implementations of crosscutting concerns, such as performance optimization and interrupt synchronization, with good results in maintainability and resource utilization. Middleware platforms and their applications are the main target of AOP so far. Several works have reported middleware refactoring with AOP with excellent results in reducing complexity and increasing configurability and maintainability. Finally, the few works published about the application of AOP for fault tolerance have been presented.

Chapter 4

BOSS operating system

This chapter describes the main features of the BOSS operating system. A brief introduction about BOSS principles, history and applications is presented, followed by a detailed description of the BOSS kernel and middleware. Finally, the middleware extensions developed in this work are presented.

4.1 Introduction

Since 2004, the Embedded Systems Research Group (ESRG) [41] at University of Minho and the Fraunhofer Institute for Computer Architecture and Software Technology (FIRST) [43] have been cooperating in the field of dependable embedded systems, based on the joint research of efficient and adaptable fault tolerance technologies applied to real-time operating systems and middleware.

BOSS is a real-time operating system (OS) developed by FIRST. Its main design principle is irreducible complexity, which means that the OS design aims to achieve the minimum complexity in delivering a basic set of functionalities [81]. The objective is to keep the OS simple and understandable, as complexity is the cause of most development faults in software. Another advantage of this approach is to make possible the validation of the OS critical parts by formal methods. BOSS targets high-dependability applications, such as satellite and medical systems.

BOSS uses object-oriented technology in C++ extensively; it is fully preemptive and presents low interrupt latency and thread switching time. It has been ported to x86, PowerPC, Atmel AVR and ARM platforms. Additionally, an on-top-of-Linux implementation is available, and it is used mostly for early testing. BOSS simplicity makes easier the task of porting it to other platforms. Communication support is provided for Ethernet and CAN networks. Furthermore, a non-preemptive version of BOSS was developed, named TinyBOSS, targeting platforms with very limited resources.

The BOSS microkernel has mechanisms for resource management and synchronization, such as semaphores and signal boxes; for inter-task communication, such as messages and mailboxes; for interrupt handling; and for input/output (I/O). The basic OS constructs are implemented in BOSS as classes that can be configured and extended by inheritance. This represents a great advantage over conventional operating systems developed in procedural languages, such as C, which are usually hard to understand and modify.

Middleware communication in BOSS is performed using a publisher-subscriber protocol. Threads send messages locally or over the network by using a string as subject, or topic. Messages are delivered to all objects which subscribe to the same

subject. This loose coupling between senders and receivers makes fault tolerance implementation easier because the communication between threads is location-transparent and dynamically changeable.

The main application of the BOSS operating system was in the BIRD (Bispectral Infrared Detector) satellite control system [81]. This micro satellite weights 92 kg and was launched in 2001 by the German Aerospace Agency (DLR) to detect fires larger than 12 m^2. In this system, four processor boards (PowerPC 623) running BOSS applications are used. One node acts as a worker, doing all the required computation, while it is being constantly checked by a supervisor node. In case of failure in the worker node, the supervisor assumes as worker. If the failed node is unable to assume as supervisor after a re-initialization, one of the two spare nodes is activated and becomes the new supervisor node. The system proved its reliability in some bursts of solar activity that exposed the system to high energy radiation and particles [24], which generated transient faults.

The BOSS operating system is also applied in CubeSat satellites. CubeSat is a standard for a research pico satellite with dimensions 10x10x10 cm^3, weighing no more than one kilogram. The Technical University of Berlin is developing a CubeSat project named BEESat [22]. TinyBOSS was selected as the operating system for the board computer [60, 82], which uses an ARM-7 processor at 60 Mhz.

Another future application of BOSS is the HiPerCAR project [118]. This project is funded by ESA and aims to provide a dependable architecture for space autonomous robotics using limited resources. At the hardware level, HiPerCAR combines radiation hardened computers with commercial computers for achieving fault tolerance with high processing power. This system configuration includes one reliable master node and several COTS nodes acting as workers. Each system function can run in a worker node, in a nominal version, or in the master node, in the basic version. The nominal software version implements the full functionality, but the basic version only guarantees the safe operation of the system. After a failure in a worker node, the master node must assume his functions promptly and try to reboot the faulty node. In case of a permanent failure in the worker node, the master node must promote a system reconfiguration using spare worker nodes.

DLR has also plans for using BOSS in its new micro satellite bus architecture termed Standard Satellite Bus (SSB) [83]. The architecture will be similar to the BIRD architecture, using also four nodes and a worker/supervisor with spares scheme.

4.2 Kernel services

In this section the BOSS kernel will be described, following a subdivision based in related functionalities as task processing, synchronization, inter-task communication and timing.

4.2.1 Task processing

In BOSS, tasks are implemented by subclasses of the *Thread* class. Figure 4.1 shows a class diagram with the main methods involved with task processing.

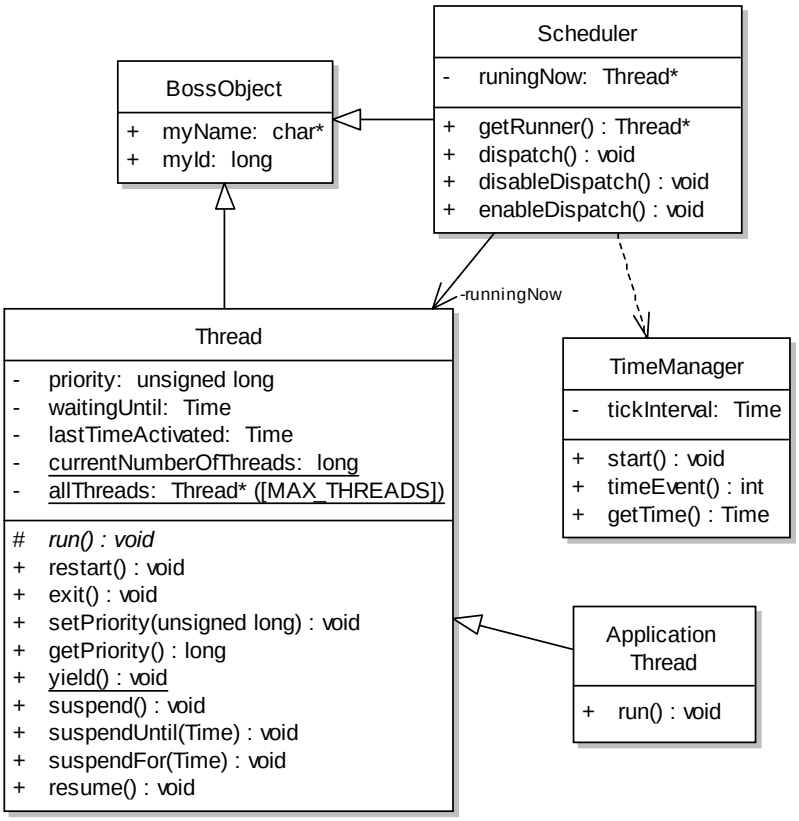

Figure 4.1: Task processing related classes.

The *Thread* class, as well as most BOSS classes, inherits from the *BossObject* class, which provides an optional name and identification number. Application threads must inherit from the Thread class and implement the *run* method, which will define the thread behavior.

BOSS uses priority-based preemptive scheduling. The priority attribute of the *Thread* class keeps the thread priority. Larger priority values represent higher priority levels. Thread priorities can be changed dynamically using the *setPriority* method. For threads of same priority, the scheduling policy selects the thread with oldest activation time, kept by the *lastTimeActivated* attribute. Time in BOSS is defined as a 64 bit quantity representing the number of microseconds passed since the system has started. The *Thread* class maintains an array of pointers, named *allTreads,* to all existing threads in the system, including the *Idle* thread. This array is used by the *Scheduler* class to select the next thread to run after a call to the *dispatch* method.

A thread can be in one of the following states: ready-to-run, running and suspended, as shown in Figure 4.2. After a thread object is created, the *restart* method prepares it for execution by setting up its stack and context information. The initial *restart* call for a thread is commanded by the operating system, but this method can also be called during the thread execution. After restarting a thread, all stack information is cleared and a call to the thread *run* method is performed. A *ready-to-run* thread can be selected for execution after a call to the *dispatch* method of the *Scheduler* class. From the *running* state, a thread can return to the *read-to-run* state if another dispatch takes places or if it calls the *yield* method. The *suspend* method causes a thread to go to the *suspended* state, while the resume method allows a thread to be ready to run again.

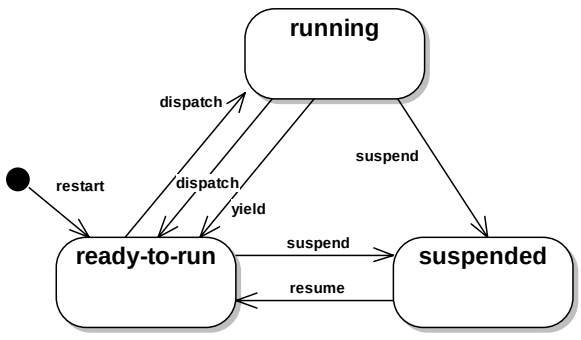

Figure 4.2: Thread states.

The mechanism of thread suspension or blocking in BOSS is implementing using a time variable (the *waitingUntil* attribute shown in Figure 4.1). If a thread needs to be suspended until a specific time, the *suspendUntil* method should be used. Alternatively, a thread can be suspended for a period of time, using the *suspendFor* method. The *suspend* method will suspend a thread forever, and it is in fact implemented by calling the *suspendUntil* method with the maximum possible value of time, which is never reached. The *Scheduler* class uses the *getTime* method of the *TimeManager* class to verify which threads are able to execute, depending on their time limit for suspension (*waitingUntil*). The *resume* method resets the thread suspension limit, making a thread ready for execution.

The *dispatch* method is called whenever a context switch is needed as, for instance, after the execution of *suspend*, *resume* or *yield*. Additionally, the dispatch method is called after each system clock tick. The clock tick interval is defined by the *tickInterval* attribute of the *TimeManager* class, shown in Figure 4.1. Besides, other interruption sources may trigger a dispatch, depending on settings defined in the related interruption management routines. The scheduler dispatch may be disabled by calling the *disableDispatch* method of the *Scheduler* class.

In BOSS, all threads share the same addressing space. Thread stacks are created in the system heap. However, the creation of kernel objects uses static memory allocation and these objects are never destroyed.

4.2.2 Synchronization

Synchronization can be classified into two categories: resource synchronization and activity synchronization [96]. Resource synchronization aims to achieve exclusive access to a shared resource, as a global variable, a data structure or an I/O device. Resource synchronization is also known as mutual exclusion. The section of the code that accesses a shared resource is termed critical section. In contrast, activity synchronization aims to ensure the correct execution order among cooperating tasks. Figure 4.3 contains a class diagram with all kernel classes related to synchronization in BOSS.

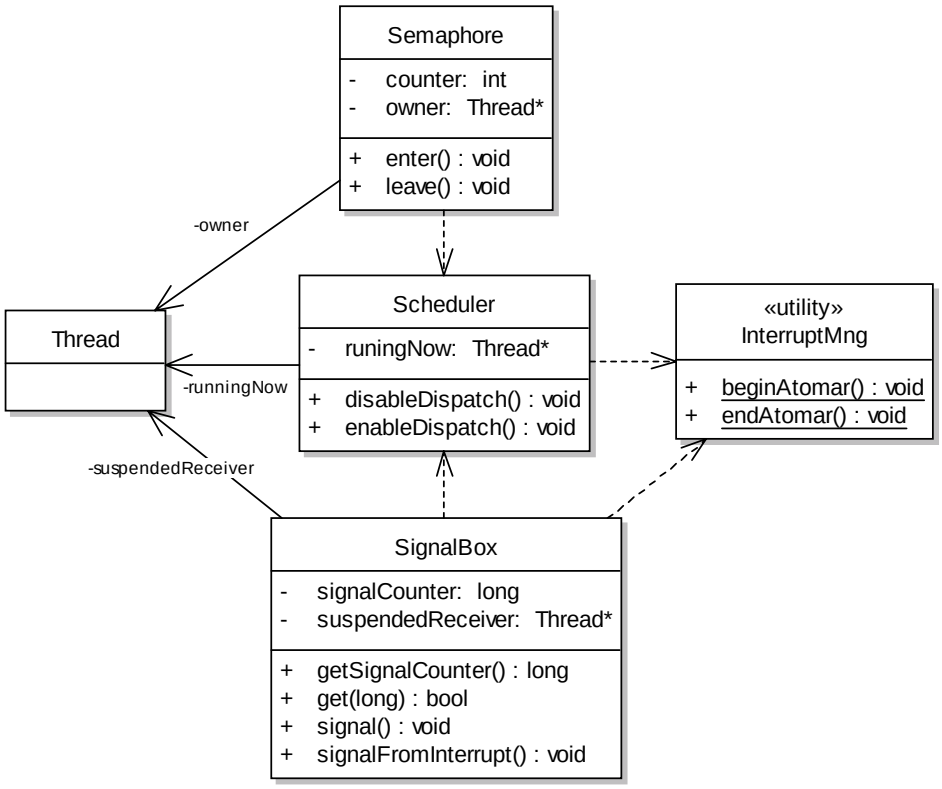

Figure 4.3: Synchronization related classes.

The following methods for supporting mutual exclusion are provided:

- **Interrupt locking**: this method consists of disabling system interrupts to synchronize exclusive accesses to shared resources between tasks and interrupt service routines (ISR). Interrupt locking affects the system interrupt latency and can be used to protect small and fast critical sections. Interrupt locking is provided by the global functions *beginAtomar* and *endAtomar*, which must enclose a critical section. Interrupt locking nesting is implemented by incrementing a global variable in *begingAtomar* and decrementing it in *endAtomar*. Interrupts are enabled by *endAtomar* only when this variable reaches its original value. As represented in Figure 4.3, several kernel classes use interrupt locking in their implementations, such as the *Scheduler* and *SignalBox*.
- **Preemption locking**: this method consists of disabling the task scheduler, or the dispatch mechanism. The application of this feature makes the scheduler non-preemptive as a low priority thread will no more be preempted by a higher priority task. However, preemptive locking does not synchronize resource

accesses between tasks and ISR. Preemptive locking is provided by the methods *disableDispatch* and *enableDispatch* of the *Scheduler* class. Similarly to interrupt locking, preemption locking allows nesting by using a global variable to control the nesting level. When this variable reaches the original value, the dispatch is executed and future dispatches are permitted. The *Semaphore* and the *SignalBox* classes use preemption locking in their implementation. The use of preemption locking for larger critical section affects high priority task reactions.

- **Semaphores:** this method of mutual exclusion mechanism causes a thread to be suspended upon calling the *enter* method if there is no resource available. BOSS implements mutex semaphores by default. Semaphores have ownership and calls to the *leave* method are only accepted by the owner. No priority inversion avoidance mechanism is implemented. When several threads are blocked in the same semaphore, the higher priority thread is released first. For equal priority threads, the one with oldest activation time is unblocked first.

The support for activity synchronization is provided by the *SignalBox* class. A signal box is a mechanism similar to a counting semaphore. Initially the *signalCount* attribute is set to zero, and its incremented each time the *signal* method is called and decremented when the *get* method is called. A thread will be suspended if it calls the *get* method when the *signalCount* attribute is zero. Differently than *Semaphore* objects, *SignalBox* objects can be used in synchronizations between ISR and threads. However, only threads are supposed to be signaled, as ISR must not be suspended. Furthermore, only one thread can be signaled, and a pointer to this thread is stored in the *suspendedReceiver* attribute of *SignalBox*.

4.2.3 Communication

The communication services of the BOSS kernel consist of passive classes that support safe data transfers between tasks and also between ISR and tasks. Figure 4.4 shows the main BOSS classes involved in communication services.

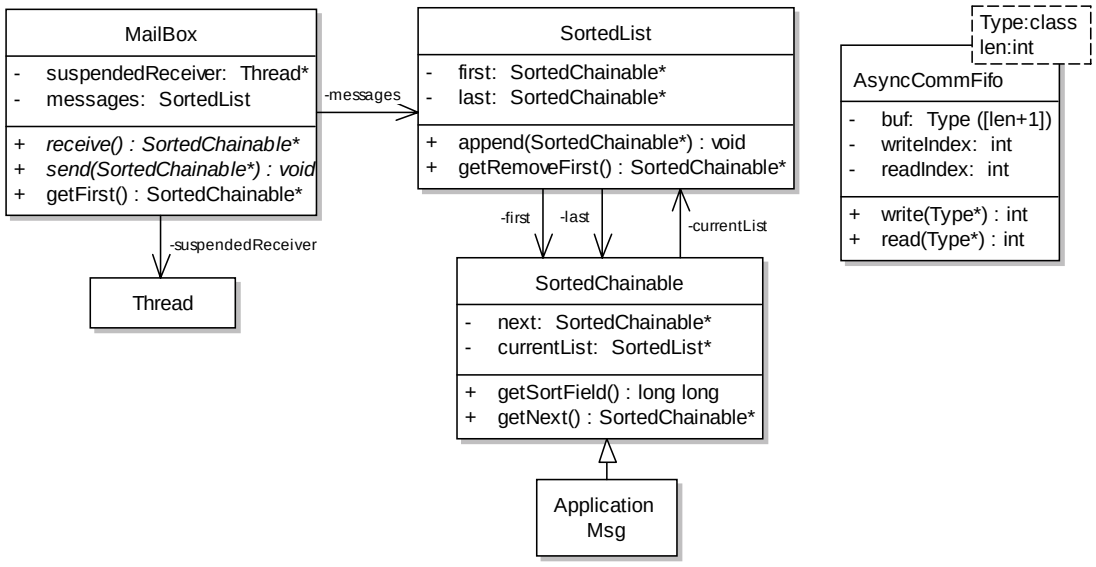

Figure 4.4: Communication related classes.

The *MailBox* class provides data communication between threads, similarly to a message queue. Several threads may send messages to a single receiver thread, which is suspended if it calls the *receive* method and no messages are available. Sending threads are never blocked and messages are stored in linked list data structure implemented by the *SortedList* class. The data message using *MailBox*, represented in Figure 4.4 as *Application Msg*, must be a subclass of the *SortedChainable* class, as it should have attributes and methods related to linked list node objects. Messages are delivered in a First in - First out (FIFO) basis, although *SortedList* objects are able to sort items using a priority field. Data objects sent to mail boxes are stored in a *SortedList* and no data copy is performed. Therefore, the sender and receiver threads are responsible for data objects creation and mutual exclusion.

The *AsyncCommFifo* class provides FIFO asynchronous non-blocking data communication between one sender and one receiver using a producer-consumer protocol. Senders and receivers can be either threads or interrupt service routines as they are never blocked. *AsyncCommFifo* is a template class that receives the type of the data objects and the internal buffer size as template parameters. The *read* and *write* methods copy these data objects to and from the internal buffer, respectively.

4.2.4 Utility classes

Besides the classes presented so far, the BOSS kernel provides several utility classes for supporting thread timing control, memory management and debugging. The main classes for timing control and memory management are presented in Figure 4.5.

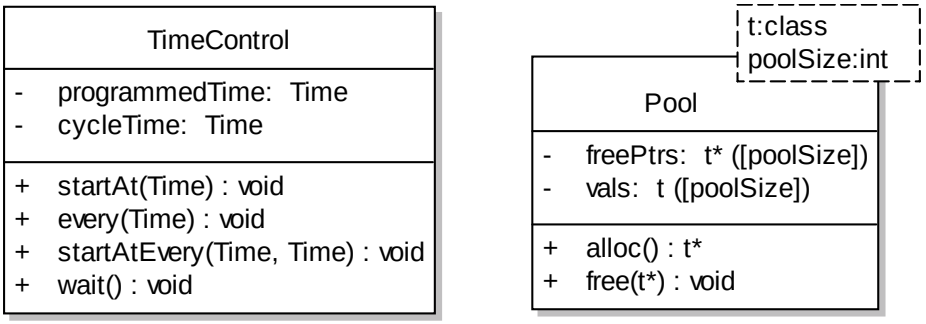

Figure 4.5: BOSS utility classes.

A thread can use one or more *TimeControl* objects to support the implementation of its temporal behavior. Each *TimeControl* object defines a startup execution time (*startAt* method) and a cycle time (*every* method). When a thread calls the *wait* method of *TimeControl*, a new wake up time is calculated and passed to the *suspendUntil* method of the *Thread* class.

The *Pool* class supports the creation and management of objects in static memory. It is a template class with two template parameters: the type and number of objects to be managed. The *alloc* method returns a pointer to an unused object, while the *free* method returns it to the pool. If the pool is empty the *alloc* method will return a null pointer. Multiple threads and ISR can share the same pool of objects as *Pool* methods are protected by mutual exclusion mechanisms.

4.2.5 Hardware interface and management

BOSS has a small Hardware Dependent Layer (HDL) which implements the platform-dependent functionality, as context switching and interrupts management. The interface between the BOSS kernel and the HDL is defined by C functions that

are implemented in one these layers. C functions are used to simplify calls from assembly code in the HDL to the kernel, and vice-versa. However, most HDL code is implemented in C and C++. Figure 4.6 shows the basic layers of the BOSS architecture.

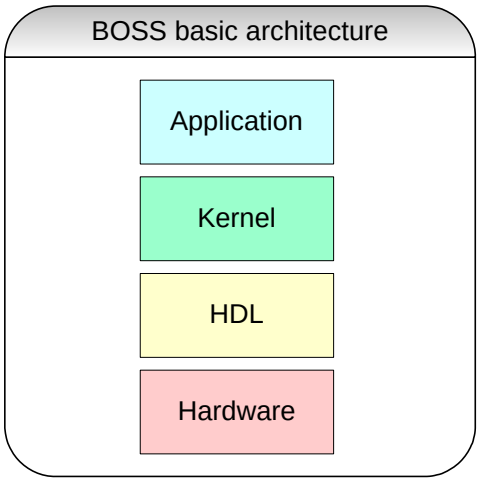

Figure 4.6: BOSS basic architecture.

The main functions of the kernel/HDL interface are presented in Figure 4.7. The *hwSetUp* function is called by the kernel to perform platform specific initialization, as setting interrupt vectors and configuring memory management. The *interruptsOn* and *interruptsOff* functions provide assembly code to enable and disable interrupts. The initialization of the clock tick timer is implemented by the *initTimer* function, which receives the clock tick interval as a parameter. The *getMicroSeconds* function returns the time base in microseconds since the system startup. The *setup* function is called by the *restart* method of the *Thread* class to initialize the thread stack frame. The context switch is performed by the *transfer* function and the *softReset* function resets the node and it may be used if an irrecoverable error is detected.

The kernel functions called by the HDL are described as follows. The *ThreadStartUp* function is the entry point of thread execution after stack initialization. This function calls the *run* method of the *Thread* class, which should not return; otherwise a node reset will take place. The *interruptPropagator* function is called by a general interrupt handler in the HDL to allow the execution of interrupt event services defined by application threads. The parameter *interrutptID* is used to identify the interrupt source and trigger the execution of the *eventServer* method of the Thread

class. Finally, the *dispatchCaller* function is called when leaving interrupt handlers if a dispatch is required.

```
// Kernel --> HDL
void hwSetUp(void);
void interruptsOn(void);
void interruptsOff(void);
Time initTimer(Time interval);
Time getMicroSeconds(void);
long *setup(long *stack, long stackSize, void *classRef);
void transfer(long **from, long *to);
void softReset(void);

// HDL --> Kernel
void ThreadStartUp(void * thread);
int interruptPropagator(int interruptID);
void dispatchCaller(void);
```

Figure 4.7: Kernel/HDL interface.

In BOSS there is no provision of mechanisms for installing and managing device drivers. The application program can access the hardware directly, making use of kernel objects and interrupt management support as needed.

4.3 Middleware services

Single node applications can be developed with the BOSS kernel classes described so far. However, support for multiple node and distributed fault-tolerant applications is provided by extra classes which implement a common communication paradigm both for intern and extern threads. This new level of functionality is termed middleware. BOSS middleware is based on asynchronous message-oriented communication using the publisher-subscriber protocol. In this section, the original BOSS middleware implementation will be described.

4.3.1 Message to message communication

The basic unit of middleware communication in BOSS is the *Message* class, shown in Figure 4.8.

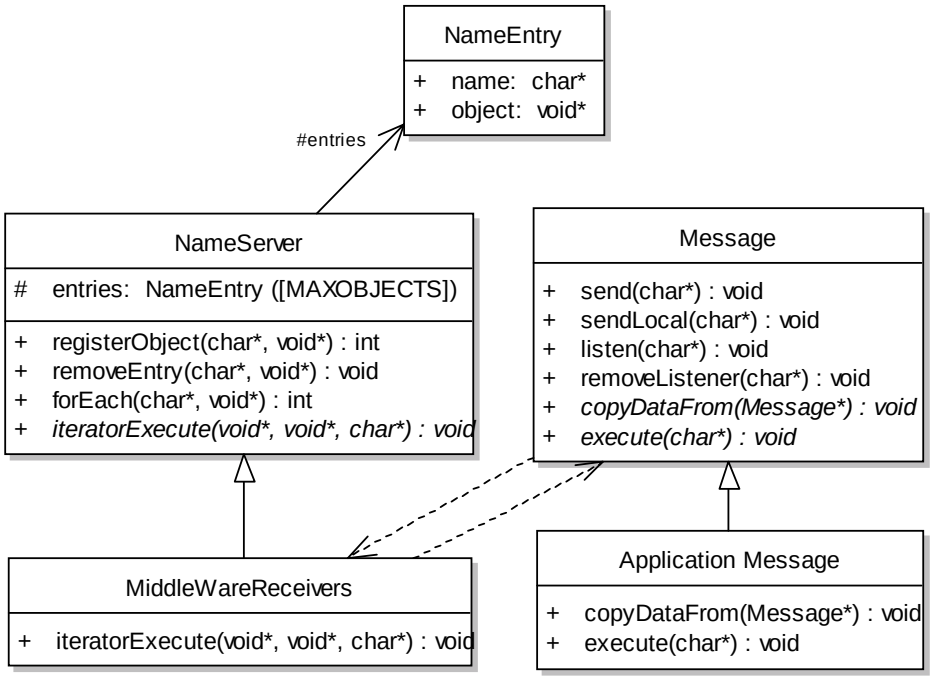

Figure 4.8: *Message* class diagram.

Application messages must inherit from the *Message* class and include data as class attributes. Besides, *Message* derived classes must implement the *copyDataFrom* method, which defines how the message data is updated with data from other message. In addition, it may optionally implement the *execute* method, which defines a specific behavior after message copying.

Figure 4.9 presents how the publisher-subscriber data structures are implemented. A *NameServer* object maintains an array of *NameEntry* objects that relates subjects to receiving messages. In the Figure 4.9 example, *Message1* and *Message3* are subscribers of the *subject1* subject. The same subject name can be subscribed by more than one message. Furthermore, the same message can subscribe more than one subject, as *Message3* does.

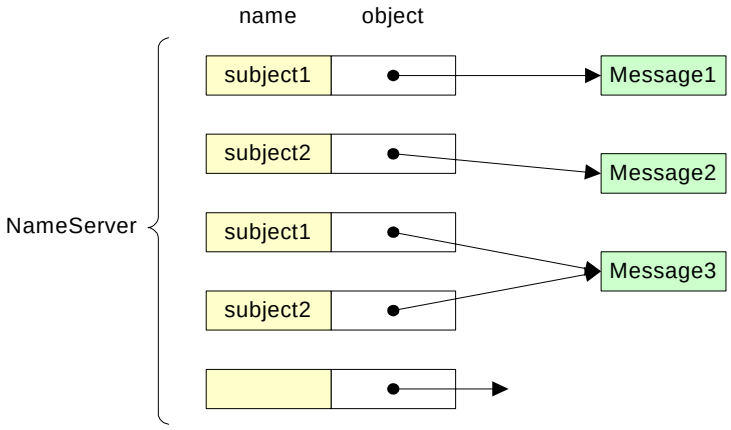

Figure 4.9: *NameServer* data structure.

A message is sent, or published, using the methods *send* and *sendLocal* of the *Message* class. For sending a message it is necessary to pass the subject name as an argument. The message distribution is performed by copying the data attributes of the receiving message to all messages that have subscribed the related subject. The distribution of messages is implemented by the *MiddlewareReceivers* class as shown the sequence diagram of Figure 4.10.

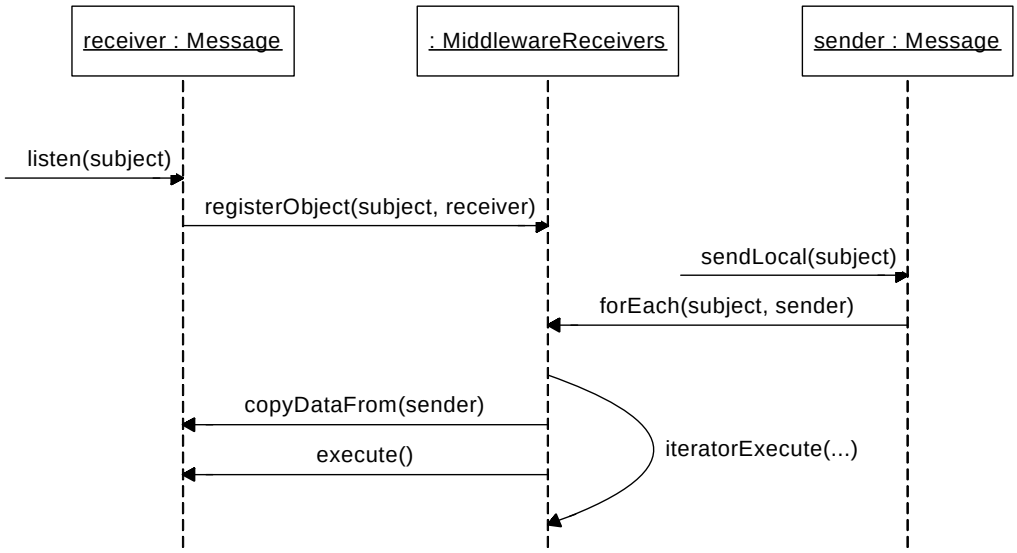

Figure 4.10: Middleware message distribution.

Initially, the *receiver* message registers his subscription to a subject by calling the *listen* method. This is accomplished by the *registerObject* method of the MiddlewareReceivers class, which sets up one entry in the data structure of Figure 4.9. When the *sendLocal* method of the *sender* message is called, the *forEach*

method of *MiddlewareReceivers* is executed, starting a search in the *NameServer* data structure for entries with the same subject name. When a match is found, iteratorExecute executes the *copyDataFrom* method of the receiving message, copying the data attributes from the sending message. Additionally, the *execute* method of the receiving message is activated, allowing the execution of the code related to message reception, as for instance, resuming a suspended thread.

The mechanism of overwriting the data of one message with the data of another message is referred here as message to message communication. Using this mechanism, message data can be written without previous data utilization. This is ideal for transmitting state messages, in which only the most recent data is significant. However, this mechanism is not suitable for event messages, in which messages convey a system event, as events may be overwritten and lost.

4.3.2 Message to thread communication

In order to deliver event messages, a thread messaging mechanism is provided. This includes support for message buffering and thread synchronization in message reception. The *IncommingMsgAdministrator* class, shown in Figure 4.11, supplies mail box functionality for threads. It is a template class that receives as template parameters an application message class and the message buffer size. Internally, *IncommingMsgAdministrator* maintains a memory pool of application messages using a *Pool* object, and a *MailBox* object to provide the mail box functionality.

The *IncommingMessageAdministrator* class derives from *Message* class, and therefore it can behave as a receiving *Message*, exactly as described in Figure 4.10. The implementation of its *copyDataFrom* method is presented in Figure 4.12. When this method is called, a pointer to a free message object is retrieved from the pool, a copy is performed using *copyDataFrom*, and the receiving message is sent to the MailBox.

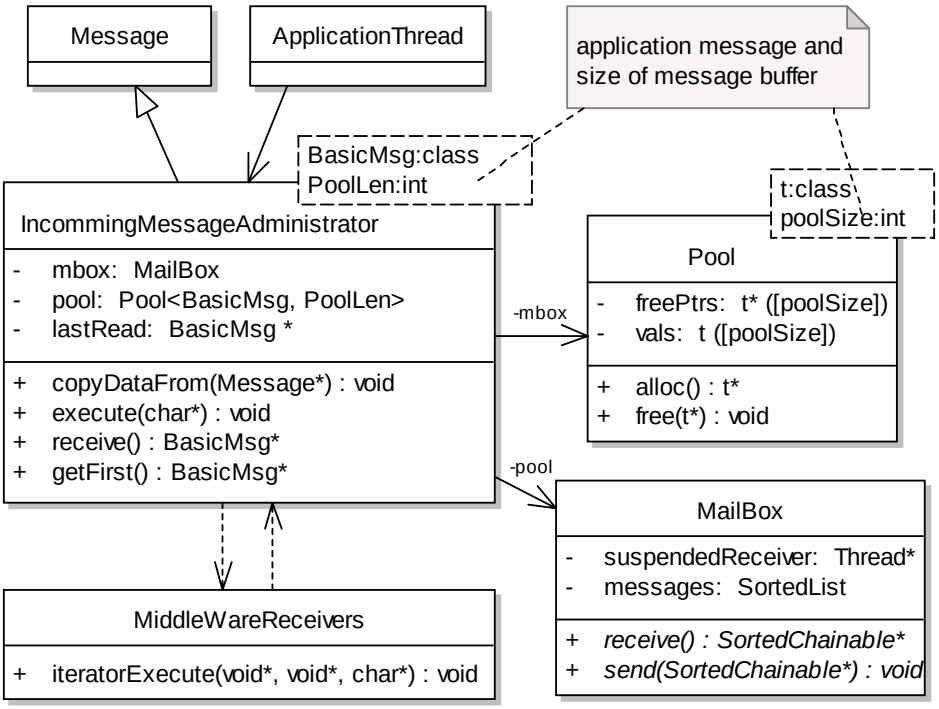

Figure 4.11: *IncommingMessageAdministrator* class diagram.

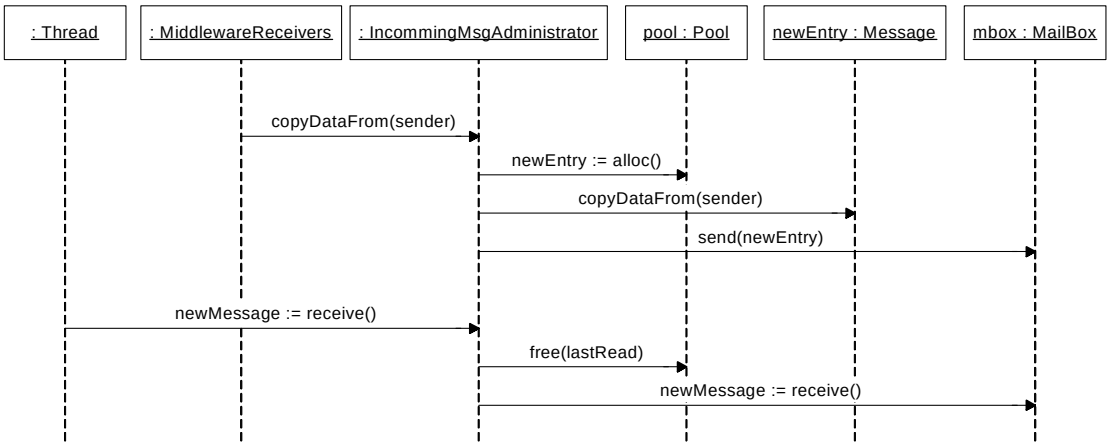

Figure 4.12: *IncommingMessageAdministrator* sequence diagram.

Threads receive messages by calling the *receive* method of *IncommingMessageAdministrator*, as shown in Figure 4.12. The previous read message is freed and sent back to the pool of message objects. Then, the *receive* method of the *MailBox* object is executed. If the mail box is empty, the thread will be suspended until a new message arrives.

4.4 Middleware extensions

The original BOSS implementation received from FIRST did not provide a generic mechanism for sending and receiving network messages. For instance, only the *sendLocal* method of the *Message* class was implemented by the kernel. The implementation of the *send* method, which is supposed to distribute a message both internally and externally, was application-dependent (an application function or method should be called). The same applied for middleware message reception, which had to be implemented by an application thread.

Additionally, the original implementation did not support message identification. Message identification is needed to discard duplicate messages and to implement voting algorithms. Figure 4.13 presents a TMR fault-tolerant configuration that will be used to discuss the reasons for providing message identification. In this configuration, three replicas of *Task A* receive the same input and send their results to three identical voters. The voter results are sent to three replicas of *Task B*. As voter output messages are redundant, *Task B* can process the first message received and discard the following messages. But for discarding messages it is necessary to recognize that they are related to the same input data. A possible solution is to include an identification number in the original input message and to retransmit this identification number in the output messages of *Task A* and *Voter A*. Besides, message identification is also useful for the voters because it provides information that can be used to detect if a new voting cycle has started.

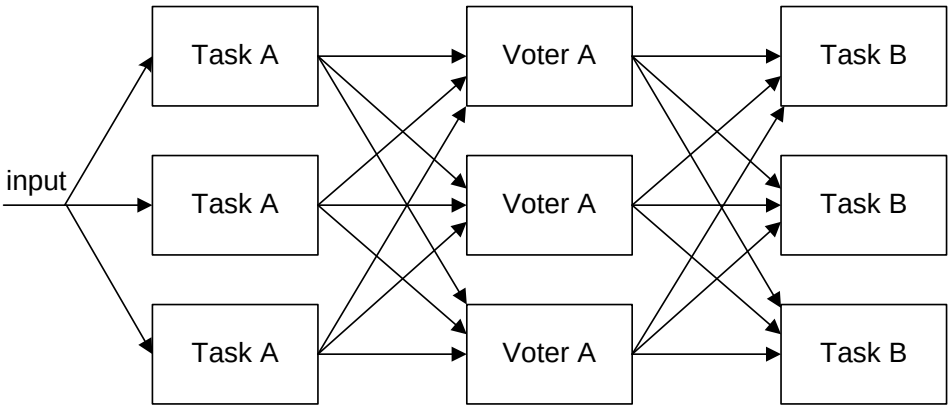

Figure 4.13: TMR configuration.

In the following sections, the new extensions to the BOSS middleware developed in this work will be presented.

4.4.1 Message identification and discarding

The implementation of message identification and discarding of duplicate messages by the middleware demanded the modification of the *Message* and *NameServer* classes. For the *Message* class, it consisted of the inclusion of the *msgID* attribute. The *listen* method of the *Message* class now accepts a Boolean value as a parameter, to define whether duplicated messages must be discarded or not. The default value for this parameter is false, meaning that no message discard is required. This option will be stored in the *discard flag* of the new *NameServer* implementation, shown in Figure 4.14.

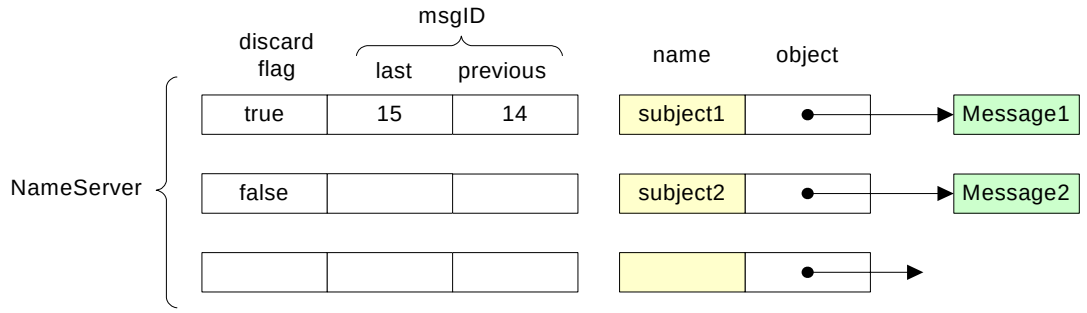

Figure 4.14: *NameServer* extension for discarding duplicate messages.

Besides the discard flag, two message identification attributes were added to each *NameServer* entry to store the last and the previous *msgID*. In the example shown in Figure 4.14, *Message1* was registered for discarding messages (discard flag equals true), the last delivered message had 15 as *msgID* and the previous delivered message had 14 as *msgID*. A new incoming message will only be delivered if it has a *msgID* different from 15 and 14. For instance, a new message with *msgID* of 16 will be delivered, and consequently the previous msgID attribute will receive the value of the last *msgID* attribute (15 in this case), and last *msgID* attribute will receive the *msgID* of the incoming message (16 in this case). If however, the new message has *msgID* of 15 or 14, it would be discarded and no modifications will be done to the last and previous *msgID* attributes.

Using the algorithm described above and considering the configuration presented in Figure 4.13, *Task B* would only receive the first message sent by a voter in each voting cycle. Any late arriving message from the previous voting cycle would also be discarded.

Message identifications are defined as "unsigned short" variables (usually 16 bits) and can be generated sequentially by sending tasks just but incrementing them for each new message. No special care is needed when they reach the maximum value (e.g. 65535) and return to zero because the discarding algorithm is not based on ordering. The only restriction is to avoid start sending messages with the maximum identification value because that is the value used to initialize the last and previous *msgID* attributes.

4.4.2 External messages handling

This section will describe the middleware extension mechanism to support the delivery of external messages. The *Message* and *MiddlewareReceivers* classes were modified by the introduction of new variables, data structures and methods, as shown in Figure 4.15.

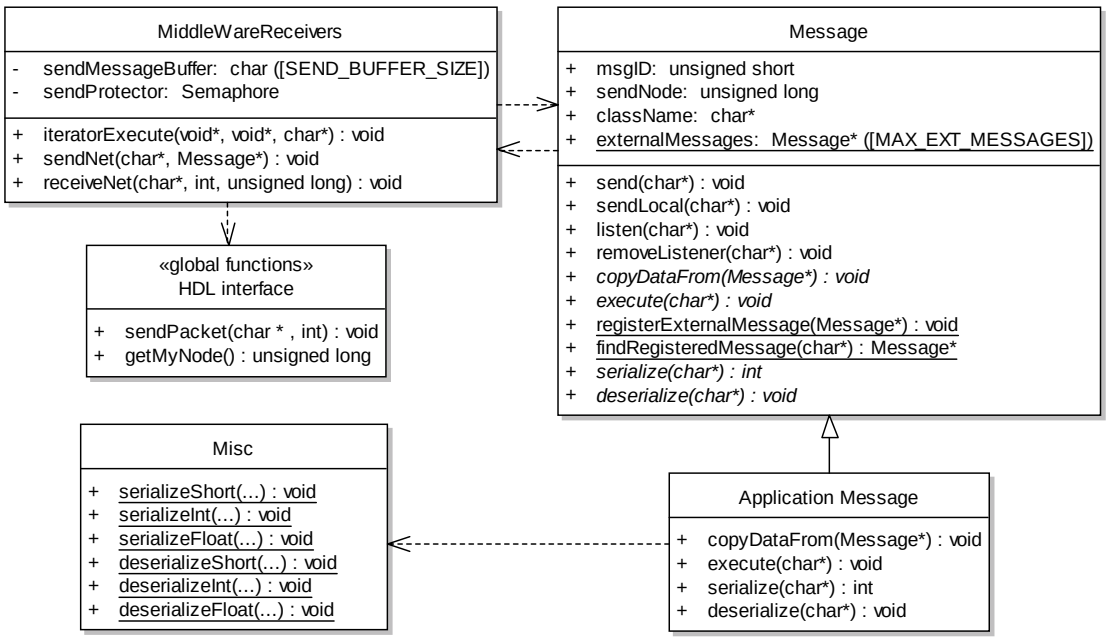

Figure 4.15: Middleware extensions class diagram.

In addition to the *msgID* attribute, already discussed in the last section, the *Message* class has been augmented by the *sendNode* attribute, which identifies the origin node of a message (e.g. IP number); by the *className* attribute, which stores the name of the subclass of *Message* that will be sent to an external node; and by the static *externalMessages* array of message pointers, which references auxiliary messages used in message reception. The *registerExternalMessage* method inserts an entry in the *externalMessages* array and the *findRegisteredMessage* method searches for an auxiliary message with a given *className* attribute in the same array. *Serialize* and *deserialize* are new virtual functions that must be implemented by external messages for marshaling and unmarshaling the message data. These functions may use the utility functions of the *Misc* class in the serialization and deserialization process.

The *MiddlewareReceivers* class has gained a data buffer named *sendMessageBuffer* to store the outgoing message data after serialization and a semaphore to protect it from multiple accesses from sending threads. Two new methods were added to *MiddlewareReceivers*: the *sendNet* method prepares the message for network transmission, eventually calling the *sendPacket* method of the HDL interface; and the *receiveNet* method distributes incoming messages, taking as input the data received by the HDL when a message arrives.

The process of sending and receiving external messages is shown in Figure 4.16. In the sender node, a message is prepared and the *send* method is called, passing the message subject as an argument. After that, the *sendNet* method of *MiddlewareReceivers* is executed and takes care of the message marshaling, by preparing the *sendMessageBuffer* according to the sequence diagram shown in Figure 4.17.

The *className* information is taken from the *sender* message object, as well as the *msgID*. The marshaling of the data message is performed by the sender itself, using the *serialize* method. All data is sent in network byte order (big-endian). The serialization functions of the *Misc* class are able to change the byte order for little-endian platforms. When the buffer is ready for transmission, a pointer to it, as well as its data size, are passed to the *sendPacket* function of the HDL interface, which

eventually will send the data over the network. Finally, the message is also sent locally using the *sendLocal* method.

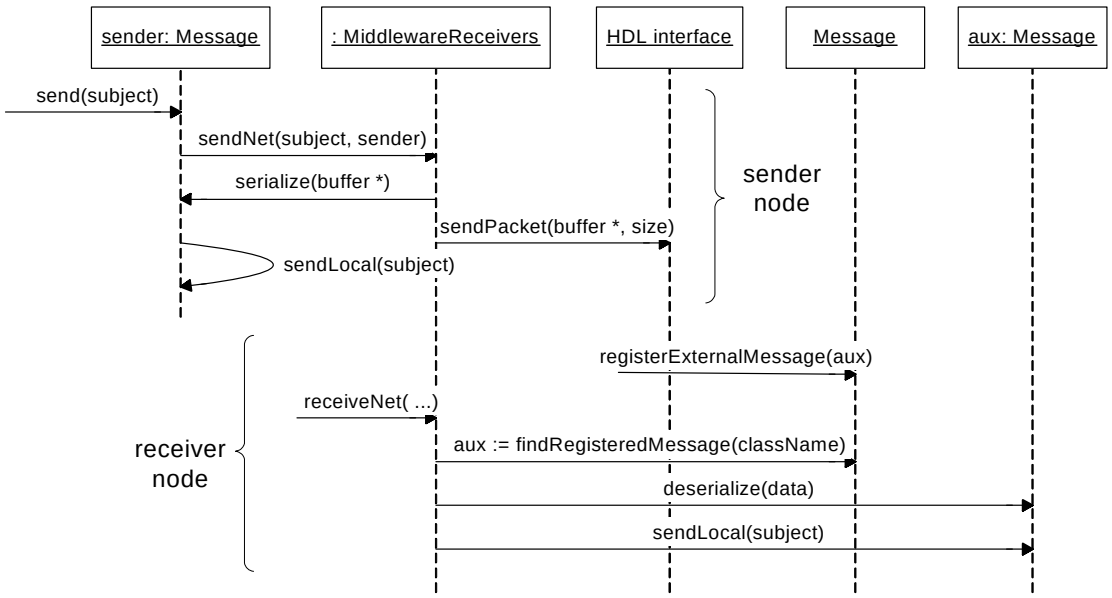

Figure 4.16: External messages processing.

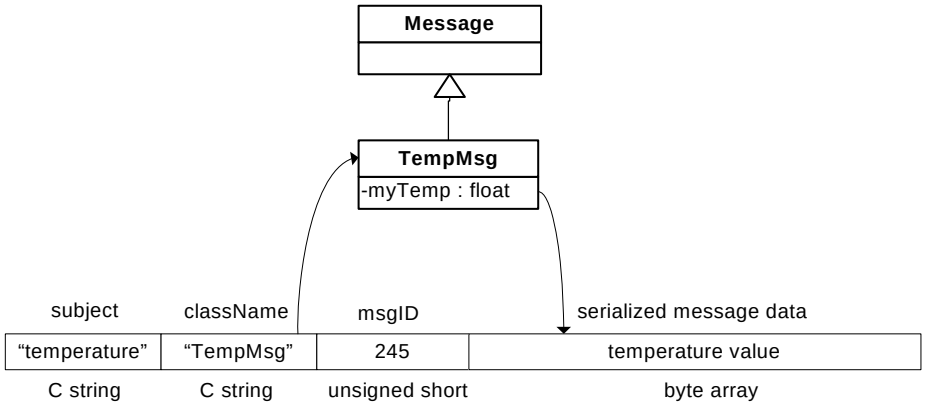

Figure 4.17: External message packet description.

In receiver nodes, the following processing takes place (Figure 4.16). Initially, an auxiliary message object of the sender message class must be created and registered using the *registerExternalMessage*. When a message is received by HDL, the *receiveNet* method of *MiddlewareReceivers* is called. Then, a receiving buffer, passed as an argument, is scanned for removing the *className* information, which is then

passed to the *findRegisteredMessage* in order to retrieve a pointer to the auxiliary message object related to the incoming message. If no matching auxiliary message exists the incoming message is discarded, otherwise the receiving data is copied to the auxiliary message. The *deserialize* method of the auxiliary message executes the unmarshalling of the message data. At this point, the auxiliary message is a replica of the original sender message, and it can be sent locally using the *sendLocal* method.

The mechanism described above provides fully transparent communication for applications. This means that the same application can run on different platforms (with possibly different byte ordering), and communicate with other applications without knowing in which platform they are running. Message objects can be sent locally and over the network using the *send* method, and only have to implement the *serialize* and *deserialize* methods. At the receiver's side, an object of the same message class must be created and registered at initialization time. No further procedures are needed to handle external messages at the application level. All platform and network dependent code is implemented at the Hardware Dependent Layer.

4.5 Summary

The BOSS operating system is a real-time OS designed for small-scale embedded systems with high-dependability requirements. Its object-oriented design aims the reduction of the operating systems complexity, which is the cause of most design faults. However, it covers all basic functionality needed to develop embedded applications, including the communication between nodes, using a publisher-subscriber protocol.

This work has improved the BOSS middleware by adding mechanisms for message identification and discarding duplicate messages. Furthermore, support for handling external messages was developed, making intra-node and inter-node communication transparent for applications. The information provided in this chapter is necessary for understanding the fault tolerance framework described in the next chapter.

Chapter 5

Fault tolerance framework

This chapter describes the fault tolerance framework developed for supporting application fault tolerance atop the BOSS operating system and its middleware. As an introduction, the framework objectives and constraints are presented. Afterwards, the framework is described in various levels of detail ranging from the application programmer perspective to specific FT strategy implementations. Finally, the benefits and drawbacks of the proposed FT framework are discussed.

5.1 Introduction

The FT framework designed for supporting application fault tolerance to the BOSS operating systems has several objectives and constraints. First, it has to be easily customizable and extensible, in order to support fault tolerance in a wide variety of projects with different dependability requirements and hardware availability. Therefore, it should provide mechanisms for hardware and software fault tolerance using single or redundant hardware systems with single or multiple software versions. Second, it has to be fully compatible with the BOSS operating system, using the basic features provided by this OS and the communication infrastructure provided by its middleware. Finally, it has to be simple and efficient to run in small-scale real-time embedded systems without incurring in too much resource consumption, such as processor and memory usage.

The FT framework described here focus in fault-tolerant computing and does not include mechanisms for tolerating communication network errors. In this work, it is assumed that the underneath communication is reliable and ordered.

As seen in Section 2.10, several object-oriented fault tolerance patterns and frameworks have been proposed and developed by the research community. In general, the unit of fault tolerance is an application object with behavior defined by a subclass of an abstract "variant" class. Considering objects as units of fault tolerance has also been applied in fault tolerance supporting systems such as FT-CORBA [88] implementations, although using replication without diversity. Other systems such as ROAFTS [68] use virtual objects (TMO objects in that case) as units of redundancy, but method calls are implemented as threads.

The chosen approach is to use BOSS threads as units of fault tolerance because threads and processes are the real units of computation in a multitasking system. Consequently, thread restarting can be employed as an effective mechanism of system recovery. The same mechanism can not be applied by object methods if, for instance, an error condition leads to an infinite loop execution. Besides, using objects (virtual or real) as units of fault tolerance increases system implementation complexity, reduces performance, and increases memory usage. The same approach was used in FT-RT-Mach [40] and in the AFT for Spacecraft work [52].

5.2 Fault tolerance thread model

Not all kinds of application threads can be used as units of fault-tolerant computing in the proposed framework. A fault-tolerant thread must comply with a specific fault tolerance thread model, shown in Figure 5.1.

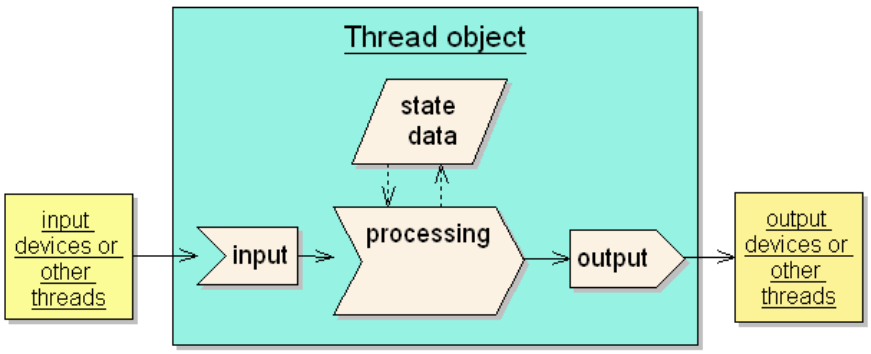

Figure 5.1: Fault tolerance thread model.

Fault-tolerant threads are supposed to read from input devices or receive input messages from other threads, process the inputs and generate an output either by writing to an output device or by sending a result message to other threads. The model supports both state threads and stateless threads. For state threads, the output result will depend both on the input data and on the previous state data. The input phase is optional, as a thread can be activated by a timing mechanism and may use no external data in the processing phase. However, the ordering of the input, processing, and output phases should be preserved. A thread performing inputs and outputs during the processing phase is non-compliant and can not be made fault-tolerant using this framework.

An example of a candidate thread for fault tolerance implementation is presented in Figure 5.2. In this example, *ExampleThread* runs cyclically, reading messages from an *IncommingMessageAdministrator* object, which consists of a mailbox for messages of the *Msg* class. The *process* method is executed next, and implements some computing algorithm using data from the incoming message and possibly from an internal state (attributes not shown). Finally the output method prepares the output message and sends it locally and over the network, using the string “exampleResult” as subject.

```
class ExampleThread : public Thread {
  Msg* recMsg;
  Msg outMsg;
  IncommingMessageAdministrator<Msg,20> inMessages;
public:
  ExampleThread(){ ... // init code}

  void run () {
    while(1) {
      recMsg = inMessages.receive();
      process();
      output();
    }
  }

  void process(){
    ... // uses msg data and state data
  }

  void output(){
    ... // prepares output message
    outMsg.send("exampleResult");
  }
};
```

Figure 5.2: Example of candidate thread for FT implementation.

The thread model explained above is commonly adopted in the design of fault-tolerant systems [7, 101]. Threads in this model behave like state machines, receiving events/data as inputs and, in consequence, changing their internal state and sending events/data as outputs. FT threads are not allowed to interact with other threads or to perform any input/output during the processing phase.

5.3 Framework general description

In this section the fault tolerance framework will be described in the perspective of the application programmer. The description approach is based on presenting how the framework can be used to modify an existing non-fault-tolerant application thread to make it fault-tolerant. The original non-fault-tolerant thread must comply with the fault-tolerant thread model presented in the previous section.

5.3.1 Framework structure

Figure 5.3 shows a simplified class diagram of the FT framework. A fault-tolerant application thread (e.g. *FTApplicationThread*) must inherit from the *FTThread* class and select an *FTStrategy* object that will implement the fault tolerance functionality. Three FT strategies have been implemented: RB, DRB and NVP, but others can be developed and integrated to the framework.

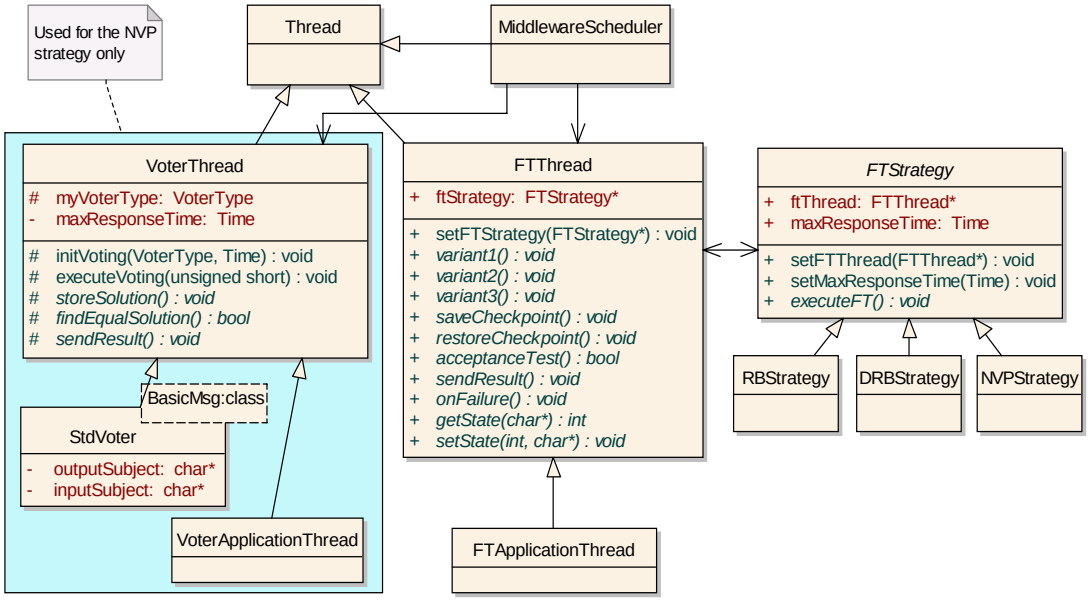

Figure 5.3: Simplified FT framework class diagram.

Differently from software structures presented in Section 2.10, where variants and adjudicators are represented by classes, here these functionalities are implemented as methods of the *FTThread* hierarchy. The *FTThread* class declares several virtual functions which must be implemented by the FT application thread, depending on the selected FT strategy, such as software variants, checkpointing support functions and the acceptance test. This approach has several advantages: (a) it simplifies the framework class structure; (b) it allows direct access from these procedures to class attributes defined by the application thread; and (c) it reduces runtime and memory costs.

The *VoterThread* class supports the development of voters, which are required by the NVP strategy. A voter application thread (e.g. *VoterApplicationThread*) must inherit from *VoterThread* and define some virtual functions, such as

findEqualSolution. Additionally, a standard voter class (*StdVoter*) is supplied. This predefined voter thread provides exact voting (bit-by-bit comparison) when both inputs and outputs are implemented by message passing.

The MiddlewareScheduler (MS) class controls all FT and voter threads. This thread periodically searches for active FT/voter threads and executes part of the required control algorithm. Besides, this thread triggers periodic middleware messages to perform role definitions and thread state synchronization.

5.3.2 Fault tolerance introduction

The modifications required to make an application thread fault-tolerant include:

- Instantiation and registration of an *FTStrategy* object that will implement the desired fault tolerance strategy, as RB, DRB or NVP.
- Execution of the *executeFT* method of the *FTStrategy* object after the thread activation.
- Implementation of application-specific methods related to the selected fault tolerance strategy (as the acceptance test in RB and DRB). Some of them consist of new functionality but others will contain the code originally defined in the processing and output methods.

Figure 5.4 shows an example of fault-tolerant implementation for *ExampleThread* of Figure 5.2, using the DRB strategy. The main differences between this version and the original code in Figure 5.2 are highlighted. The application thread now inherits from the *FTThread* class, instead of the *Thread* class. A concrete *FTStrategy* is instantiated as a *DRBStrategy* (*myDRB*). In the class constructor, the maximum response time for execution is set to 20,000 microseconds and the *setFTStrategy* method is called, assigning the address of the *DRBSstrategy* object to the *ftStrategy* pointer (see Figure 5.3). In the *run* method, the original *process* and *output* methods are replaced by a call to the *executeFT* method of the *FTStrategy* class. This method is responsible for executing the particular strategy and for activating the application specific methods defined in the application thread, as for example, *variant1* (primary block) and *acceptanceTest*. Some of these methods correspond to original

implementations, but others, like *variant2* (recovery block) and saveCheckpoint should be defined to allow the execution of the DRB strategy.

In this example, *ExampleThread* is stateless; otherwise *FTExampleThread* should also implement the methods *getState* and *setState*. These methods are needed to provide state initialization between the primary and the shadow nodes in DRB. None of these methods are necessary in the original version, as only one *ExampleThread* instance runs in a single node.

```
class FTExampleThread : public FTThread {

  DRBStrategy myDRB;
  Msg* recMsg;
  Msg outMsg;
  IncommingMessageAdministrator<Msg, 20> inMessages;
public:

  FTExampleThread(){
    ... // init code
    myDRB.setMaxResponseTime(20000);
    setFTStrategy(&myDRB);
  }

  void run () {
    while(1) {
      recMsg = inMessages.receive();
      ftStrategy->executeFT();
    }
  }

  void variant1(){
    ... // same code of original process method
  }

  void sendResult(){
    ... // same code of original output method
  }
  // to be defined
  void variant2(){ ... }
  void saveCheckpoint(){ ... }
  void restoreCheckpoint(){ ... }
  bool acceptanceTest(){ ... }
};
```

Figure 5.4: Example of FT application thread.

5.3.3 Application-specific entities

Each FT strategy instantiation and usage demands the definition of strategy attributes and application specific behavior. These requirements are summarized in

Table 5.1, Table 5.2 and Table 5.3. Table 5.1 presents requirements for multiple version software, Table 5.2 for single version software and Table 5.3 for voters.

Single version strategies use the same *FTStrategy* classes used for multiple version software, but do not implement their full functionality. If some application thread does not implement a given method, a default implementation is inherited. For example, the default implementation for *save/restoreCheckpoint* is empty and for *acceptanceTest* is to return true (success).

Table 5.1: Multiple version strategies requirements.

Definition Requirements		RB	DRB	NVP
Entity	**Type**			
FT Strategy	object	RBStrategy	DRBStrategy	NVPStrategy
Response time	parameter	Yes	Yes	Yes
variant 1	method	Yes	Yes	Yes
variant 2	method	Yes	Yes	Yes
variant 3	method	-	-	Yes
saveCheckpoint	method	Yes	Yes	-
restoreCheckpoint	method	Yes	Yes	-
acceptanceTest	method	Yes	Yes	-
sendResult	method	Yes	Yes	Yes
onFailure	method	Optional	Optional	Optional
Voter Thread	object	-	-	Yes
getState	method	-	state threads only	state threads only
setState	method	-	state threads only	state threads only

Table 5.2: Single version strategies requirements.

Definition Requirements		Restart	Checkpoint and Restart	PSP	TMR
Entity	**Type**				
FT Strategy	object	RBStrategy	RBStrategy	DRBStrategy	NVPStrategy
Response time	parameter	Yes	Yes	Yes	Yes
variant 1	method	Yes	Yes	Yes	Yes
variant 2	method	-	-	-	-
variant 3	method	-	-	-	-
saveCheckpoint	method	-	Yes	Yes	-
restoreCheckpoint	method	-	Yes	Yes	-
acceptanceTest	method	-	Yes	Yes	-
sendResult	method	Yes	Yes	Yes	Yes
onFailure	method	Optional	Optional	Optional	Optional
Voter Thread	object	-	-	-	Yes
getState	method	-	-	state threads only	state threads only
setState	method	-	-	state threads only	state threads only

Table 5.3: Voter requirements.

Definition Requirements		Application Specific Voter	Standard Voter (StdVoter)
Entity	**Type**		
Thread name	parameter	Yes	Yes
Coordination method	parameter	Yes	Yes
Response Time	parameter	Yes	Yes
Input subject	parameter	-	Yes
Output subject	parameter	-	Yes
storeSolution	method	Yes	-
findEqualSolution	method	Yes	-
sendResult	method	Yes	-

The simplest single version FT strategy is the Restart strategy. In this technique only one variant is defined, and the acceptance test is not implemented. Therefore, the only possible error detection mechanism is deadline expiration, which is set by the *Response time* parameter. The Checkpoint and Restart strategy can be implemented as a single version simplification of the RB strategy. In this case, only one real variant is defined, and the body of *variant2* should contain a call to the *variant1* method. In a similar way, PSP is implemented with the DRB strategy and TMR with the NVP strategy.

The *onFailure* method in Table 5.1 and Table 5.2 is always optional. It can be used to define application-dependent fault handling mechanisms when a failure in the strategy execution occurs. After running the code defined in the *onFailure* method, the thread will be restarted by the operating system.

Table 5.3 displays the requirements for voting threads. These threads are only needed when using TMR or NVP. In the general case, a voter is application-specific and this thread must implement the *VoterThread* methods shown in Table 5.3. The *Coordination method* parameter defines if all replica voters will execute the *sendResult* method or if only a master voter will do it. The definition of the master voter in a coordinated voting is performed by the FT framework. The *Response time* of a voter is the maximum time allowed for a voting cycle. A cycle begins when the voter receives the first solution. Voters try to find a match between two solutions (2 out of 3), but if only one solution is received and the voting cycle period has finished, that solution is considered correct and it is sent as a result.

In Table 5.3, the column labed “Standard Voter” lists the requirements for the initialization of *StdVoter* objects. This class provides exact voting using messages for receiving solutions and sending the results. Using this standard voter, other parameters must be defined, as the subject of input and output messages.

5.4 Framework general implementation

This section describes the general FT framework implementation, which consists of patterns and mechanisms used in all FT strategy development.

5.4.1 Timing behavior

The *MiddlewareScheduler* (MS) thread runs at the beginning of every clock tick interval (e.g. 1ms; see Section 4.2.1) and controls the behavior and execution of each FT thread and voter. Besides, this thread is also responsible for activating the middleware thread that delivers external incoming messages.

Figure 5.5 shows an example of the execution of a Recovery Blocks (RB) strategy. The MS thread runs periodically and releases the message reception thread each two activation periods. The message reception thread is not executed in every cycle in order to reduce CPU utilization and to provide at least one cycle in two for FT threads free execution. In the first cycle, the *FTThread* receives a message and starts the FT execution. This example shows a failure in the primary block and a success in the recovery block.

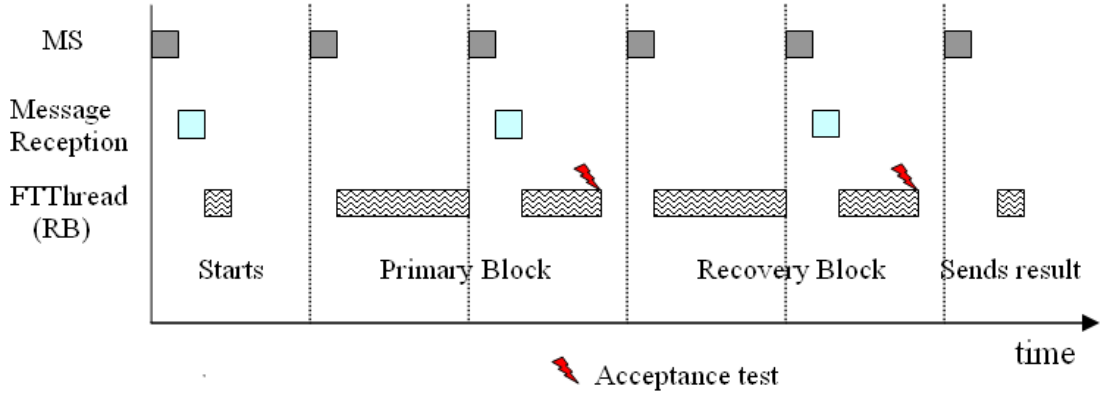

Figure 5.5: RB execution timing example.

Figure 5.6 contains an activity diagram that shows the interaction between the *FTThread* and the *MiddlewareScheduler* thread in the execution of the RB strategy. After being activated, an FT thread sets up a deadline for execution, based on the actual time and the maximum allowed response time, the thread suspends. In subsequent MS activations, this thread restarts the *FTThread* if the deadline has expired. This situation represents a failure in delivering the correct response on time, but after restarting the *FTThread* is ready to receive the next request. If the deadline has not expired, the MS thread commands the next actions to be performed by the *FTThread* thread and schedules it for execution. After executing the right operations (save/restore state, run primary/recovery block, run acceptance test) the RB thread suspends again and the MS thread checks the acceptance test (AT) result. If the *FTThread* succeeds in the AT, the MS thread allows it to send its results and the interaction finishes. If the *FTThread* fails in both blocks, it is restarted by the MS thread.

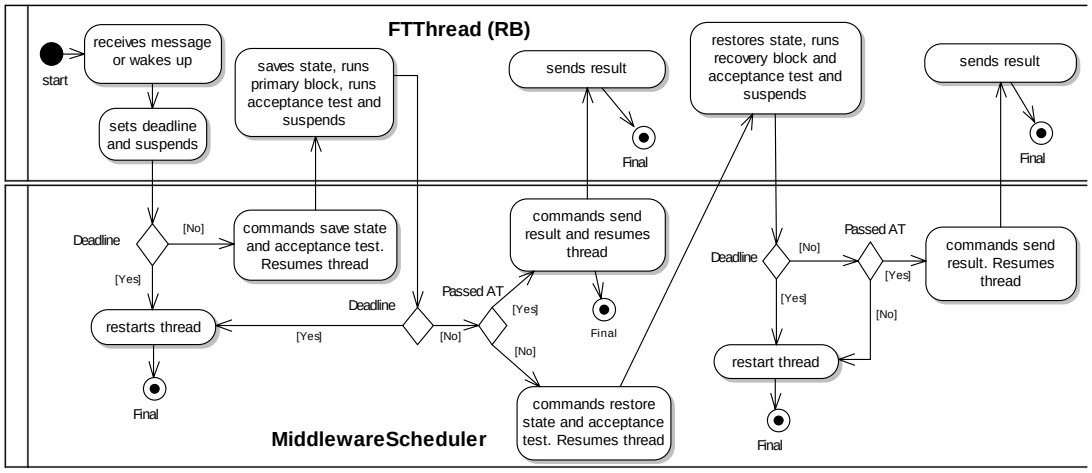

Figure 5.6: RB execution activity diagram.

5.4.2 Class structure

Any FT strategy is executed in the context of two separate threads: the *FTthread* and the *MiddlewareScheduler* thread. The *FTThread* executes methods in response to the control algorithm performed by the *MiddlewareScheduler* thread. However, all the code related to a given FT strategy is defined by its *FTStrategy* concrete class. Figure 5.7 shows a class diagram describing the main methods involved in the execution of an FT strategy.

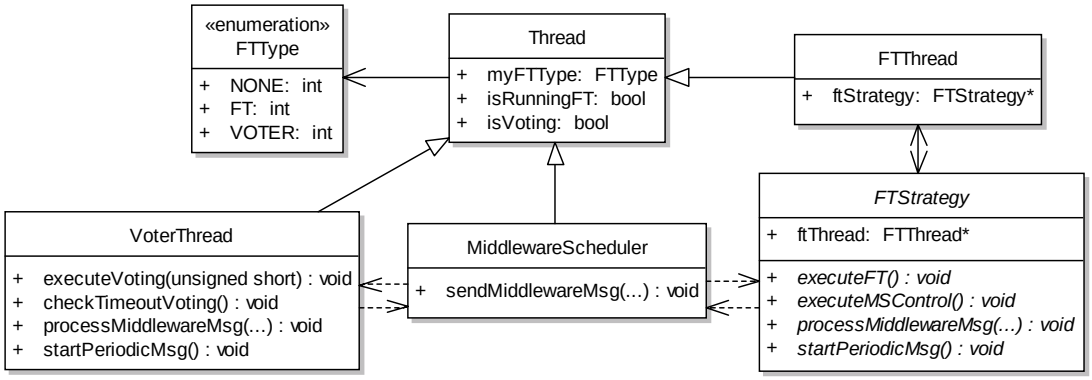

Figure 5.7: FT strategy execution class diagram.

Every *FTStrategy* subclass must implement the *executeFT* method, which performs the FT control algorithm that runs in the context of the FT thread (upper part of Figure 5.6, excluding message reception). It must also implement the

executeMSControl method, which performs the MS control algorithm for that stategy (botton part of Figure 5.6). Using this approach, the *MiddlewareScheduler* class does not depend on any FT strategy implementation, and FT strategies can be added to the framework transparently.

The MS thread controls the execution of the voter threads in a similar way. However, the MS control is simpler, as it only have to detect if the voting deadline has elapsed. The *executeVoting* method is executed by the *VoterThread*, while the *checkTimeoutVoting* method is called by the MS thread.

In contrast with the RB strategy presented so far, other FT strategies involve the utilization of multiples instances of the FT thread, running in different nodes. These FT threads have to communicate in order to coordinate, establish roles and initialize states. In this framework, the required communication between FT threads is executed by message passing between the *MiddlewareScheduler* threads of each node. If an FT thread needs to send a message, it calls the *sendMiddlewareMessage* method of MS. The sending message is broadcasted to all other nodes and their MS threads will distribute it to the related FT threads in their nodes, if any, by calling the *processMiddlewareMessage* method of the corresponding FT strategy. The same applies to *VoterThreads* that can communicate using the same methods described above.

Another feature performed by *MiddlewareScheduler* is the activation of the *startPeriodicMsg* of FTStrategy and *VoterThreads* periodically (e.g. 300ms), in order to trigger the execution of periodic tasks as, for instance, role conflicts detection in the DRB strategy.

Finally, the *MiddlewareScheduler* thread is responsible for changing the FT threads priorities according to the Earliest Deadline First (EDF) scheduling. Therefore, in each MS activation the FT thread with earliest deadline is found and its priority is raised to a maximum among application threads. This feature can be enabled or disabled in the framework.

Figure 5.8 contains a sequence diagram that describes the activities performed by *MiddlewareScheduler* each time it runs. First it reads messages coming from other *MiddlewareScheduler* objects in other nodes. These messages are sent by external

FTThreads and *VoterThreads* and must be delivered to internal objects of the same type and name, if any. Therefore, *MiddlewareScheduler* checks if there is any local thread with the name received in the incoming message. If there is, it determines if it is an FT thread or voter, based on the *myFTType* attribute of the Thread class, and calls the *processMiddlewareMsg* method of the related class (*FTStrategy* or *VoterThread*). Next, if a predefined number of *MiddlewareScheduler* activations have been executed; periodic messages of FT threads and voters are triggered, using the *startPeriodicMsg* method. Finally, the control algorithm of active FT threads is performed by running the *executeMSControl* method. Similarly, MS checks the timeout for active voting threads by calling the *checkTimeoutVoting* method. Active FT threads and voters are represented by a true value in the *isRunningFT* and *isVoting* Boolean attributes of the Thread class.

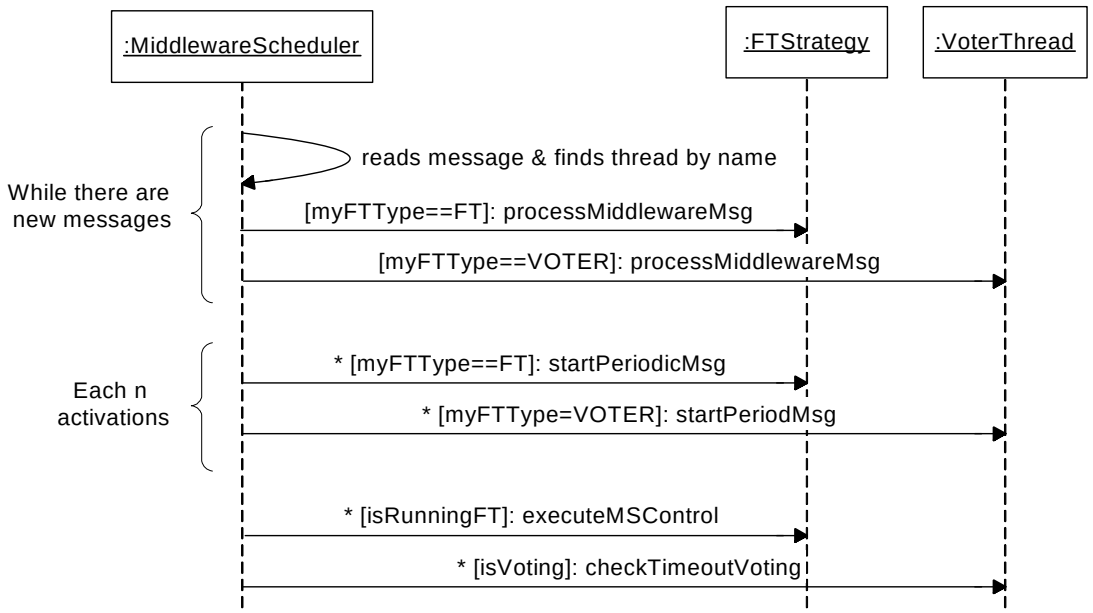

Figure 5.8: *MiddewareScheduler* thread sequence diagram.

Figure 5.8 does not represent FT thread scheduling, but this operation is performed, if selected, at the end of each MS activation.

5.5 FT strategies implementation

This section describes the implementation of the fault tolerance strategies which were provided by the FT framework: RB, DRB and NVP. These strategies are described in Section 2.8. The implementation of an FT strategy consists basically in developing an *FTStrategy* class that contains the algorithm performed by the FT technique. Each strategy may also define algorithms for role definitions and state coordination. The implementation of these operations is supported by the *MiddlewareScheduler* thread, as presented in the previous section.

5.5.1 Recovery Blocks strategy

The Recovery Blocks (RB) strategy, described in Section 2.8.2, consists of the sequential execution of software variants, or alternates, using an acceptance test as adjudicator. The implementation of RB in this framework is limited to two software variants because it is the minimum configuration that is able to tolerate one active fault. The utilization of more software variants would require additional development efforts and it would increase the memory consumption.

Figure 5.9 shows a class diagram that only presents classes, attributes and methods directly related to the RB strategy operation. The Recovery Blocks technique is implemented by the *RBStrategy* class. This class derives from *FTStrategy* and implements the *executeFT* method, which defines the FT thread behavior, and the *executeMSControl* method, which defines the *MiddlewareScheduler* thread behavior. Other virtual functions defined in *FTStrategy* are not implemented, as this strategy runs in a single node and does not send messages to other replicas. The *waitingForMS* attribute is used to indicate to the MS thread that the FT thread is waiting for further commands. The *passedAT* attribute contains the result of the last acceptance test and it is used by the MS to define the next commands. These commands are issued through the following class attributes: *tryBlock*, which defines the next variant to run and *mustSendResult*, which defines if a result can be sent. An example of the RB strategy execution has been shown in Figure 5.5 and the coordination between the FT thread and MS has been shown in Figure 5.6.

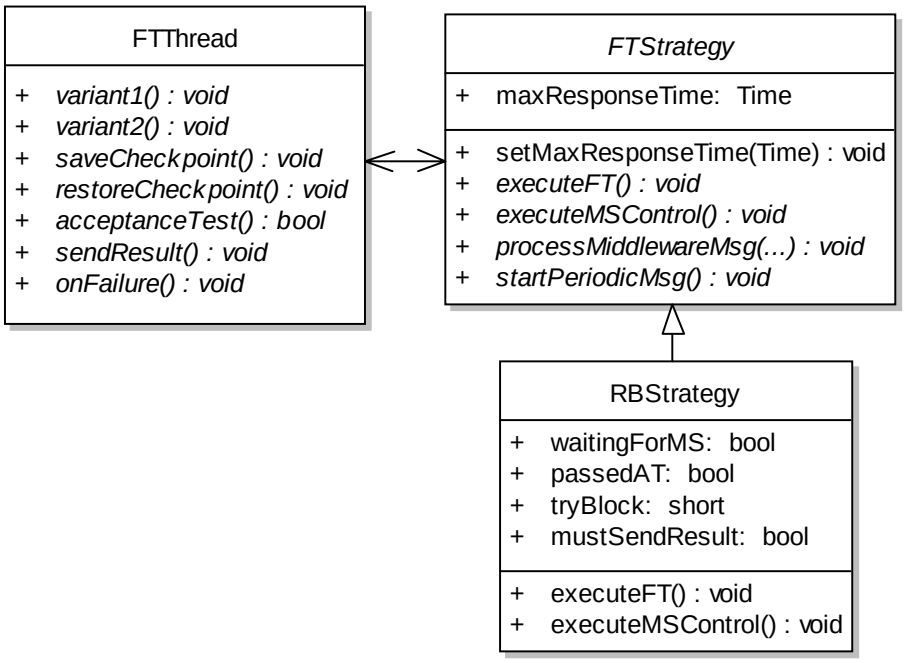

Figure 5.9: RB strategy class diagram.

The execution of the RB strategy is presented in Figure 5.10. Two software versions or alternates are applied, defined by the *variant1* and *variant2* methods. The entry point is the execution of the *executeFT* method of *RBStrategy*, which sets the deadline based on the *maxResponseTime* attribute and suspends. The MS resumes the FT thread, which then executes the checkpointing (*saveCheckpoint* method), the primary block (*variant1* method), and the acceptance test (*acceptanceTest* method); after that, the FT Thread suspends again. In the next activation, the MS checks the AT result and, if it succeeded, commands the execution of the *sendResult* method and finishes the strategy operation. Otherwise, it will command the checkpoint restoration (*restoreCheckpoint* method), the execution of an alternate block (*variant2* method), and the reexecution of the acceptance test. If both variant executions fail in the acceptance test, or if deadline expiration is detected by the MS thread, no results will be sent and the *onFailure* method will be called (see description in Section 5.3.3). After returning from the *onFailure* method, the thread will be restarted.

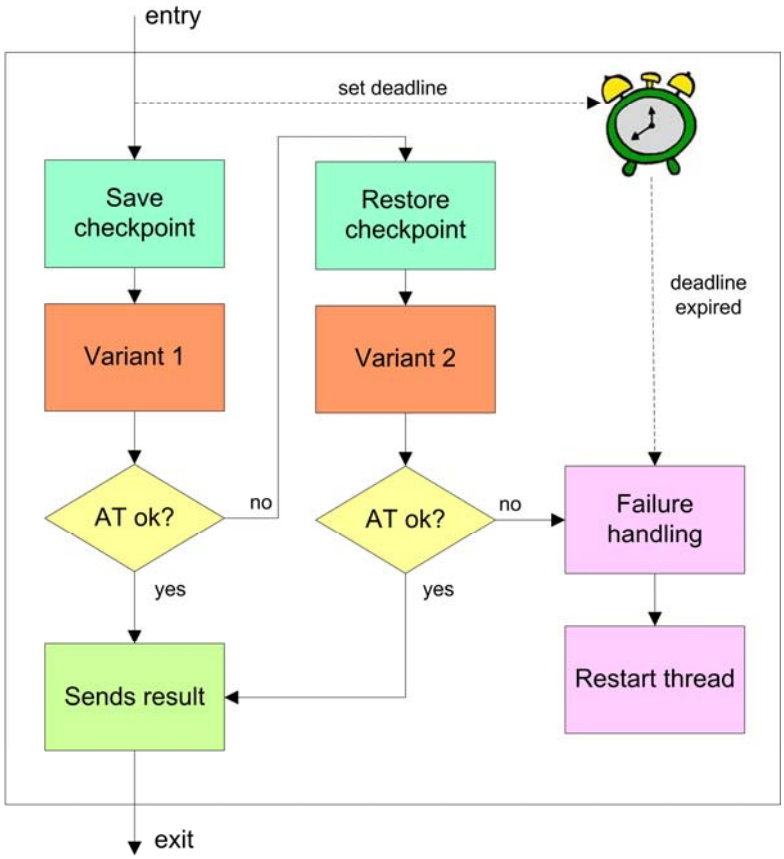

Figure 5.10: RB strategy execution.

The maximum response time parameter (*maxResponseTime*) must include the extra time needed to execute the second variant if the first variant fails. The minimum value for this parameter is equal to three times the clock tick interval, as shown in Figure 5.11, in which the clock tick interval is represented by 20 time units. In the first period the FT thread only sets the deadline, in the second period it executes the first variant, and in the third period it executes the second variant. When the FT thread is sending the results the deadline verification is already disabled.

The checkpointing mechanism is application-dependent and it must save all static variables, global variables and class attributes that can be modified by the first variant, in order to be restored to their original values before running the second variant. This might include state data and input data that is overwritten during the computation process (see Figure 5.1). Non-static local variables and variables initialized by the software variants do not need to be saved. For stateless threads with unmodified input data no checkpoint is required, and the application thread may use the default empty

implementation defined in the *FTThread* class. The acceptance test, implemented by the *acceptanceTest* method, is also application-dependent and should return true in case of success and false in case of failure. The default implementation of this method returns true.

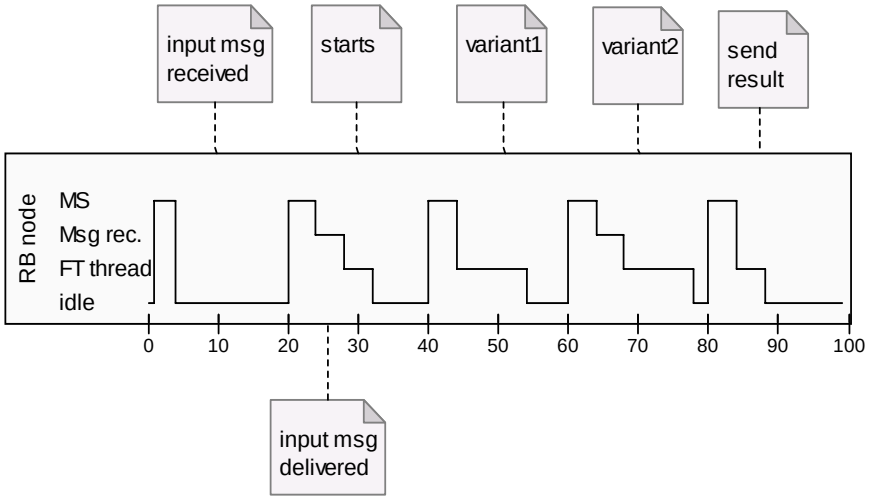

Figure 5.11: RB timing example.

The *RBStrategy* class can also be applied to implement single version software techniques, as described in Table 5.2. The Checkpoint and Restart strategy only differs to RB because the second variant is equal to the first. Therefore, the implementation of the *variant2* method should consist of a call to the *variant1* method. As discussed in Section 2.8.1, the Checkpoint and Restart strategy has limited software fault tolerance capability.

The Restart strategy is the simplest configuration. In that strategy, the default *acceptanceTest* method it used, and therefore, the only error detection mechanism is deadline expiration, which causes a thread restart. Despite recovering the faulty thread and allowing it to respond to further activations, this strategy can not avoid failures.

The RB strategy described here is based in sequential execution in a single node. Therefore, state consistency mechanisms are not provided. However, it is possible to implement fault-tolerant configurations applying multiple replicas of RB threads, if these threads are stateless, as shown in Figure 5.12.

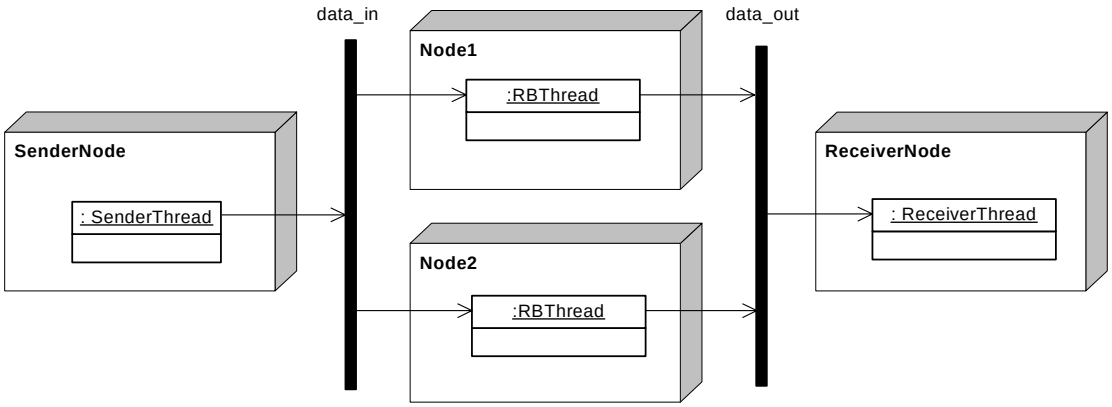

Figure 5.12: Stateless RB threads configuration example.

In this configuration, both *RBThreads* receive messages with the string "data_in" as subject from *SenderThread*, and send their results in a message with the string "data_out" as subject. The *ReceiverThread* subscribes to "data_out" messages but sets the option for discarding duplicate messages (see Section 4.4.1). Therefore, only the first message from *RBThreads* will be delivered. The RB strategy provides software fault tolerance while its redundant execution provides hardware fault tolerance. If the RB threads were not stateless, the output messages of the two *RBThreads* would diverge in case of failure in one of them.

5.5.2 Distributed Recovery Blocks strategy

The Distributed Recovery Blocks (DRB) strategy, described in Section 2.8.3, coordinates the execution of two RB-like threads in distinct nodes, using a primary/shadow configuration, in which only the primary thread sends its results. Although not defined by the DRB strategy [64], the implementation of DRB in this framework provides a mechanism for maintaining the state consistency between replicas, in order to support state threads.

Figure 5.13 shows a class diagram that only presents classes, attributes and methods directly related to the DRB strategy operation. The DRB technique is implemented by the *DRBStrategy* class, which derives from *FTStrategy*. The class structure differences from *RBStrategy* to *DRBStrategy* are:

- *DRBStrategy* implements the *processMiddlewareMsg* method to handle messages received from the other replica.
- *DRBStrategy* implements the *startPeriodicMsg* method to trigger the transmission of messages used to detect role conflicts between replicas.
- *DRBStrategy* contains an enumeration object called *myDRBRole* to define the thread role: primary or shadow.
- *DRBStrategy* uses the *isPrimaryDone* and *isShadowDone* attributes to keep track whether the primary and shadow threads have succeeded in the acceptance test.
- *DRBStrategy* uses the *isFirstActivation* and *hasFinishedInitialization* attributes to support the implementation of state initialization.

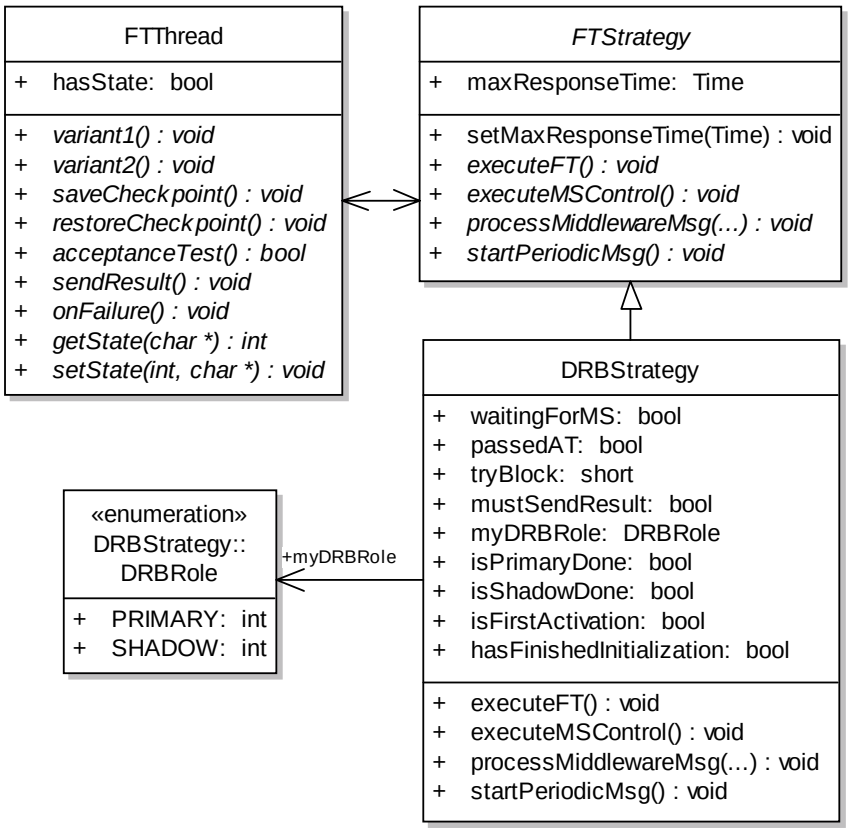

Figure 5.13: DRB strategy class diagram.

The execution of the *DRBStrategy* is presented in Figure 5.14. The primary thread runs *variant1* as primary block and *variant2* as an alternate. The shadow thread runs these variants in the reverse order.

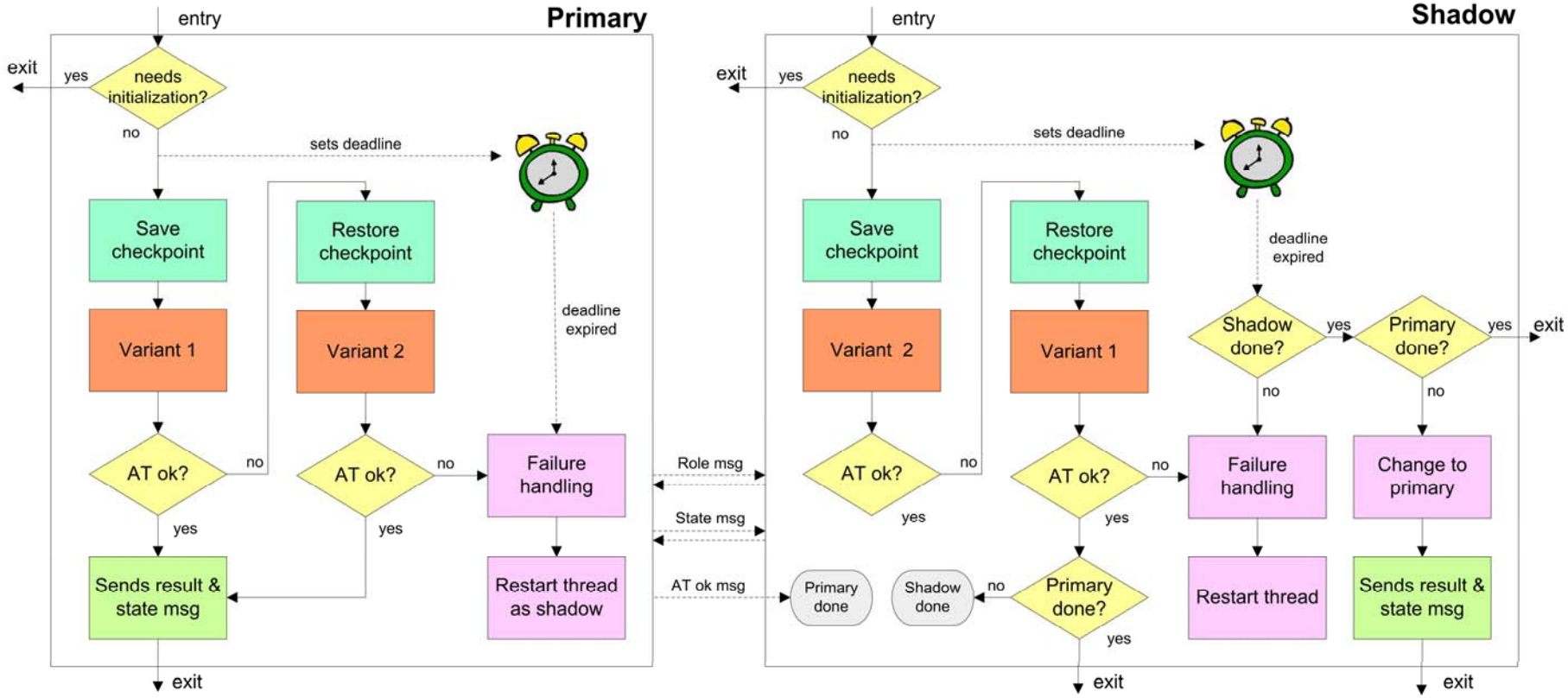

Figure 5.14: DRB strategy execution.

At the entry point, the thread checks the need for state initialization based on the *hasState* attribute of *FTThread* and the *isFirstActivation* attribute of *DRBStrategy*, which is reset at startup. If state initialization is needed, the thread aborts operation and will only execute in the next activation. In the meanwhile, a state message from a previously initialized DRB thread should be received. If this message is not received, the thread assumes that no other node has been running previously, and therefore the default initialization values are taken. In both cases, the deadline is set and the thread begins its normal RB-like execution.

There are different control algorithms for primary and shadow DRB threads. A primary thread executes as if it were alone, similarly to the RB strategy execution shown in Figure 5.10. However, if it misses the deadline it will be restarted as a shadow thread. Additionally, a primary thread sends a message indicating success in the acceptance test, and also a state message just after sending its results.

The shadow thread behaves differently, as it must execute the second variant, perform the acceptance test, and wait for the acceptance test message from the primary thread. In this implementation, the timeout for waiting this message is equal to the execution deadline. Therefore, when this deadline expires, the MS thread in the shadow node verifies the *isPrimaryDone* and *isShadowDone* variables to decide one of the possible outcomes:

- exit the shadow execution silently, if both threads have succeeded;
- restart the shadow thread, if it has failed; or
- change the role of the shadow thread to primary and allow it to send its results, if only the former primary has failed.

As shown in Figure 5.14, three types of messages are exchanged between DRB replicas:

- **AT success message**: this message is generated by the primary thread to inform the shadow thread about the success in executing an acceptance test.
- **State message**: this message contains state data needed to initialize a state thread. The *getState* method of *FTThread* must be implemented for state threads. This method serializes the state data that is assembled in the state message. The

returning value of *getState* is the state data size. In the receiving side, the *setState* method deserializes this data to the proper variables. A stateless FT thread should not implement the *getState* and *setState* methods. In that case, the default implementation of *getState* will return zero as state data size, which will set the *hasState* attribute of *FTThread* to false. This is accomplished just at initialization time.

- **Role message:** this message is sent periodically by both the primary and shadow nodes to allow the detection and correction of role conflicts (primary/shadow definitions). The receiving thread checks if the other replica role is equal to its own, and if so, executes a role conflict resolution algorithm based on the node identification number (see Section 4.4.2). The priority used in role conflicts is inversely proportional to the node identification number. A DRB thread always starts executing as shadow and therefore two DRB threads starting at the same time will have the same role. That situation will be corrected as soon as the higher priority thread receives the first role message from the lower priority thread, and changes its role to primary. The period of the role message is defined by the MiddlewareScheduler, as described in Section 5.4.2. Another role conflict situation occurs when both the primary and shadow threads fail, and consequently are restarted as shadow.

Figure 5.15 shows a DRB strategy configuration example similar to the one presented in Figure 5.12 for the RB strategy. This configuration uses middleware messages with subject "FTStatus" to send role, state and AT success messages as explained above. Note that only the primary *DRBThread* sends its results to *ReceiverThread*.

The minimum value for maximum response time (*maxResponseTime* parameter) in the DRB strategy depends on many factors, such as the message transmission time from the primary to the shadow node. Figure 5.16 shows a timing diagram that presents the worst situation, in which the primary node starts executing after the shadow node and fails in the execution of the first variant. In this figure, one clock tick interval is represented by ten units of time.

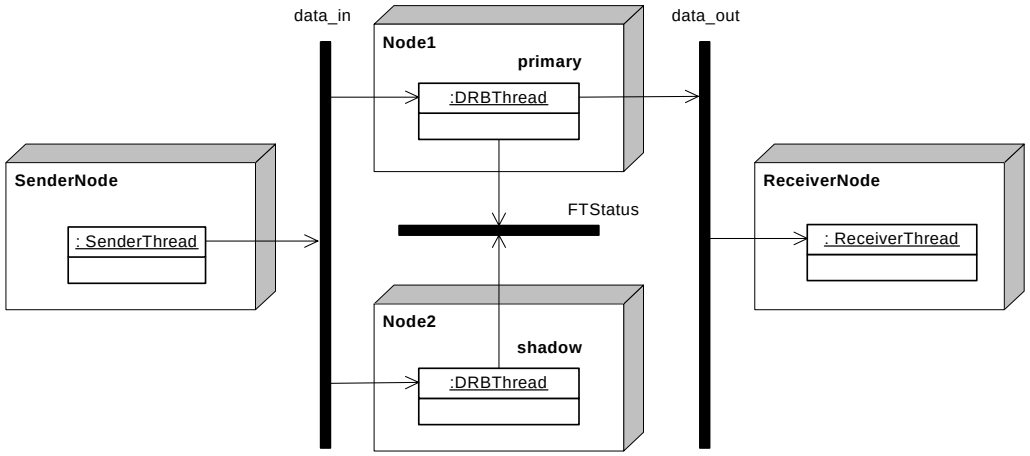

Figure 5.15: DRB strategy configuration example.

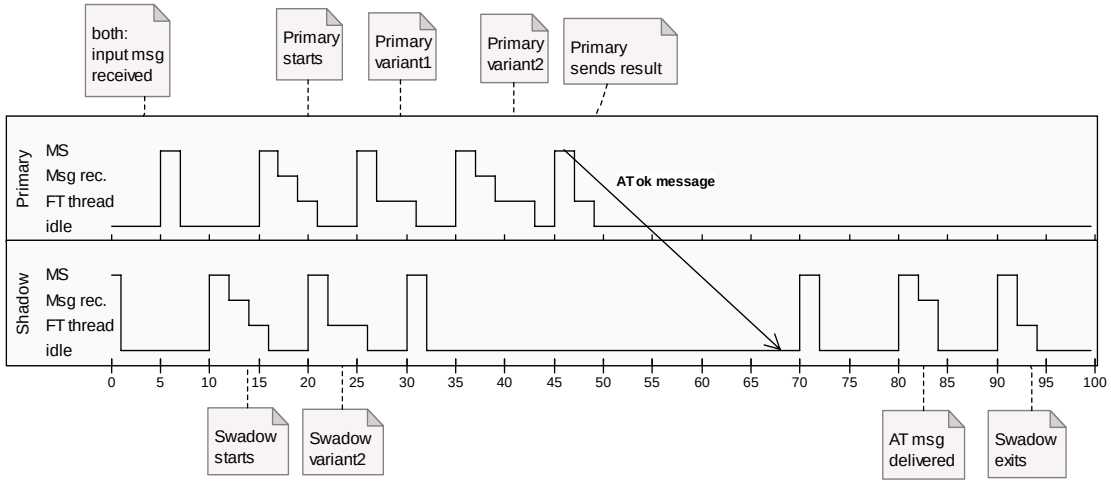

Figure 5.16: DRB timing diagram.

If we consider a small computing time for each variant execution and the worst possible timing between nodes, as they are not synchronized, the following time delays should be considered:

- Difference between DRB threads start times. The starting time of each thread depends on the input message delivery to the FT threads, which happens once in two MS activations. In the worst scenario, the time difference between FT thread activations is equal to two clock tick intervals (**2** Δ_{ck}).
- DRB primary thread execution time. If the computation time of the variants is small (e.g. half of a clock tick interval), the total execution time, excluding the time spent in sending the results, is equal to three clock tick intervals (**3** Δ_{ck}).

- Message transmission time. This is the delay in transmitting the AT success message between the primary and the shadow node. This time is called Δ_M.
- Message delivery time. This is the time delay between receiving a message in the shadow node and delivering it to the MS thread mailbox. In worst case scenario this delay is equal to two clock tick intervals ($\mathbf{2}\ \Delta_{ck}$).
- Shadow exit delay. This is the time spent from the message delivery to the MS mail box to the moment when the FT thread runs again after this AT message is read by the MS thread. This delay is about one clock tick interval (Δ_{ck}).

The sum of all the delays described above is equal to $\mathbf{8}\ \Delta_{ck} + \Delta_M$. This represents the minimum value of *maxResponseTime* for DRB execution. For example, if the clock tick interval is 2 ms, and the network transmission delay is 10 ms, then *maxResponseTime* should be at least 26 ms.

The *DRBStrategy* class can be applied to implement the PSP single version software technique, as described in Table 5.2. In that case, the *variant2* implementation should call the *variant1* method. As explained in Section 2.8.3, this configuration has limited software fault tolerance capability, but is effective against hardware permanent and transient faults.

5.5.3 N-Version Programming strategy

The N-Version Programming (NVP) strategy, described in Section 2.8.4, consists of the concurrent execution of software variants followed by a decision mechanism, usually implemented by majority voting. The implementation of NVP in this framework is limited to three software variants because it is the minimum configuration needed to mask one active fault. The utilization of more software versions would require additional hardware resources that are usually not available for small-scale embedded systems.

Figure 5.13 shows a class diagram that only presents classes, attributes and methods directly related to the NVP strategy operation. The NVP technique is implemented by the *NVPStrategy* class, which derives from *FTStrategy*. This class implements the *executeFT* method, which defines the FT thread behavior and the

executeMSControl method, which defines the *MiddlewareScheduler* thread behavior. Additionally, this class also implements the *processMiddlewareMsg* method for processing state messages received from NVP threads in other nodes.

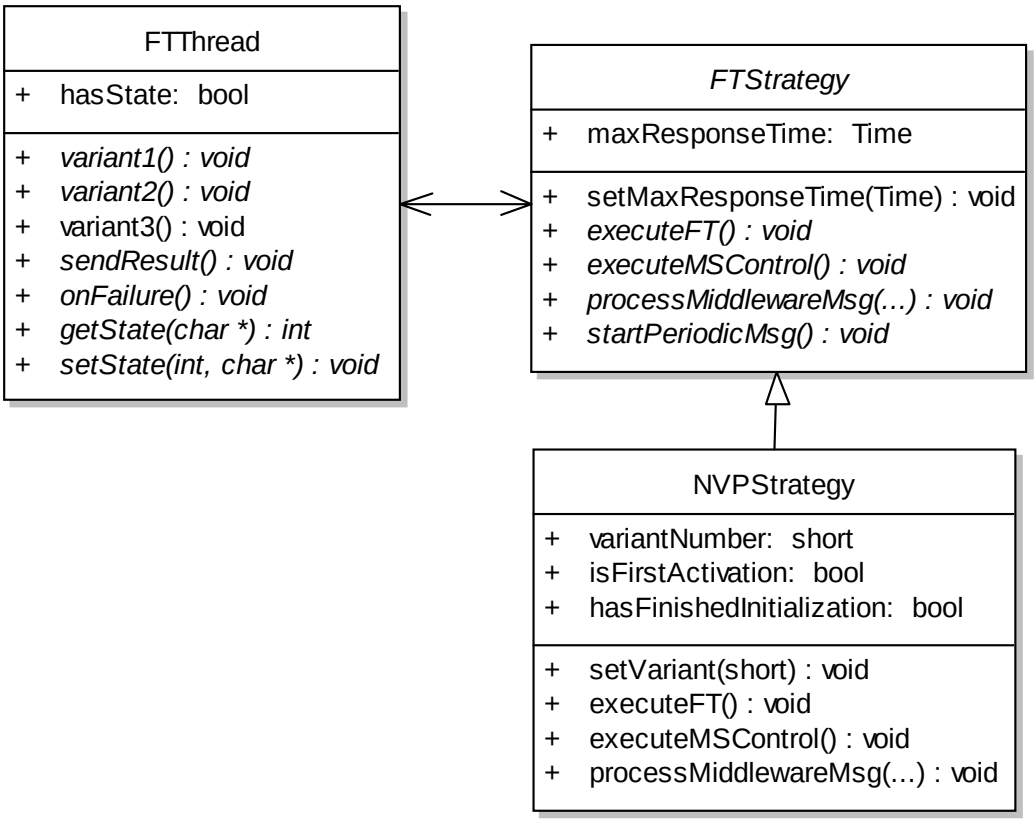

Figure 5.17: NVP strategy class diagram.

The *NVPStrategy* class has an attribute to define the variant that should be executed (*variantNumber*). This attribute is set at initialization time by the *setVariant* method, and it is not supposed to change at runtime. The advantage of having three software variants in the same class, instead of defining three different application threads, is to smooth the process of deployment. Using this design solution, the same software can be loaded in all embedded systems, and the definition about which variant a node will execute can be taken at runtime. A possible implementation is to define the variant to execute based on some node identification (e.g. IP number). Similarly to *DRBStrategy*, *NVPStrategy* uses the *isFirstActivation* and *hasFinishedInitialization* attributes to support the implementation of state initialization.

If only one software version is available, the *NVPStrategy* can be used to implement a TMR strategy. In that case, only the *variant1* method must be implemented and the *variantNumber* attribute must be set to one.

The execution of the NVP strategy is presented in Figure 5.18. At the entry point, the thread checks the need for state initialization based on the *hasState* attribute of *FTThread* and the *isFirstActivation* attribute of NVPStrategy, which is reset at startup. If state initialization is needed, the thread aborts operation and will only execute in the next activation. In the meanwhile, a state message from a previously initialized NVP thread should be received. Then, the deadline is set and the thread selects one variant for execution based on the *variantNumber* attribute. At the end of the variant execution, a result message is sent to one or more voter threads. Besides, a state message is sent if the *hasState* attribute is set.

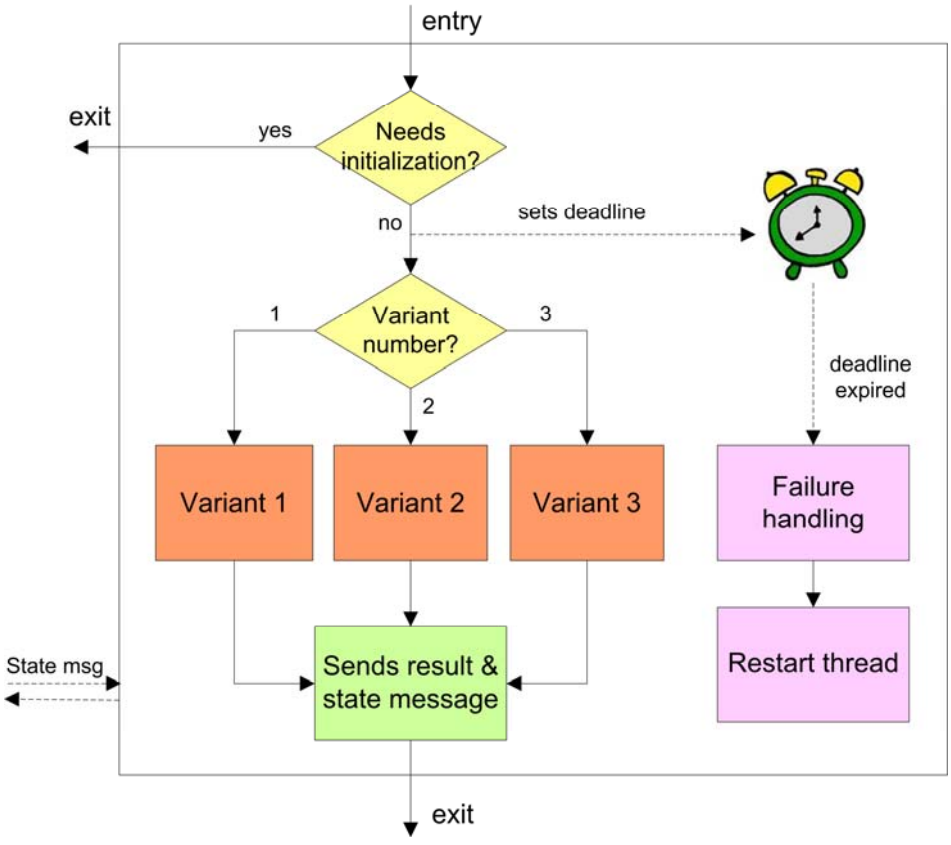

Figure 5.18: NVP strategy execution.

In this strategy, the *MiddlewareScheduler* thread only verifies if the deadline has expired. If the deadline expires, the *onFailure* method is called and the thread is restarted.

The state initialization mechanism is exemplified in Figure 5.19. In this situation, the NVP thread #1 was already running when NVP threads #2 and #3 started. In the first activation after starting, the joining NVP threads skip any processing and wait for a state message to initialize state data. In the next activation they start their normal execution. If the joining threads do not receive any state message they still start running in the next activation, but in that case they use the default state data. That situation happens if all threads start at the same time. This means that state threads loose one activation event to perform state initialization.

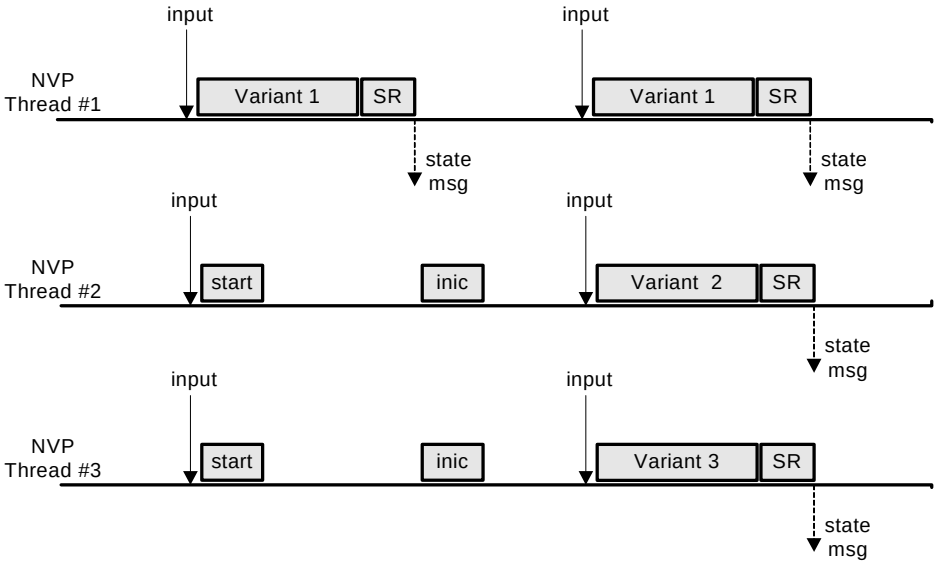

Figure 5.19: NVP state initialization example.

The *NVPStrategy* class only takes care of the computation process. The NVP technique also depends on one or more voter threads that receive result messages from the FT threads and select a result based on majority voting. Figure 5.20 presents three possible voting configurations. NVP threads subscribe to the "input_data" subject and send their results using "unvoted-data" as subject. One or more voter threads receive the result messages and select one result which is sent using "voted_data" as subject. The communication between NVP threads (state messages) and voter threads (role messages) are not shown in these figures. In Figure 5.20a, only one voter is used and therefore a failure in the node where the voter is running will lead to a system failure. Figure 5.20b contains a configuration that uses one voter for each NVP thread, usually running in the same node. That configuration tolerates permanent failures in one or two nodes. The configuration in Figure 5.20c is similar to the one in Figure 5.20c, but

only the master voter sends the selected result. That mechanism is termed **coordinated voting**, while the configuration in Figure 5.20b is termed **free voting**.

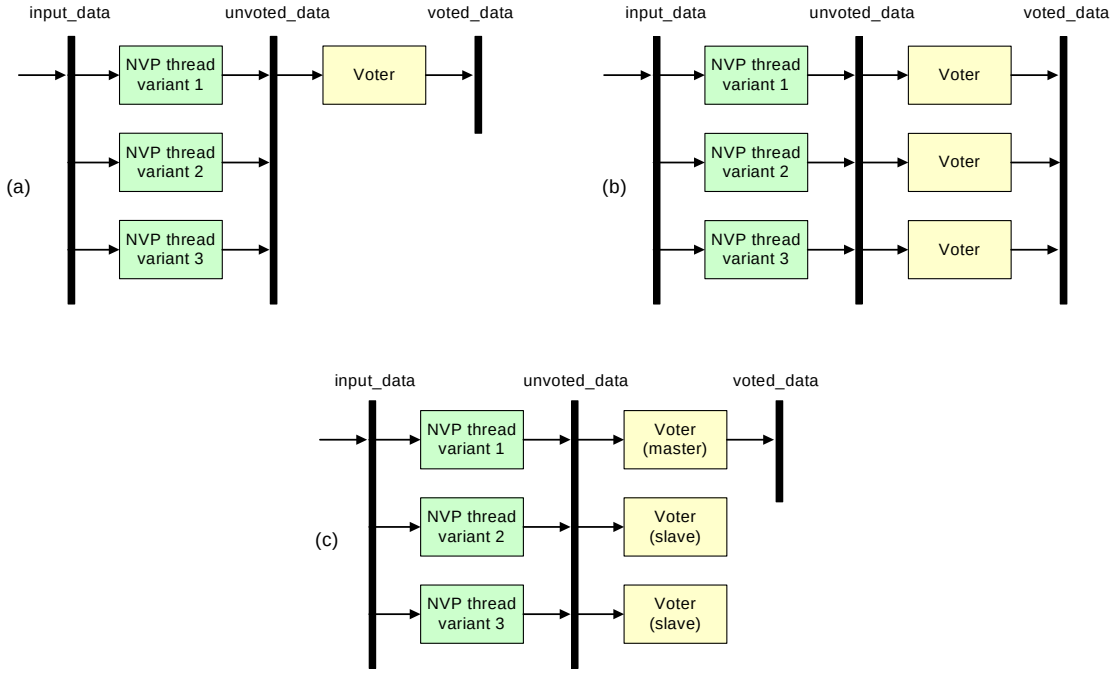

Figure 5.20: Voting configurations.

The implementation of voters depends on the middleware support for message identification, as described in Section 4.4.1, in order to define when a new voting problem or cycle begins. Therefore, NVP threads must send a common identification number in their result messages for each voting cycle. One possible solution is to include this identification in the input message. However, the activation of NVP threads and the output of voter threads can be performed without messages. NVP threads can be activated by a timing mechanism but they must agree in the generation of the message identification number sent to voter threads. This can be accomplished by the state data coordination mechanism provided by the NVP strategy. Similarly, voter threads can send or use the selected result by other means, as for instance, sending data to hardware devices. If the outputs of the voter threads are sent to another BOSS thread, there is no need for coordinated voting, as the middleware is able to discard duplicate messages. If otherwise only one message must be sent to the successor task or if only one voter must drive a hardware device, then coordinated voting should be used.

Voter threads are implemented by means of the *VoterThread* class, shown in Figure 5.21. *VoterThread* derives from thread and defines the following virtual functions that must be implemented by application voters:

- ***storeSolution***: saves result data (also called "solution") received from a NVP thread.
- ***fingEqualSolution***: compares the last received data from a NVP thread with previous received and stored data, trying to find a match ("equal" solution) using an application-specific procedure.
- ***sendResult***: outputs the selected result.

The *nextSolIndex* attribute is initialized with zero at the beginning of a voting cycle and should be used by application voters as an index to store and compare result data.

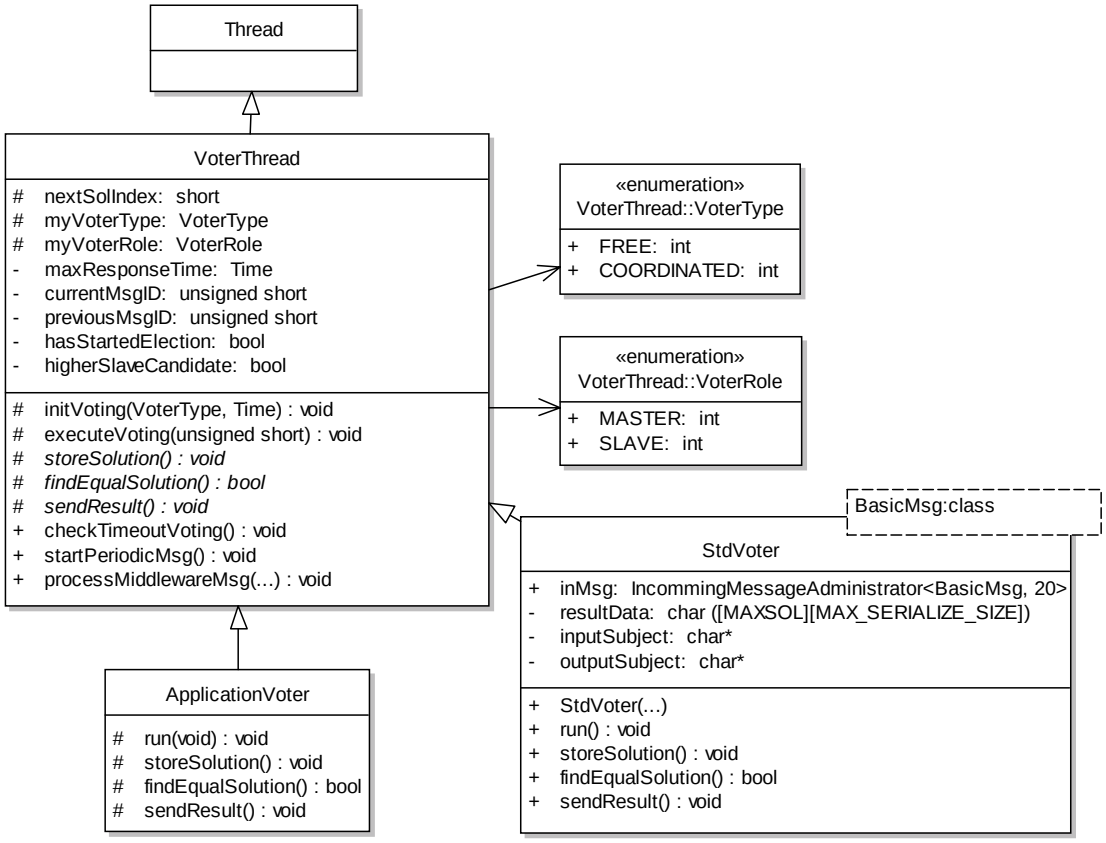

Figure 5.21: *VoterThread* class diagram.

VoterThreads can run free or coordinated, as defined by the *myVoterType* attribute. In coordinated voting, the *myVoterRole* keeps the role definition: master or slave. The *maximumResponseTime* attribute is used to define the deadline for each voting cycle. Both the coordination method and the response time are initialized by the *initVoting* method. The *executeVoting* method must be called after each voter activation (e.g. after receiving a message) and it must receive an identification number as a parameter. The *currentMsgID* and *previousMsgID* attributes keep track of the recent identification numbers and are used to detect a new voting cycle and also late arriving messages from previous cycles. The *hasStartedElection* and *higherSlaveCandidate* attributes are used to perform the master election in coordinated voting. The deadline of a voting cycle is verified periodically by the MiddlewareScheduler thread, using the *checkTimeoutVoting* method. Other methods called by the MS thread are *startPeriodicMsg*, which triggers a role message transmission in case of coordinated voting and *processMiddlewareMsg*, which processes incoming role messages.

The *StdVoter* class defines a standard application voter that performs inputs and outputs using messages and executes exact voting, as described in Section 5.3.3. This class embeds a mail box for receiving messages of a type defined by a template parameter (*BasicMsg*). The memory used to store incoming results from NVP threads is defined by the *resultData* attribute. The subjects of the input and output messages is kept by the *inputSubject* and *ouputSubject* attributes. The initialization of these attributes, as well as others, as coordination method and response time is performed by the class constructor.

A voting execution diagram is presented in Figure 5.22. The entry point is the execution of the *executeVoting* method of VoterThread. The detection of a new voting cycle is performed by comparing the received identification number of the current and previous cycle's identification numbers. If a new voting cycle is detected, the *storeSolution* method is called and a deadline is set. If otherwise, the received identification is compared to the previous cycle's identification to detect a late arriving message, which will cause the discarding of the result data. The result data is also discarded if a result has been previously selected by the voter. If none of these discarding situations occur, the *storeSolution* and *findEqualSolution* methods are

called. If the *findEqualSolution* method return true, indicating that a match between different stored result data ("equal" solution) was found, and the voter is allowed to send outputs (free voter or master), the *sendResult* method is called.

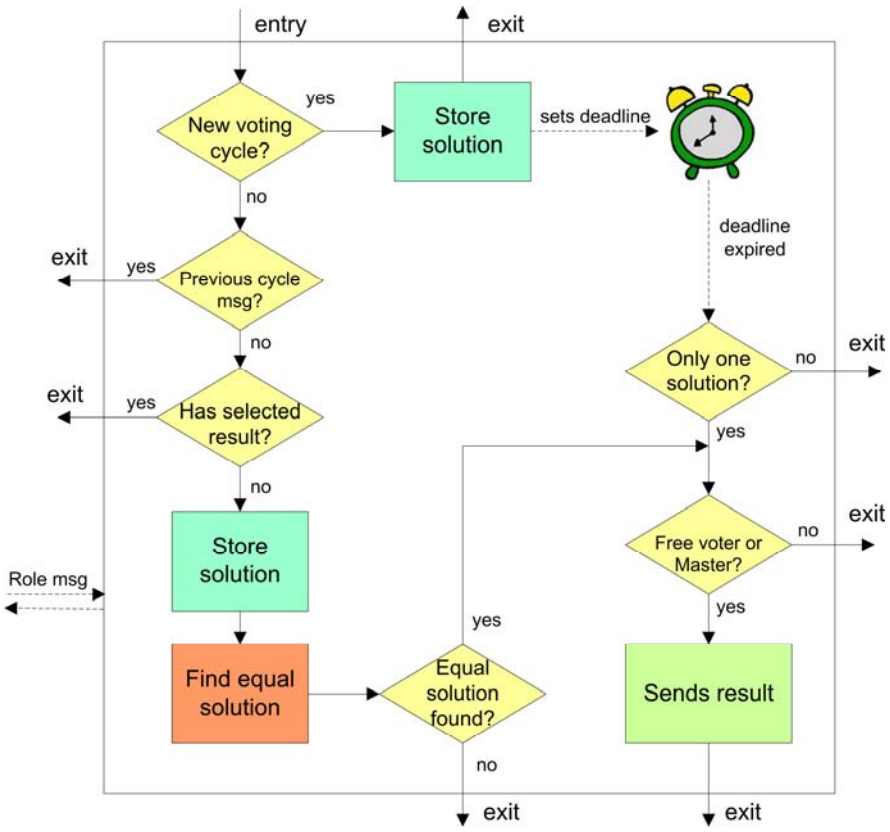

Figure 5.22: Voting execution diagram.

While a voting cycle is still running (no result has been selected), the MS thread keeps checking the *VoterThread* deadline. When this deadline expires, two situations could have happened:

- Only one result data have been received – in that case this solution is considered correct, and the voter selects it for output.
- Two or three solutions have been received but they haven't matched – in that case no output is produced by the voter.

The master election algorithm used in coordinated voting is represented by the state diagram in Figure 5.23. At the beginning of an election process in a voter, the initial state of the algorithm will be Master or Master Candidate, depending on the present role of the voter, master or slave, respectively.

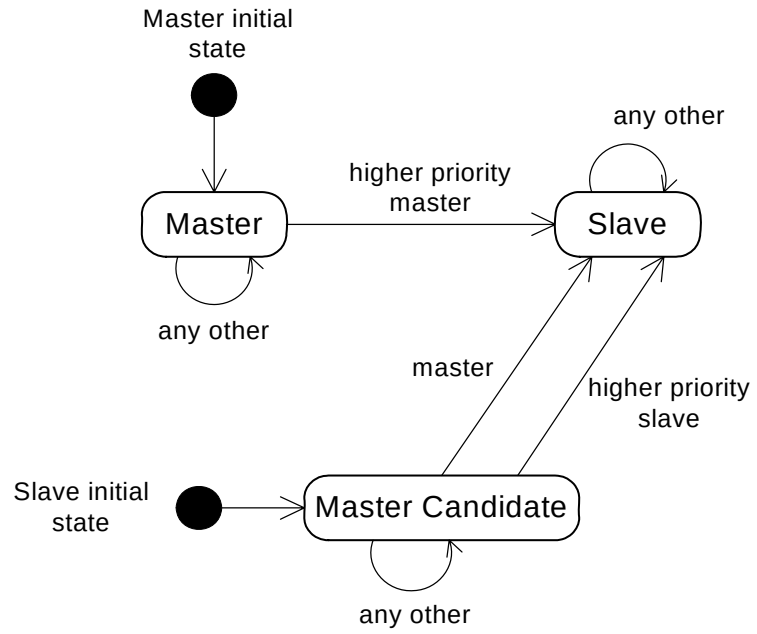

Figure 5.23: Master election state diagram.

Transitions in this state diagram are triggered by the reception of role messages from other nodes that satisfy the conditions shown in the figure. For instance, if a slave voter starts the election as a Master Candidate and further receives a message from a master voter or from a higher priority slave voter, then it changes its state to Slave. A voter priority is inversely proportional to its node identification number.

Role messages are sent periodically, triggered by the MS thread, as described in Section 5.4.2. The duration of an election process is twice as big as the role message period. At the end of the election, the final state of a voter determines its role. A role change is performed if one of the following situations occurs:

- A master voter ends the election in the Slave state; or
- A slave voter ends the election in the Master Candidate state.

When a voter thread starts it assumes a slave role. Therefore, a lower priority master voter will keep its role regardless of the arriving of new higher priority voters. This design decision aims to minimize role changes between coordinated voters.

The master election algorithm is executed continuously, meaning that a new election period starts immediately after the previous election finishes. If a master voter fails, the worst possible scenario occurs when it happens just after sending its

role message and when this message arrives to the highest priority slave at the beginning of an election period, as shown in Figure 5.24.

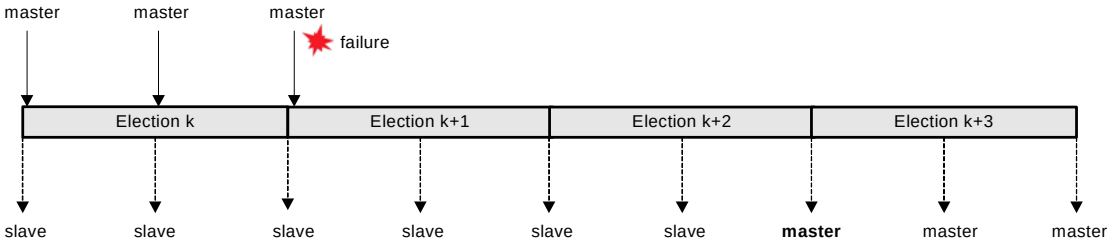

Figure 5.24: Master voter failure worst scenario.

Figure 5.24 presents a sequence of elections carried out by the higher priority slave voter. The last master voter role message is received at the beginning of the k+1 election. Therefore, this voter only detects the master failure and assumes as master at the end of the k+2 election. The worst case recovery time from a master failure is equal to two election periods. For example, in a system with clock tick interval of 2 milliseconds and running periodic messages each 100 *MiddlewareScheduler* activations, the election period would be equal to 400 milliseconds, and so the worst case recovery time would be equal to 800 milliseconds.

5.6 Discussion

The utilization of the proposed FT framework for the development of embedded fault-tolerant systems has several benefits:

- It simplifies the application level programming, as programmers don't have to implement fault tolerance mechanisms, but just have to provide application-specific parameters and procedures. The same happens regarding other distributed mechanisms, such as state initialization and output coordination.
- The application program follows a standard structure in which changing the FT strategy becomes easy and straightforward. This reduces efforts in strategy selection, configuration and testing.

- It facilitates the creation and integration of new fault tolerance strategies. The proposed framework is easily extendable by adding new *FTStrategy* and *VoterThread* derived classes, as described in Section 5.4.
- Provides a means of implementing adaptive fault tolerance [52], as changing the FT strategy can be performed at runtime by simply calling the *setStrategy* method. The strategy can be modified based on the reliability requirements of each mission phase, or even for other factors as resource availability and power consumption.

The drawback of the proposed FT framework is the increase in the OS memory footprint and runtime overhead. The FT framework is fully integrated into the operating system code. Therefore, even for non-fault tolerant applications some extra resources will be used, as it will be presented in Chapter 7. A possible solution to this problem is providing two versions of the operating system: with and without the FT framework. However, this solution demands the utilization of more than one version of some operating system classes, which makes software development and maintenance more difficult. In Chapter 6 it will be presented a solution to this problem using AOP.

5.7 Summary

A framework to support application-level fault tolerance was designed and implemented. This framework is easily customizable and extensible, providing mechanisms for hardware and software fault tolerance. The design goals were simplicity and efficiency, in order to run in small-scale real-time embedded systems.

The units of fault tolerance in this framework are the BOSS threads. This approach provides better mechanisms for system recovery, such as thread restarting. The thread model for fault-tolerant threads supports both state threads and stateless threads, and it is commonly used in the design of fault-tolerant systems.

The application of fault tolerance in an existing system is straightforward. An FT object must be created and registered. Additionally, some parameters and methods must be provided by the application program. The framework is responsible for

executing the selected FT strategy and for exchanging messages needed in the implemention of these strategies, such as the ones related to role definitions and state consistency.

Three main fault tolerance strategies were implemented: RB, DRB and NVP, but other single version strategies may be derived from them. Detailed descriptions of each strategy class structure and execution algorithm were presented. Furthermore, worst case scenarios in terms of execution times for each FT strategy have been exhibited. The development and integration of new FT strategies into the framework is simple and do not imply modifications in other framework classes.

The utilization of the proposed FT framework featured several advantages over ad-hoc implementations, simplifying the application-level programming and improving the system configurability and extensibility.

Chapter 6

Applying AOP for fault tolerance

This chapter describes the application of AOP to support the implementation of fault tolerance. In this work, AOP was applied for three different purposes: (1) modularize the fault tolerance code at the application level; (2) integrate the FT framework into the operating system; and (3) implement fault tolerance at the operating system level.

6.1 Application-level fault tolerance

The FT framework described in the last chapter can be used to convey fault tolerance to an existing application. In order to build a fault-tolerant application, the source code of critical threads must be modified. For instance, the source code shown in Figure 6.1 presents a non fault-tolerant thread, while Figure 6.2 shows the source code of the same thread after fault tolerance introduction. The main differences between these source codes are highlighted. This modification can lead to the introduction of coding errors and also can make maintenance more difficult, as two software versions now exist: the original and the fault-tolerant. These versions should remain compatible throughout their evolution, aiming to allow configurability and reuse. As a mean to improve fault tolerance integration and maintenance, AOP techniques were applied to modularize all fault-tolerant code, keeping the original source code intact. In this work, AOP was mainly used to automatically generate the source code of fault tolerant threads (e.g. Figure 6.2) by weaving the original non fault-tolerant thread source code (e.g. Figure 6.1) with FT aspects.

```
class ExampleThread : public Thread {
  Msg* recMsg;
  Msg outMsg;
  IncommingMessageAdministrator<Msg,20> inMessages;
public:
  ExampleThread(){ ... // init code}

  void run () {
    while(1) {
      recMsg = inMessages.receive();
      process();
      output();
    }
  }

  void process(){
    ... // uses msg data and state data
  }

  void output(){
    ... // prepares output message
    outMsg.send("exampleResult");
  }
};
```

Figure 6.1: Example of thread source code before fault tolerance introduction.

```
class FTExampleThread : public FTThread {

  DRBStrategy myDRB;
  Msg* recMsg;
  Msg outMsg;
  IncommingMessageAdministrator<Msg, 20> inMessages;
public:

  FTExampleThread(){
    ... // init code
    myDRB.setMaxResponseTime(20000);
    setFTStrategy(&myDRB);
  }

  void run () {
    while(1) {
      recMsg = inMessages.receive();
      ftStrategy->executeFT();
    }
  }

  void variant1(){
    ... // same code of original process method
  }

  void sendResult(){
    ... // same code of original output method
  }
  // to be defined
  void variant2(){ ... }
  void saveCheckpoint(){ ... }
  void restoreCheckpoint(){ ... }
  bool acceptanceTest(){ ... }
};
```

Figure 6.2: Example of thread source code after fault tolerance introduction.

6.1.1 Code generation

The process of generating the executable code using this approach is shown in Figure 6.3. Inputs and outputs of weavers, compilers and linkers are represented by continuous lines, while application source code dependencies are represented by dashed lines. The operating system, already integrated to the fault tolerance framework, is compiled and an OS library is generated. Abstract Strategy Aspects are developed for each FT strategy in the system. They define virtual pointcuts and standard advices used for all related Concrete Strategy Aspects. A concrete aspect must be defined for advising each future fault-tolerant application thread, as it will be

later discussed. The weaving process using AspectC++ generates a fault-tolerant application that is eventually compiled and linked to the OS code.

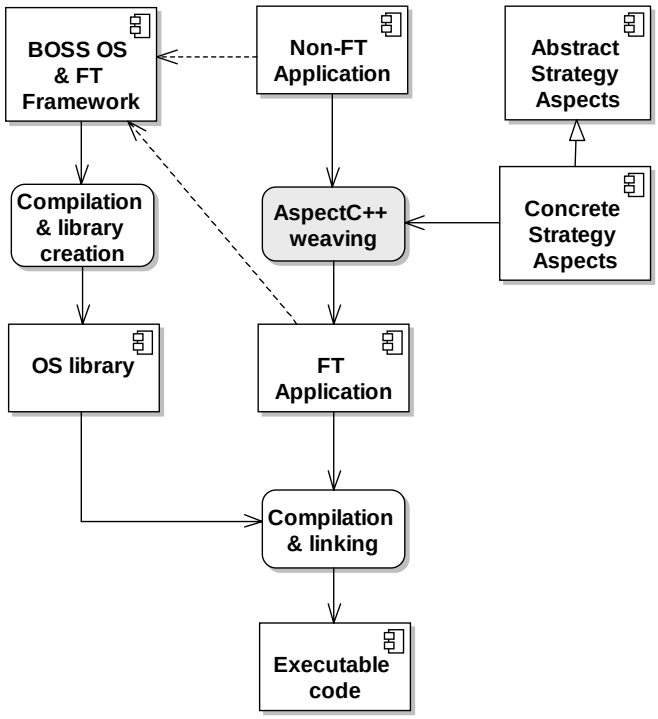

Figure 6.3: Code generation process using AOP at the application level.

Using this process, all fault-tolerant code is defined inside the aspect code, and the non-FT application remains unchanged. The fault tolerance concern is consequently separated from the main functionality.

6.1.2 AspectC++ restriction

AspectC++ has a restriction related to the introduction of base classes that had an impact on this work. In AspectC++, base classes can be included but they can never replace an existing base class (as AspectJ does). The introduction of a base class in AspectC++ can lead to multiple inheritance if the target class of the introduction has already one base class. In Figure 6.4 we can see that the application of a base class introduction in the original code of Figure 6.1 does not result in the FT code of Figure 6.2. Instead, it adds *FTThread* as a base class of *ExampleThread* in a multiple inheritance mechanism.

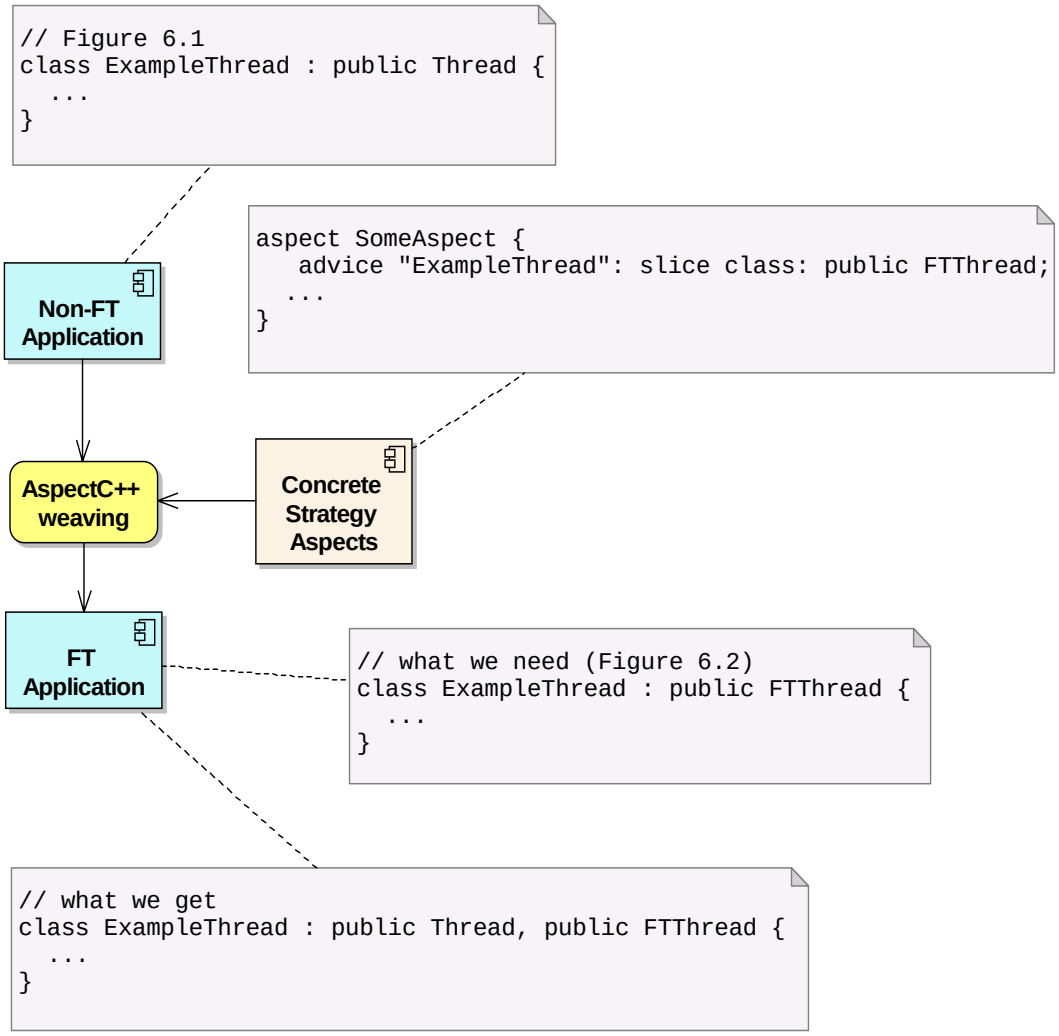

Figure 6.4: AspectC++ base class introduction example.

In AspectJ, the introduction of a base class by an aspect is always performed by substitution, as Java does not allow multiple inheritance. As C++ allows both single and multiple inheritance, AspectC++ should provide support for two kinds of base class introduction: by substitution and by addition. The suggestion to include the base class substitution functionality in AspectC++ has been sent to the AspectC++ mailing list [14, 15] in January and April of 2007. We hope that new versions of AspectC++ can support that feature.

Some workarounds can be applied to deal with this AspectC++ restriction, but all involve modifications in the FT framework and cause performance or memory footprint penalties. The selected solution was to eliminate the *FTThread* class from the FT framework and include all its attributes and methods into the *Thread* class, as

shown in Figure 6.5 (compare with Figure 5.3). This solution avoids the usage of the base class introduction shown in Figure 6.4, and consequently does not incur in the performance overheads related to multiple or virtual inheritance. However, it increases the memory footprint of the final application, as non-FT threads have their memory size enlarged.

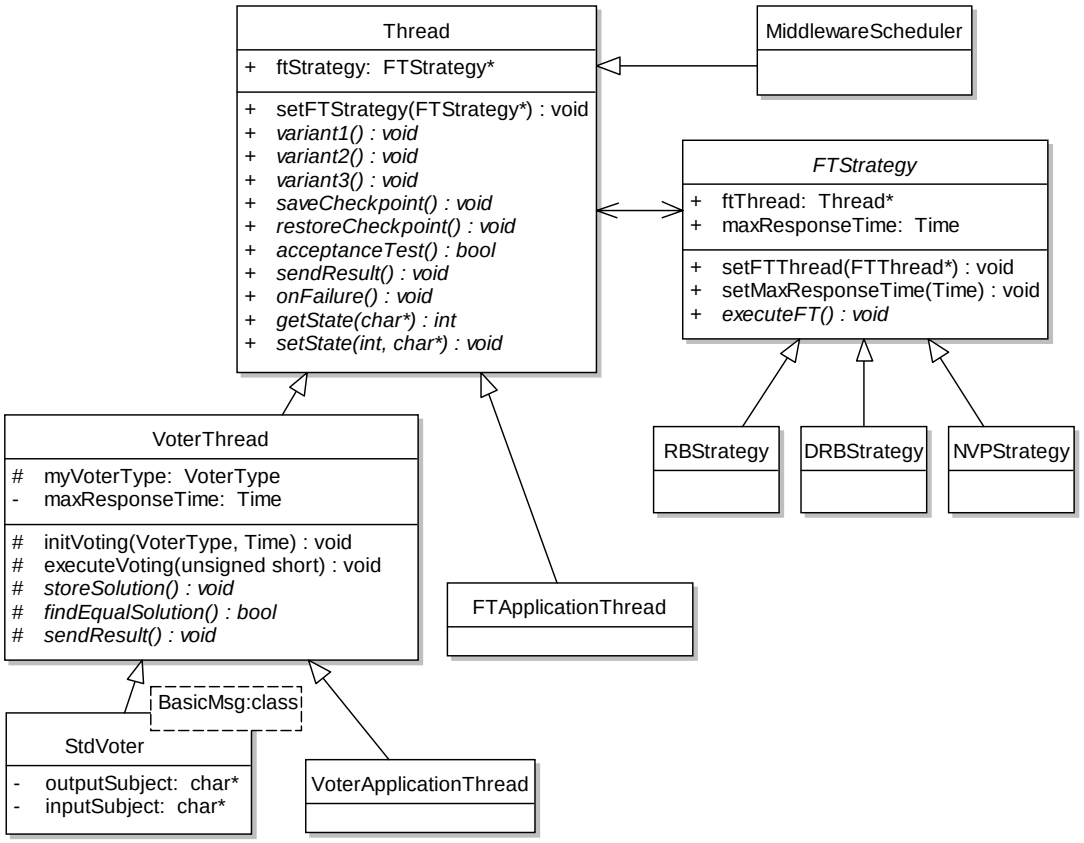

Figure 6.5: FT framework modified for AOP application.

6.1.3 AOP implementation

This section describes the implementation of abstract and concrete aspects that introduce fault tolerance to threads. As an example, the *ExampleThread* class shown in Figure 6.1 will be made fault-tolerant, using the DRB and NVP strategies.

Figure 6.6 shows the abstract aspect related to the DRB strategy. This aspect is general and can be applied by all threads using this strategy and other single version strategies related to it. Similar abstract aspects were developed for the RB and NVP strategies. Initially this aspect declares three virtual pointcuts (lines 2-4) that will be

defined by concrete aspects. These pointcuts represent the thread class under modification (*DRBClass*) and the original methods for processing (*ProcessMethod*) and output (*OutputMethod*). The integer *maxResponseTime* (line 5) keeps the maximum response time for execution, which must be defined by concrete aspects. The introduction of the *DRBStrategy* object definition is carried out using the AspectC++ slice construction (line 7), which is used to extend the static structure of a program. The initialization of this object, as well as its registration, is performed by the advice in line 13, which uses the *constr* pointcut (line 11), similarly as done in the constructor code of the non-AOP version in Figure 6.2.

```
01  aspect DRBStrategyAbstract {
02    pointcut virtual DRBClass() = 0;
03    pointcut virtual ProcessMethod() = 0;
04    pointcut virtual OutputMethod() = 0;
05    int maxResponseTime;
06
07    advice DRBClass(): slice class {
08      private:
09        DRBStrategy myDRB;
10    };
11    pointcut constr() = construction(DRBClass());
12
13    advice constr(): after(){
14      tjp->target()->myDRB.setMaxResponseTime( maxResponseTime );
15      tjp->target()->setFTStrategy(&(tjp->target()->myDRB));
16    }
17
18   pointcut compute()= call(ProcessMethod()) &&
19       target( DRBClass() ) && !within( "% ...::variant%(...)" );
20
21    advice compute(): around(){
22      tjp->target()->ftStrategy->executeFT();
23    }
24    pointcut result()= call(OutputMethod()) &&
25     target( DRBClass() ) && !within( "% ...::sendResult(...)" );
26
27    advice result(): around(){
28    }
29  };
```

Figure 6.6: DRB strategy abstract aspect.

The *compute* pointcut (line 18) defines a condition in which the processing method of the non-FT thread is called in the original code. The around advice related to this pointcut (line 21) will replace this call by the activation of the *executeFT* method of the *FTStrategy* class. Similarly, the *result* pointcut (line 24) defines a condition in which the output method of the non-FT thread is called in the original

code. The around advice related to this pointcut (line 28) will just suppress this call, as the activation of the thread output is going to be controlled by the *FTStrategy* object.

The concrete aspect to make *ExampleThread* fault-tolerant is shown in Figure 6.7. The aspect inherits from the *DRBStrategyAbstract* aspect and initially defines its virtual pointcuts (lines 3-5). In this case, the target thread is "ExampleThread", the processing method is "process" and the output method is "output", as seen in Figure 6.1.

```
01  aspect DRBExampleConcrete: public DRBStrategyAbstract {
02
03    pointcut DRBClass() = "ExampleThread";
04    pointcut ProcessMethod()= "% ...::process()";
05    pointcut OutputMethod() = "% ...::output()";
06
07    DRBExampleConcrete(){
08      maxResponseTime = 20000;
09    }
10
11    advice DRBClass() : slice class {
12      public:
13        void variant1(){ process(); }
14        void sendResult(){output(); }
15
16        // methods to be defined
17        void variant2(){ ... }
18        void saveCheckpoint(){ ... }
19        void restoreCheckpoint(){...}
20        bool acceptanceTest(){...}
21    }
22  };
```

Figure 6.7: DRB strategy concrete aspect example.

The maximum response time for this strategy is set to 20.000 microseconds in the aspect constructor (line 8), by initializing a base abstract variable. After that, several methods are introduced in the target thread. The virtual method *variant1* (line 13) is responsible for running the primary block in DRB, and in this case it must execute the original processing of *ExampleThread*. Similarly, the virtual method *sendResult* (line 14) must call the original output method. Here it can be noticed that the calls to process and output in the introduced methods *variant1* and *sendResult* will not trigger the execution of the advices defined by the *compute* and *result* pointcuts in the *DRBStrategyAbstract* aspect of Figure 6.6 because the *within* scope pointcut function is being applied. Finally, the application-specific methods are defined for this strategy (lines 17-20), such as *variant2* (recovery block) and *saveCheckpoint*. After the

weaving process, the new *ExampleThread* code becomes functionally equivalent as the non-AOP version of Figure 6.2.

Figure 6.8 presents the abstract aspect to implement the NVP strategy using a *StdVoter* class. The differences to the abstract aspect for the DRB strategy of Figure 6.6 are highlighted.

```
01  aspect NVPstdStrategyAbstract {
02    pointcut virtual NVPClass() = 0;
03    pointcut virtual ProcessMethod() = 0;
04    pointcut virtual OutputMethod() = 0;
05    int maxResponseTime;
06    int variant;
07    char inputSubject[20];
08
09    advice NVPClass(): slice class {
10      private:
11        NVPStrategy myNVP;
12    };
13    pointcut constr() = construction(NVPClass());
14
15    advice constr(): after(){
16      tjp->target()->myNVP.setMaxResponseTime( maxResponseTime );
17      tjp->target()->myNVP.setVariant(variant);
18      tjp->target()->setFTStrategy(&(tjp->target()->myNVP));
19    }
20
21   pointcut compute()= call(ProcessMethod()) &&
22       target( NVPClass() ) && !within( "% ...::variant%(...)" );
23
24    advice compute(): around(){
25      tjp->target()->ftStrategy->executeFT();
26    }
27    pointcut result()= call(OutputMethod()) &&
28     target( NVPClass() ) && !within( "% ...::sendResult(...)" );
29
30    advice result(): around(){}
31
32    pointcut sendMessage()= call("%Message::send(...)") &&
33      that( NVPClass() ) && within ( OutputMethod() );
34
35    advice sendMessage(): before(){
36      *tjp->arg<0>() = inputSubject;
37    }
38  };
```

Figure 6.8: NVP strategy with StdVoter abstract aspect.

The *variant* attribute (line 6) keeps the variant number executed by this node, and it is defined by the concrete aspect. The variant number is set in *NVPStrategy* object in line 17, inside the advice of the target thread constructor. The most remarking difference to the DRB strategy is the need to advise the call to the send method of the

Message class inside the output method of the target thread, as defined by the *sendMessage* pointcut (line 32). This happens because the fault-tolerant thread now has to send its results to a voter thread instead of the final destinations, and so the subject of the output message has to change. In the advice in line 35, the input argument to the *Message::send* method is changed to the subject of input messages to the voter (line 7), which is defined by the concrete aspect.

An example of NVP concrete aspect (using *StdVoter*) applied to the same *ExampleThread* of Figure 6.1 is shown in Figure 6.9. The main differences to the DRB concrete aspect of Figure 6.7 are highlighted.

```
01  aspect NVPExampleConcrete: public
02                                NVPstdStrategyAbstract {
03
04    pointcut NVPClass() = "ExampleThread";
05    pointcut ProcessMethod()= "% ...::process()";
06    pointcut OutputMethod() = "% ...::output()";
07    StdVoter<Msg> myVoter;
08
09    NVPExampleConcrete()
10      : myVoter(“voter”, VoterThread::COORDINATED, 15000,
11              “toTheVoter”, “exampleResult”)
12    {
13      maxResponseTime = 20000;
14      variant = defineMyVariant();
15    }
16
17    advice NVPClass() : slice class {
18      public:
19        void variant1(){ process(); }
20        void sendResult(){output(); }
21
22        // methods to be defined
23        void variant2(){ ... }
24        void variant3(){ ... }
25    }
26  };
```

Figure 6.9: NVP strategy with *StdVoter* concrete aspect example.

The *StdVoter* object is defined in line 7. This object cannot be an attribute of the abstract aspect because it depends on the type of the Message object used to exchange the results (e.g. *Msg*), and this is application-dependent. The *StdVoter* constructor is called by the aspect constructor (line 10). The parameters taken by this constructor are described in Table 5.3. In this example, coordinated voting is selected and the maximum response time for a voting cycle is set to 15.000 microseconds. The subject of input messages to the voter is defined arbitrarily as “toTheVoter” and the subject of

output messages from the voter is defined compulsorily as “exampleResult”, the same used by the *output* method of *ExampleThread*.

The *variant* attribute set in line 14 will define which *variant* method (1, 2, or 3) will be called for processing. In this example, all nodes will be able to run any variant, and the definition about what variant they will run will be taken at runtime, using the *defineMyVariant* function. A possible implementation of this function can be to define the variant based on some non-volatile identification of the node. Finally, the application-specific methods are defined for this strategy (lines 23-24). The NVP strategy requires an extra variant in relation to the DRB strategy (line 24), but in contrast does not require the implementation of checkpointing or acceptance tests. If only TMR is implemented, there is no need to define *variant2* and *variant3*, and the variant attribute should be set to 1(one) in line 14.

6.1.4 Discussion

The basic goal of the AOP implementation shown in the previous sections was to modularize all fault tolerant code used at the application thread level, keeping the original code unchanged. The advantages of this approach are:

- It is less prone to errors in porting a non-FT system to a FT one. The task of changing an existing system to introduce fault tolerance capabilities may insert software faults in the original code. Using AOP the original code is preserved.
- The programmer can initially write applications without fault tolerance in mind, and concentrate his efforts in the development of the functional code. Using AOP, fault tolerance can be applied in a second stage, after validating the core functionality.
- It facilitates the evaluation and comparison of several FT configurations, as the developer may easily select what set of application threads will be made fault tolerant and on which strategy.
- It contributes to product line development, as single or redundant systems may be generated by introducing or not fault tolerant aspects.

- It contributes to code reuse because the same functional code can be applied in other projects with different dependability requirements.

Using this approach, the base code remains oblivious to the fault tolerant concern, but on the other hand, the aspect code is very dependent on the base code it applies to. This fact is related to the nature of fault tolerance domain, where for each FT instantiation we may need to define specific deadlines, error detection, alternative procedures, checkpoints, state coordination, voting specifications, and so on. For that reason, concrete aspects are normally heterogeneous and can target only one application thread. However, depending on the characteristics of the application process and the selected fault tolerant strategy, less application-specific code may be needed. In our opinion, a complete homogeneous fault tolerance injection is very hard to achieve.

The main drawback of using AOP for application-level fault tolerance introduction is related to the very limited availability of aspect-oriented weavers and tools for embedded development. The AspectC++ compiler used in this work is still in beta testing and has some restrictions, such as the one described in Section 6.1.2. In Section 6.4 we discuss the need to use special configuration tools for AOP development.

We conclude that AOP is very useful in the fault tolerance domain because it reduces efforts and errors in making a legacy system fault-tolerant, simplifies system development by allowing the validation of the functional part in advance, facilitates the evaluation and comparison of various FT configurations, and contributes to product line development and code reuse.

6.2 FT framework integration

The FT framework implementation is intertwined with some of the BOSS operating system classes. For instance, the *Thread* class of BOSS includes additional attributes and methods related to the fault tolerance implementation, as seen in Figure

6.5. Even in the former implementation of the FT framework[2], the original BOSS *Thread* class had to be modified to include some attributes, as seen in Figure 5.7.

Ideally, the utilization of an FT framework should not affect the OS development. The application of AOP techniques can provide the complete physical separation of the FT framework from the OS code. Therefore, the development of these concerns can be made separately and be composed, if needed, at weaving/compilation time.

6.2.1 Code generation

The process of weaving the FT framework with the operating system and further generation of the executable code is shown in Figure 6.10.

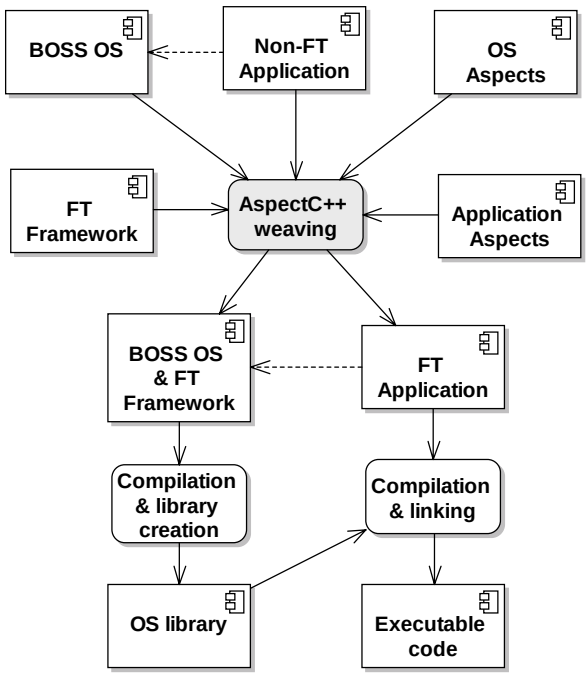

Figure 6.10: Code generation process when using AOP at the OS level.

The weaving process now also applies to the original operating system code. The weaving process at the application level occurs simultaneously with the weaving

[2] The former implementation of the FT framework is the one presented in Chapter 5. The current implementation of the FT framework is the one shown in Figure 6.5, which avoids multiple inheritance when using AspectC++, as described in Section 6.1.2.

process at the operating sytem level. The FT framework is injected into the OS by one or more aspects. There are no modifications in abstract and concrete FT aspects used at the application level. Using this approach it is possible to reduce the code size for non-FT implementations and also to apply aspects for other concerns at the operating system level, as logging, synchronization and middleware customization.

Figure 6.11 shows an alternative code generation process, where two executions of AspectC++ weaving are performed: the first for weaving at the OS code and the second for weaving at the application code. This configuration avoids the regeneration of the OS library each time the application code is changed. However, it can not be applied if the same aspect has to advice both the OS and the application. AspectC++ was designed to allow weaving on a pre-woven source code, which is the case in this configuration, as the include files related to the OS were modified by the first weaving process.

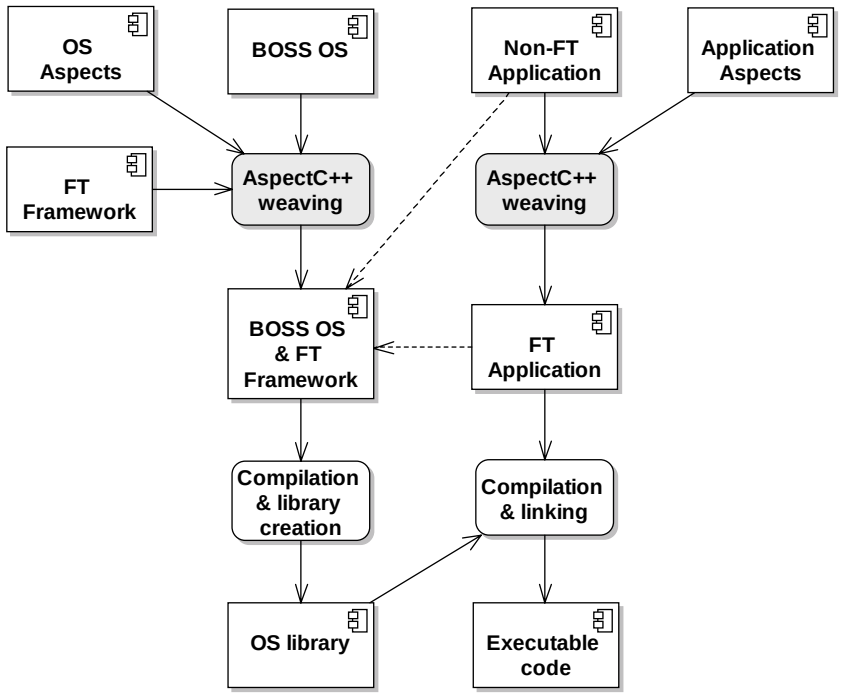

Figure 6.11: Alternative code generation process with double weaving.

6.2.2 AOP implementation

This section describes the AOP implementation that integrates the FT framework into the BOSS operating system. Two aspects were used. The first one, shown in Figure 6.12, modifies the *TimeManager* class to activate the *MiddlewareScheduler* (MS) thread at the beginning of each clock tick interval. The *timeEvent* method is called from the clock tick ISR. This aspect adds an *after* advice to this method execution (line 5), which resumes the MS thread (resetting *waitingUntil*) if this thread is not waiting a resource (e.g. semaphore).

```
01  aspect FTFramework_1: {
02    pointcut MSActivation()=
03           execution("%...:: TimeManager::timeEvent()");
04
05    advice MSActivation(): after(){
06       if(middlewareScheduler.waitingForSignalFrom == NULL){
07          middlewareScheduler.waitingUntil=0;
08    }
09  };
```

Figure 6.12: MiddlewareScheduler activation aspect.

The second aspect is presented in Figure 6.13. This aspect introduces FT attributes and methods to the *Thread* class, and also advises its constructor (line 33). The named slice class *FTThreadSlice* defines a set of data members and member functions that will be added to the *Thread* class (line 31). Most methods are virtual functions and have empty or default implementations (see Table 5.1). Others, such as *initFTThread* and *setFTStrategy* are non-inline functions whose implementations are defined in lines 26 and 27.

As a result of the aspect code defined in Figure 6.13, AspectC++ will append *FTThreadSlice* to the *Thread* class declaration (in Thread.h), but the implementation of *initFTThread* and *setFTStrategy* will be added to the *Thread* implementation file (Thread.cc).

```
01  slice class FTThreadSlice {
02  public:
03    enum FTType{NONE, FT, VOTER};
04    FTType myFTType;
05    bool isRunningFT;
06    bool isVoting;
07
08    FTStrategy * ftStrategy;
09    bool hasState;
10
11    virtual void variant1(){}
12    virtual void variant2(){}
13    virtual void variant3(){}
14    virtual void saveCheckpoint(){}
15    virtual void restoreCheckpoint(){}
16    virtual bool acceptanceTest(){return true;}
17    virtual void sendResult(){}
18    virtual void onFailure(){}
19    virtual int getState(char * stateBuff) {return 0;}
20    virtual void setState(int size, char * stateBuff){}
21
22    void initFTThread();
23    void setFTStrategy();
24  };
25
26  slice void FTThreadSlice::initFTThread() {...};
27  slice void FTThreadSlice::setFTStrategy() {...};
28
29  aspect FTFramework_2{
30
31    advice “Thread” : slice FTThreadSlice;
32
33    advice construction(“Thread”): after() {
34      tjp->target()->initFTThread();
35    }
36  };
```

Figure 6.13: Aspect for introducing FT attributes and methods in the *Thread* class.

6.2.3 Discussion

The application of AOP to integrate the FT framework into the operating system allows a complete physical separation of the FT framework from the OS code. This approach solves the problem described in Section 5.6 in regard to the maintainance of more than one version of the same operating system class, in order to optionally build the operating system without FT support.

6.3 Operating system fault tolerance

The application of fault tolerance at the operating system level requires the implementation of error detection mechanisms, as presented in Section 2.3. These mechanisms involve the execution of extra processing at pre-defined points of the OS code, also termed executable assertions. Assertions can check if preconditions and post conditions are fulfilled when performing a given OS functionality. Error detection mechanisms can also apply structural checks to detect errors in variables and data structures. Information redundancy is commonly used in order to allow the detection of errors in data structures.

The application of error detection mechanisms at the operating system level results in resource costs, such as memory size and runtime overhead. Therefore, fault tolerance at the OS level is normally avoided in resource constrained embedded systems. However, for systems demanding high level of dependability, such as safety-critical applications, the implementation of FT mechanisms in the OS can be of great importance.

In this section we presented how to implement error detection mechanisms at the operating system level using AOP. The examples shown are inspired by the work with fault containment wrappers [13, 100]. Wrappers are used to implement the interface between the application code and the OS, monitoring the flow of information and applying error detection and error handling mechanisms. The proposed wrappers were meant to be used to detect errors in off-the-shelf microkernels whose source code is not available for modifications. However, the application of wrapper with no information on the internal OS state has limited error detection capability. Consequently, the proposed wrappers require access to some selected internal OS data by means of a meta-interface, which is accessed by meta-level programming. The same predicates, or invariants, defined in [100] for semaphore error detection are implemented here in the BOSS operating system using AOP.

6.3.1 Semaphore error detection

Semaphores in BOSS were described in Section 4.2.2. As shown in Figure 4.3, the *Semaphore* class has two main methods (*enter* and *leave*) and a *counter* attribute to keep track of the number of resources available.

Two predicates are defined for the semaphore operation [100]. The first predicate defines a condition in which the counter attribute is consistent with the number of calls to the *enter* and *leave* methods, as stated in equation (1). The current value of the *counter* attribute should be equal to the its initial value minus the number of calls to method *enter* and plus the number of calls to method *leave*:

counter = init_value - #enter + #leave (1)

The implementation of this predicate as an execution assertion demands the introduction of three new attributes to the Semaphore class (init_value, #enter and #leave).

The second predicate defines a condition in which the value of the counter attribute is consistent with the number of suspended threads waiting for the semaphore. This predicate is represented by Equation (2), where the number of suspended threads should be equal to the maximum between zero and the negated value of the current counter attribute. For instance, if the counter attribute is -3 there should be 3 suspended threads waiting for this semaphore, but if the counter attribute is greater or equal than zero no thread is suspended.

#Suspended = max(0, -counter) (2)

The implementation of this predicate as an execution assertion demands the implementation of a search procedure for counting the number of suspended threads waiting for the semaphore.

The predicates described above can be applied as preconditions or post-conditions of the semaphore operation in calls to the methods *enter* and *leave*.

In Section 6.3.3 we present how to implement the verification of the above predicates in the *Semaphore* class of the BOSS operating system, using Aspect-Oriented Programming.

6.3.2 Code generation

The code generation process applied to introduce fault tolerance at the OS level is the same described in Section 6.2.1. The aspects used for OS fault tolerance are represented in Figure 6.10 and Figure 6.11 as “OS Aspects”, similarly to the aspects for integrating the FT framework into the OS.

6.3.3 AOP implementation

Figure 6.14 shows the aspect code to implement the first predicate for semaphore error detection, described by Equation (1) in Section 6.3.1.

```
01  slice class CounterSlice {
02    int initialCounter;
03    int enterCounter;
04    int leaveCounter;
05    void checkCounter();
06  };
07
08  slice void CounterSlice::checkCounter(){
09    int calculatedCounter = initialCounter
10                            - enterCounter + leaveCounter;
11
12    if(counter != calculatedCounter)
13       doErrorHandling();
14  }
15
16  aspect SemaErrorDetect1{
17
18    advice "Semaphore": slice CounterSlice;
19
20    advice construction("Semaphore"): after(){
21      tjp->target()->initialCounter = tjp->target()->counter;
22      tjp->target()->enterCounter = 0;
23      tjp->target()->leaveCounter = 0;
24    }
25
26    advice execution("% Semaphore::enter(...)") : before()  {
27      tjp->target()->checkCounter();
28      tjp->target()->enterCounter += 1;
29    }
30
31    advice execution("% Semaphore::leave(...)") : before()  {
32      tjp->target()->checkCounter();
33      tjp->target()->leaveCounter += 1;
34    }
35  };
36
```

Figure 6.14: Semaphore error detection aspect for the first predicate.

CounterSlice defines the new attributes (lines 2-4) for the *Semaphore* class that are needed for the execution of the first predicate, as well as the *checkCounter* method (line 4), which implements Equation (1) in lines 9-10 and calls an error handling routine if the assertion fails (line 13). As it is hard to diagnose and correct the system state after this kind of error, a possible error handling procedure can be to reset the node. The *SemaErrorDetect1* aspect applies *CounterSlice* to the *Semaphore* class (line 18) and defines three advices. The fist advice (line 20), initializes the introduced attributes at the *Semaphore* class constructor. The second advice (line 26), checks the predicate before the execution of the *enter* method and then increments the *enterCounter* attribute. The third advice (line 31) checks the predicate before the execution of the *leave* method and then increments the *leaveCounter* attribute.

The implementation of the predicate in Equation (2) is presented in Figure 6.15. This aspect code uses a slice (*SuspendedSlice*) that defines two new methods to the *Semaphore* class: *checkSuspended and numberOfSuspended.* The first method implements the predicate (line 6) using the second method (line 13) as a utility function that returns the number of threads suspended by the semaphore. The implementation of the *numberOfSuspended* method is not shown. The *SemaErrorDetect2* aspect introduces *SuspendedSlice* into the *Semaphore* class and defines advices to execute *checkSuspended* before the execution of the *enter* and *leave* methods.

The AOP implementations presented in Figure 6.14 and Figure 6.15 apply the predicates as preconditions to the semaphore operations. Implementations considering the predicates as post-conditions can be performed by using *after* advices.

The configuration of the semaphore functionality, i.e., if no fault tolerance is used, or if one or more error detection mechanisms are used, can be decided at compile time, by including or not the above aspects. AOP allows a complete modularization of the fault tolerance code, keeping the original semaphore implementation unchanged.

However, the AOP implementations presented to this point have a serious flaw: there is no mutual exclusion between the error detection procedure and the semaphore normal operation. A race condition can occur, for instance, if a thread is suspended

during the execution of an error detection procedure and another running thread executes an operation in the same semaphore. In this situation it is possible that the error detection mechanism results in a false indication. In order to solve this problem, synchronization primitives must be employed in the aspect code.

```
01  slice class SuspendedSlice{
02    void checkSuspended();
03    int numberOfSuspended();
04  };
05
06  slice void SuspendedSlice::checkSuspended(){
07    int calculatedSuspended =(counter >= 0 ? 0 : counter*(-1));
08
09    if( mumberOfSuspended != calculatedSuspended)
10       doErrorHandling();
11  }
12
13  slice int SuspendedSlice::numberOfSuspended(){...}
14
15  aspect SemaErrorDetect2{
16
17    advice "Semaphore": slice SuspendedSlice;
18
19    advice execution("% Semaphore::enter(...)") : before()  {
20      tjp->target()->checkSuspended();
21    }
22
23    advice execution("% Semaphore::leave(...)") : before()  {
24      tjp->target()->checkSuspended();
25    }
26  };
```

Figure 6.15: Semaphore error detection aspect for the second predicate.

The aspect shown in Figure 6.16 solves the synchronization problem described above. This aspect injects the mutual exclusion mechanism (disabling dispatching) in the semaphore implementation. In preparation for the application of this aspect, the Semaphore methods *enter* and *leave* were modified in order to expose their critical sections, which were enclosed by the new methods *enter_in* and *leave_in*. Additionally, the original calls to mutual exclusion procedures were removed. The *SemaSyncronize* aspect defines the execution of *enter_in* and *leave_in* as a pointcut for synchronization advices (lines 3-4). The *before* advice in line 6 disables the dispatch of other threads, by calling the *disableDispatch* method of *Scheduler* (see Section 4.2.2), while the *after* advice in line 10 enables the dispatch again by calling *enableDispatch.* The *SemaErrorDetect1* (Figure 6.14) and *SemaErrorDetect2* (Figure

6.15) aspects must be modified to advise over methods *enter_in* and *leave_in*, instead of methods *enter* and *leave*.

```
01  aspect SemaSyncronize{
02
03    pointcut sync()= execution(% ...::Semaphore::enter_in(...) ||
04                     execution(% ...::Semaphore::leave_in(...);
05
06    advice sync(): before() {
07      scheduler.disableDispatch();
08    }
09
10    advice sync(): after() {
11      scheduler.enableDispatch();
12    }
13
14    advice sync(): order("SemaSyncronize", "SemaErrorDetect1");
15    advice sync(): order("SemaSyncronize", "SemaErrorDetect2");
16    advice sync(): order("SemaErrorDetect1", "SemaErrorDetect2");
17  };
```

Figure 6.16: Synchronization aspect applied to the *Semaphore* class.

The precedence of the several *before* advices that affect the *sync* pointcut is defined by advice ordering declarations in lines 14-15. Three aspects can inject code at these join points (execution of the *enter_in* and *leave_in* methods): *SemaErrorDetect1*, *SemaErrorDetect2* and *SemaSyncronize*, all them advising before the join points. The given ordering declarations establish the following precedence, from higher to lower: *SemaSyncronize*, *SemaErrorDetect1* and *SemaErrorDetect2*. Therefore, the synchronization aspect is the first *before* advice to be executed and disables dispatch for the whole period concerning error detection and critical semaphore operation. The *after* advice restoring the dispatch mechanism is executed after the exit of the semaphore critical section.

A sequence diagram representing the *enter* method behavior after weaving is shown in Figure 6.17. In this example of operating system FT implementation, we could see how AOP was able to compose three crosscutting concerns: semaphore basic functionality, fault tolerance and synchronization. In special, the synchronization aspect *SemaSyncronize* can be modified to apply this kind of mutual exclusion mechanism in other operating system functionalities, just by adding the desired join points to the *sync* pointcut in Figure 6.16. This experiment has been performed in the context of this work and has been reported in [4].

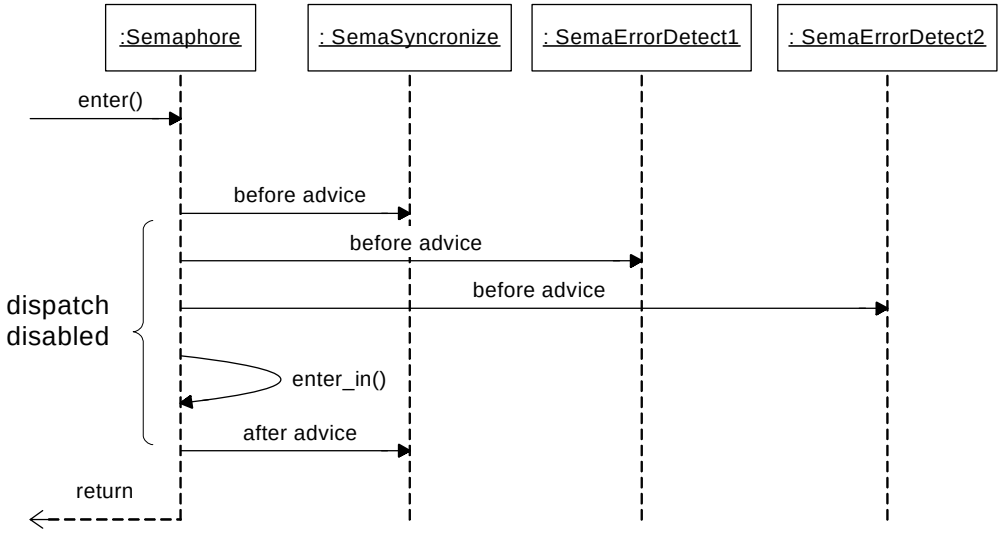

Figure 6.17: *Semaphore* enter method sequence diagram.

6.3.4 Discussion

The examples presented in the previous section show that AOP can be used effectively to introduce fault tolerance at the operating system level. In the context of this work, this approach has not been further explored, as our main target of fault tolerance introduction was the application level. The examples shown here were inspired in fault containment wrappers, but other fault tolerance mechanisms can be applied.

6.4 Project configuration using AOP

As discussed in the previous sections, the application of AOP in projects involving an operating system, frameworks and applications can make use of diverse code generation processes. Additionally, using AOP the project configuration depends on the set of aspects to be woven into the base code, which must be defined prior to code generation. AspectC++ only considers in the weaving process the aspects contained in files with the .ah extension. Thus, the simpler way to disable an aspect is to rename the aspect file with a different extension (e.g. .ah_off). Other option is to copy the selected aspect files from a repository to the project directory.

An important project configuration issue is dealing with two or more versions of a base source code. The following types of base code software versions can coexist in a project:

- The original version. This version may have one or more functionalities that can be introduced by aspects. An example of this situation is the original version of the Semaphore class in BOSS, which uses synchronization features, as described in Section 6.3.3.
- A refactored version for applying AOP functionality. Some refactoring in the source code may be needed to allow the AOP application, as for instance, the creation of new methods to expose joint points to the aspect code, as described for the Semaphore class in Section 6.3.3.
- A modified version without a previous implemented functionality. This consists of a modified version that had some functionality removed from the base code in order to be introduced by aspects.

Therefore, if a project can be configured to implement a given feature with or without AOP, more than one version of the base source code must be maintained, which impacts software maintenance and evolution. Possible related scenarios include projects where AOP is being evaluated as an alternative implementation or projects where AOP is applied just for debugging and is not employed in final versions. In these cases, the management of more than one version of the same source code file may be required.

In the context of this work, project configuration had to be very flexible, in order to evaluate AOP implementations against pure OOP implementations. The configuration was entirely based on bash scripts running from the Linux's command line. Scripts were used to: (a) define if AOP is applied; (b) define the AOP code generation process; (c) enable or disable individual aspects; and (d) select base source code files to be used in the code generation process. Ideally, aspect-oriented projects should have its configuration supported by special graphic tools used for product line software development, such as pure::variants [95]. An alternative approach might be the adoption of the configuration language used for building the Linux kernel and a buildroot system with graphical support like xconfig and gconfig.

6.5 Summary

This chapter has described how to apply Aspect-Oriented Programming to support the implementation of fault tolerance at various software levels and purposes.

The main target of AOP application is the introduction of fault tolerance at the application level, using the FT framework described in the previous chapter. This approach can be used to convey fault tolerance to an existing application without modifying its source code. Additionally, it modularizes the fault-tolerant code with advantages in flexibility and maintenance. The implementation is based on the definition of general abstract aspects for each FT strategy and application-specific concrete aspects that define the target thread and the required parameters and additional methods for the FT strategy execution. Abstract and concrete aspects have been explained based on an example of FT application using DRB and NVP.

The utilization of AOP to apply fault tolerance at the application level has several benefits: it reduces errors in introducing fault tolerance to legacy systems; it allows the validation of the functional part in advance; and it contributes to product line development and code reuse. The main drawback of AOP application for embedded systems development is the limited support in terms of aspect weavers and tools.

The integration of the FT framework into the operating system has also been discussed. Previously integrated to the operating system, the FT framework has been completely separated from the OS code, allowing its optional integration to be postponed to weaving/compilation time. This modularization reflects in easier software maintenance and reduced memory footprint for non-FT applications.

This chapter has also described the application of AOP to implement fault tolerance in the OS, by adding fault tolerance error detection mechanisms implemented as executable assertions that verify predicates or invariants related to the OS basic functionality. The FT functionality is introduced by aspects that can be optionally selected. The application of AOP for this purpose, as well as the relationship between the fault tolerance and the synchronization concerns has been exemplified using the semaphore functionality in BOSS.

Although AOP aims to simplify software maintenance, there may be projects in which AOP and pure OOP versions must coexist. In this situation several versions of the base code should be maintained, impacting software maintenance and evolution and increasing the complexity of the project configuration process; therefore the utilization of a graphic product line configuration tool is recommended.

Chapter 7

Case studies and evaluation

This chapter describes the case studies developed to test the proposed FT framework and compares performance and costs of several configurations and implementations. Two case studies are presented: a sorting application and a radar filtering application. The description of the development and test environment applied in this work is initially presented.

7.1 Development and test environment

This section describes the development and test environment, including the target and host systems and related software tools.

7.1.1 Target systems

The target board selected for the testing environment was the STK823L starter kit board from TQ Components [113]. The board uses a MPC823 microprocessor, which integrates a high performance PowerPC embedded processor with a Communication Processor Module (CPM) and a System Interface Unit (SIU). This microprocessor has a 32-bits RISC architecture with 2 KB instruction cache and 1 KB data cache. The CPM provides support for Ethernet, serial communications including USB, I^2C and SPI. The SIU contains a memory controller, a real-time clock and a PCMCIA interface. The microprocessor is mounted in a TMQ823L module that provides 8 MB of flash memory and 16 MB of SDRAM. A clock of 80 MHz is used in this module, which results in a processing power of about 100 MIPS. This module is connected to the STK823L main board that provides power DC conversion and several connectors for I/O and debugging.

The communication among PowerPC boards in the testing environment was performed using an Ethernet network. Figure 7.1 shows a testing configuration using three PowerPC 823 boards and a notebook computer connected by Ethernet interfaces.

As mentioned in Section 4.1, the BOSS operating system has a version that runs on top of the Linux operating system. Therefore, any hardware running Linux is a potential target for BOSS applications. In fact, notebooks and desktop computers were used extensively as targets in the development and testing phases of this work. In order to improve the real-time behavior of the BOSS applications running over Linux, a modification in Linux version of BOSS was implemented, changing the scheduling priority of the BOSS process to the highest in the system. In the configurations of the case studies presented later in this chapter, a notebook computer was used to act as a sensor or actuator. However, the utilization of personal computers (PC) to implement

fault-tolerant applications was avoided, because this work aims to deliver fault tolerance to embedded systems.

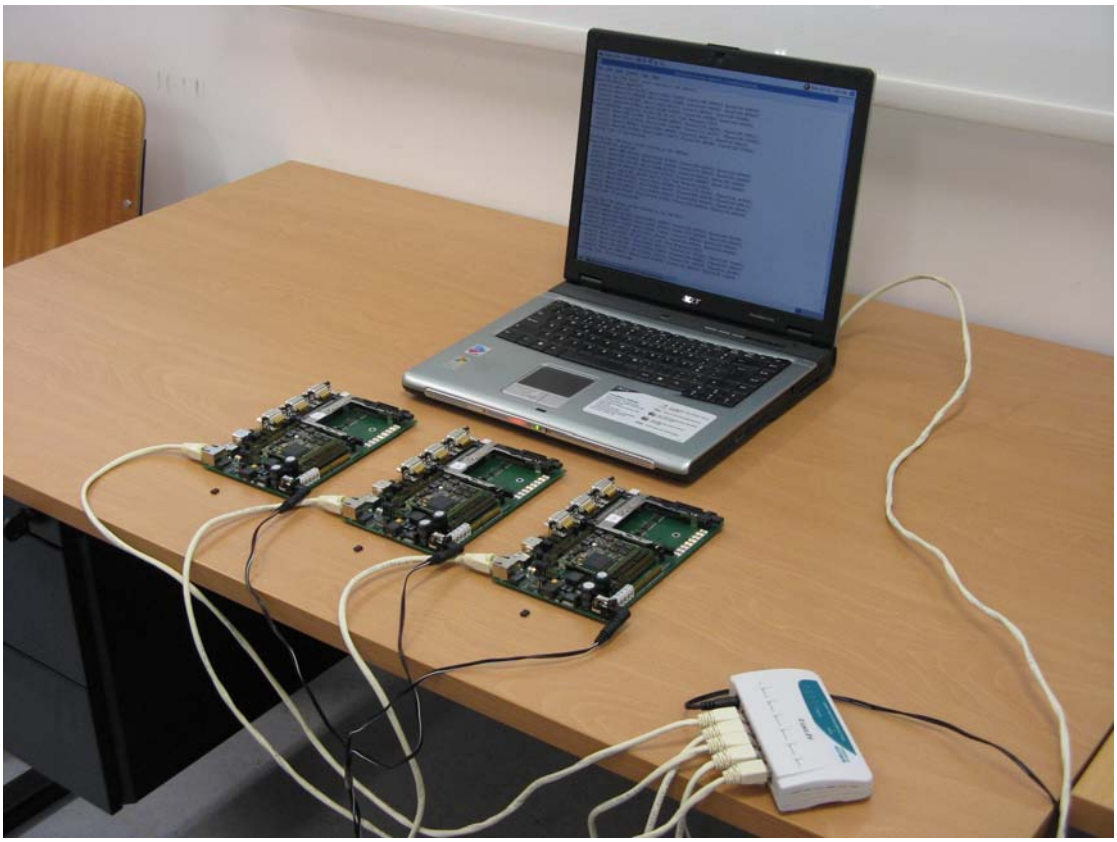

Figure 7.1: Test environment.

7.1.2 Host system

The host system used in this work consisted of a PC running Linux and a cross-compiler based on GNU gcc versions 3.2.3 or higher. Several Linux distributions were used, such as Fedora Core 3/4/5, and Ubuntu 5.04. The AspectC++ weaver version was 1.0pre3.

The cross-compiler toolchain received from FIRST uses an old version of GNU gcc (2.9) that is not compatible with current AspectC++ versions. Therefore, in this work, several other toolchains were tested, including a built from scratch. Eventually, the MPC8xx POMP cross-compiler toolchain [33] was selected because of its better compatibility with the PowerPC libraries received from FIRST.

7.1.3 Porting BOSS to the target board

The BOSS operating system had been previously ported to a PowerPC 823 based board (transCON module from Yacoub Automation [123]) by the FIRST institute. However, the TQ board selected as target had a different configuration in terms of clock frequencies and memory configuration. Consequently, the initialization code and the PowerPC libraries received from FIRST had to be modified in order to run in the new board. Differently from the original port, in this port the monitor program received from the board vendor is kept in flash memory and starts the board initialization, transferring control to the OS code in a specific flash memory position (if a jumper is removed). The OS initialization code concludes the board initialization and finally loads the OS and application code to SDRAM and jumps to it. This configuration allows the utilization of the board supplier's monitor software for loading programs in SDRAM or flash memory if the mentioned jumper is not removed. The interface to the target board for loading and debugging programs is based on EIA-232 communication at 115 kbps.

7.2 Case study I: sorting application

The first case study developed to evaluate the application of the FT framework was a sorting application. This application aims to sort an array of integer numbers generated at random using different algorithms as variants: Insertion Sort, Selection Sort and Bubble Sort. We chose a sorting application because they are commonly used as test cases for software fault tolerance strategies, such as the one described in [120].

7.2.1 Testing configurations

In this case study the following configurations were employed: non-fault-tolerant (non-FT), RB, DRB and NVP. Figure 7.2, Figure 7.3 and Figure 7.4 show these configurations for Non-FT/RB, DRB and NVP respectively. In these figures, broadcast messages are represented by buses with the message subject on top.

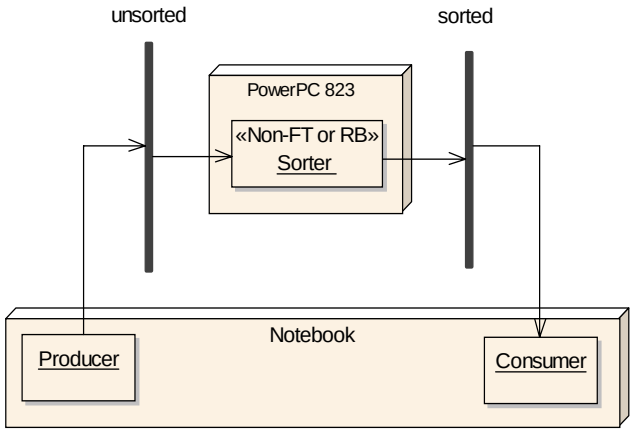

Figure 7.2: Case study I - non-FT or RB configuration.

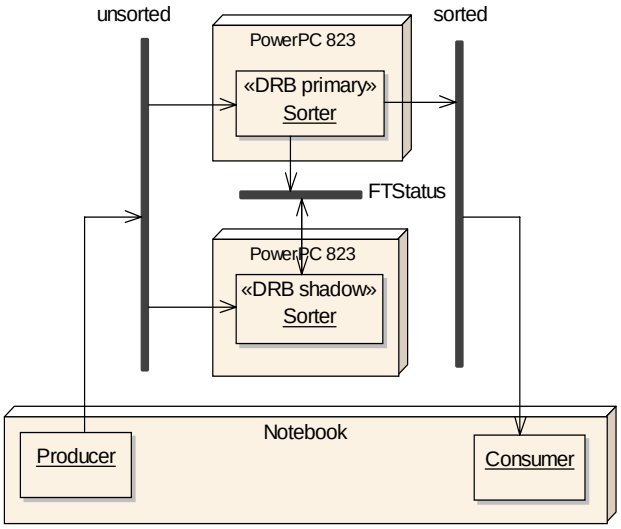

Figure 7.3: Case study I - DRB configuration.

The *Producer* thread is a BOSS thread that runs over Linux in the notebook computer and generates 200 integer numbers that are sent by an external message to the network using the string "unsorted" as subject. The *Producer* thread sends this message periodically (each 2 seconds).

The *Sorter* thread is a BOSS thread that runs in the PowerPC boards and sorts the incoming numbers using different sorting algorithms. In the non-FT configuration, only one algorithm is executed and no FT mechanism is applied. In RB and DRB configurations, Insertion Sorts runs as the primary block and Selection Sort runs as the recovery block. In the NVP configuration, each node runs a different algorithm:

Insertion Sort as variant 1, Selection Sort as variant 2 and Bubble Sort as variant 3. The sorted array of integers is sent out by a message using “sorted” as subject in all configurations except NVP, in which the subject “unvoted” is used. In case of the NVP configuration, each node has an additional *Voter* thread that defines the final result based on incoming “unvoted” messages. For this particular NVP configuration free voting is applied, and therefore all voters send theirs results concurrently.

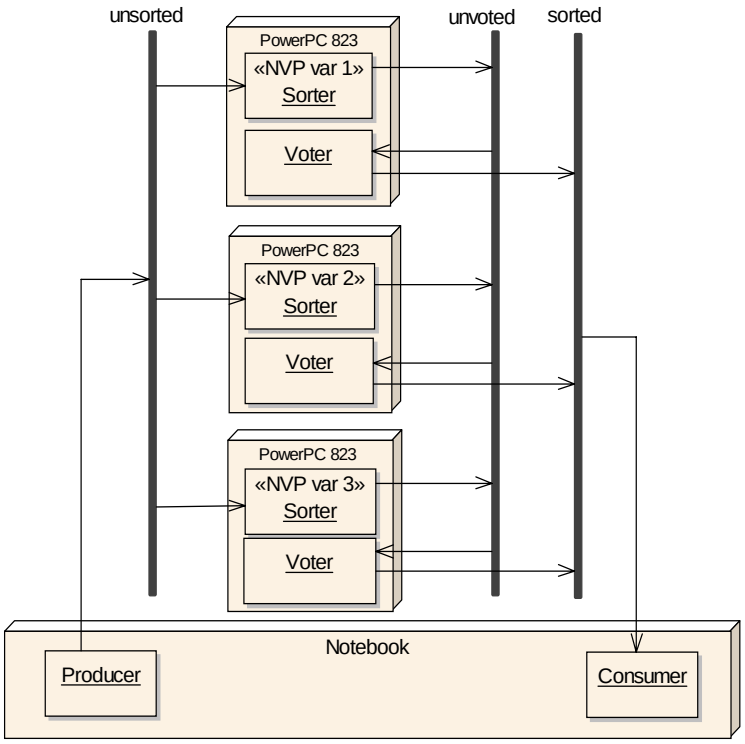

Figure 7.4: Case study I - NVP configuration.

The *Consumer* thead is a BOSS thread that runs over Linux in the notebook. It receives messages with “sorted” as subject and displays its data on the computer screen for verification. Additionally, it computes the total execution time of the sorting application, considering the time interval from the moment that the *Producer* thread is about to send a message to the moment that the resulting message is received by this thread. As discussed in Section 4.4.1, redundant messages from voters can be discarded automatically by the BOSS middleware based on message identification generated by the *Producer* thread and propagated by *Sorter* and *Voter* threads.

In this case study, *Sorter* threads are stateless, and therefore the FT versions of these threads do not need to implement the *getState* and *setState* methods described in Section 5.3.3 and referred to in Table 5.1.

7.2.2 Execution time measurements

In this experiment, the execution times of the configurations described in the previous section were measured. A **total execution time** is defined as the time interval between sending a message with "unsorted" subject (by the *Producer* thread) and receiving the message with "sorted" subject (at the *Consumer* thread). A **local execution time** is defined as the time interval between receiving a message with "unsorted" subject (at a *Sorter* thread) and sending the message with "sorted" subject (by a *Sorter* or *Voter* thread). Consequently, local execution times exclude any communication overhead between the notebook computer and the PowerPC boards.

The execution times for several configurations and failure conditions are shown in Table 7.1 and Table 7.2. Table 7.1 presents **local** execution times, while Table 7.2 presents **total** execution times. The results shown in these tables consist of an average of 10 executions. Table 7.3 presents the time settings used in this case study for the several configurations, as they have effect in some measured execution times.

Table 7.1: Case study I - local execution time.

Configuration	Failure conditions			
	No failure	Failure in variant 1	Failure in one node	Failure in two nodes
Insertion sort	1743	-	-	-
Selection sort	3511	3511	-	-
Bubble sort	3123	3123	-	-
RB	4250	8249	-	-
DRB	4781	8701	20375	-
NVP	9444	12716	10792	24175

Table 7.2: Case study I - total execution time.

Configuration	Failure conditions			
	No failure	Failure in variant 1	Failure in one node	Failure in two nodes
Insertion sort	9704	-	-	-
Selection sort	12495	12495	-	-
Bubble sort	10918	10918	-	-
RB	12353	17105	-	-
DRB	12690	17339	27932	-
NVP	18815	21345	19907	32504

Table 7.3: Case study I - time settings.

Setting	Value (microseconds)
Clock tick interval	2,000
RB maximum response time	10,000
DRB maximum response time	20,000
NVP maximum response time	6,000
Voter maximum response time	20,000

For each configuration, four failure conditions were applied. The first one was a condition with no failures in any variant or node. The Insertion Sort algorithm presents the shortest execution time, and therefore it was selected to run as variant 1 (primary block in RB/DRB). As seen in these tables, the FT configurations have longer executions times because of the coordination with the *MiddlewareScheduler* thread, as described in Chapter 5. The clock tick interval definition, set in this case study to 2,000 microseconds, affects the execution time of all FT strategies. This setting also affects the communication times between nodes, as the distribution of external incoming messages to threads are performed with a period of two clock ticks (4,000 microseconds). In special, the NVP configuration is supposed to be the slowest, as additional time for results dissemination and voting is needed. Consequently, the maximum response time in a NVP configuration is bound to the sum of the maximum response time of NVP and Voter threads.

Figure 7.5 presents a graphical comparison of the several configurations for the no-failure condition. The left part of the graph is related to local execution times while the right part of the graph is related to total execution times. The lines above the bars represent the standard deviation in 10 measurements. Configurations that depend on message communications present the largest standard deviations.

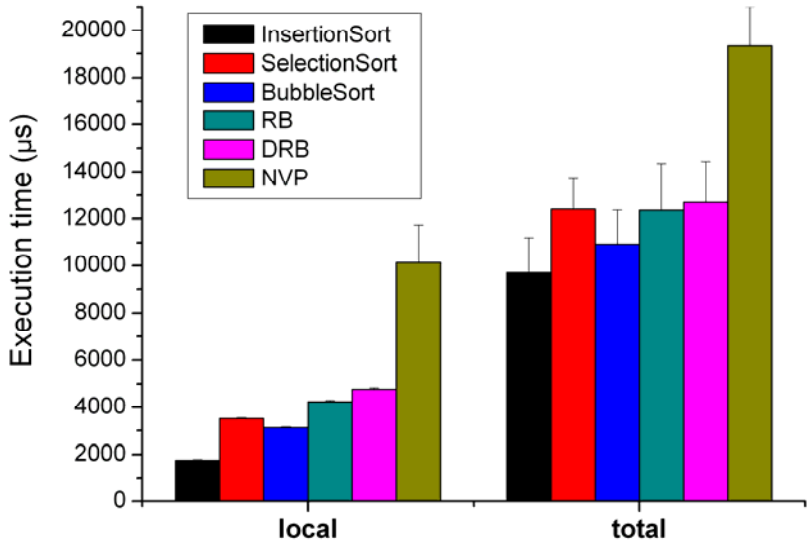

Figure 7.5: Case study I - no-failure condition.

The second failure condition presented in Table 7.1 and Table 7.2 is related to the failure of variant 1. An error was simulated by introducing an out of order integer to the results of the Insertion Sort algorithm. The error is detected by the acceptance tests of the RB/DRB strategies, which triggers the execution of variant 2 (Selection Sort). This error is masked by the voting mechanism in NVP, as variants 2 and 3 generate identical results. Figure 7.6 shows a comparison of local execution times between the no-failure condition and the variant 1 failure condition. The extra time spent by RB/DRB configurations is due to the execution of the second variant, while the longer execution time for NVP is explained by the fact that local execution times are measured in the node that runs variant 1, and therefore the voting decision was taken only after receiving the results of the other two nodes.

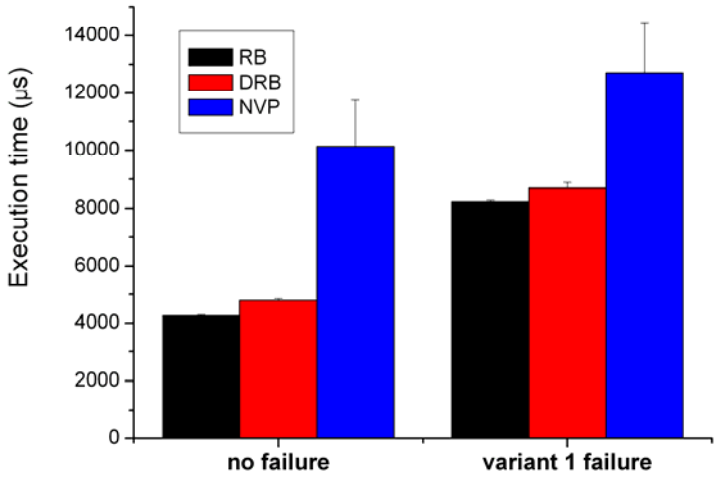

Figure 7.6: Case study I – comparison between no-failure and variant 1 failure conditions.

The third failure condition presented in Table 7.1 and Table 7.2 is related to a silent failure of a node. In that case, non-FT configurations fail as well single node FT configurations as RB. This failure was simulated by switching off the first node (the one running variant 1). Before turning it off, this node is acting as a primary node in the DRB configuration. Measurements of local execution times were taken in the second node (running variant 2).

A comparison of local execution times between the no-failure condition and the one node failure condition is shown in Figure 7.7. The NVP execution time is not affected much, as the voter in the second node will get to a decision after receiving a message from the local NVP thread and the external message from the NVP thread of the third node. However, for the DRB configuration, the failure of the primary node has to be detected by the shadow node and consequently the DRB maximum response time of 20,000 microseconds is taken into account (see Table 7.3). This larger execution time for the DRB configuration only applies for the first activation after a primary node failure because the shadow node will change its role to primary and the execution time will drop to the same value of the no-failure condition. The RB configuration does not survive to a node failure and consequently its execution time is not represented in Figure 7.7 for this condition.

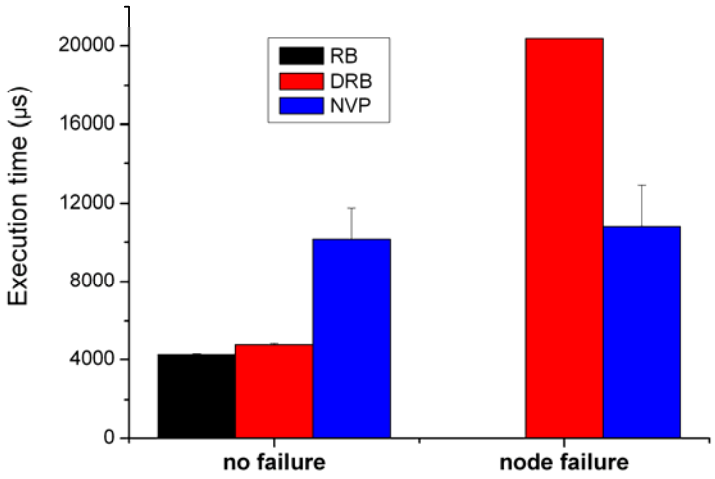

Figure 7.7: Case study I - comparison between no-failure and one node failure conditions.

Finally, the last failure condition presented in Table 7.1 and Table 7.2 is related to the silent failure of two nodes. If that situation occurs, only the NVP configuration succeeds. In that case, the NVP execution time depends on the deadline for voting, which is determined by the maximum response time of the *Voter* thread (see Table 7.3).

7.2.3 Runtime costs

The execution times results achieved in this case study are in accordance with the FT framework implementation details described in Figure 5.5 and validate the correct functionality of the framework. However, a more demanding configuration was defined in order to measure runtime overheads imposed by the FT framework, considering different version implementations, using or not AOP (Section 6.1) and FT scheduling (Section 5.4.2).

The measurement of runtime overheads in these experiments is based on CPU utilization. The BOSS idle thread computes the sum of CPU utilization of all other threads (including OS and application threads) based on its inactive periods.

This test configuration consisted of the same sorting application, but now being executed concurrently by 8 *Sorter* threads in a single target board as shown in Figure 7.8. The *Producer* thread runs periodically in the target board and generates 5

random integer numbers that are sent to the *Sorter* thread using a local message. Two versions of *Sorter* threads are implemented: non-FT and RB. In both versions, no error is simulated and these versions always execute the Insertion Sort algorithm. At the end of their processing, the *Sorter* threads prepare the output message with the sorting results, but this message is not sent so as to decrease the CPU utilization.

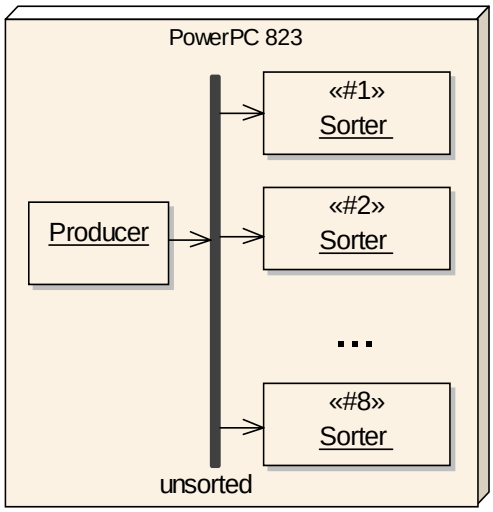

Figure 7.8: Case study I - CPU utilization configuration.

In this experiment, five different software versions were evaluated:

- Non-FT #1: in this version the *Sorter* threads do not apply any fault tolerance and the application program was linked to a version of the OS without the FT framework.
- Non-FT #2: same as above, but linked to a version of the OS integrated to the FT framework.
- FT: in this version the *Sorter* threads apply the RB strategy.
- FT-AOP: same as FT, but using AOP to implement fault tolerance.
- FT-Sched: same as FT, but implementing FT scheduling.

The CPU utilization results related to each version for several activation frequencies of the *Producer* thread are shown in Figure 7.9 and represent an average CPU utilization over a period of one minute, using maximum compiler optimization (O2).

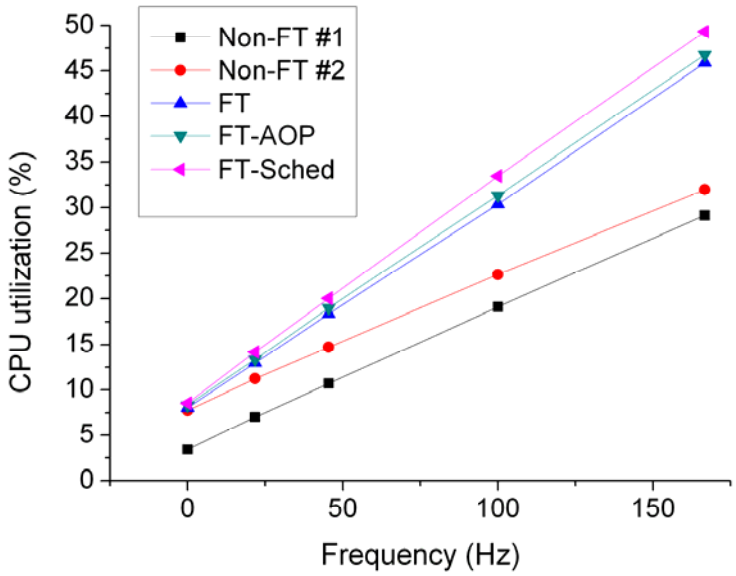

Figure 7.9: Case study I - CPU utilization results.

In Figure 7.9, a zero frequency indicates a condition where the *Producer* thread is always suspended and therefore no sorting is performed. Other frequencies plotted in this graph correspond to the following activation periods of the *Producer* thread: 46, 22, 10 and 6 milliseconds. As it can be noticed, CPU utilization is directly proportional to the activation frequency. The minimum activation frequency selected for this experiment was 6 ms, which is a multiple of the clock tick interval of 2 ms. It has been verified that this period is sufficient for delivering all input messages to the *Sorter* threads in the first clock tick period, to execute the sorting/acceptance test in the second period and to prepare the results for sending in the last period.

The difference in CPU utilization between the Non-FT #1 and Non-FT #2 versions ranges from 3 to 4.3%. This overhead is related to the activation of the *MiddlewareScheduler* (MS) thread at the beginning of each clock tick period, even if no FT threads exist. The CPU utilization spent by MS is similar to the utilization of the Non-FT #1 version with no application threads running (about 3%). In this no-load condition, the only BOSS thread running is the one that checks for new external messages, which is activated each two clock tick periods. If this thread is released in every clock tick period, the no-load utilization for the Non-FT #1 version rises to 6%.

As can be noticed in Figure 7.9, the FT version has a higher CPU utilization than Non-FT versions. This overhead is caused by the acceptance test executed at the end of the sorting algorithm and also by the coordination between the MS thread and the FT thread, as described in Section 5.4.1.

The FT-AOP version introduces a small overhead compared to the FT version (non-AOP). The same happens with the FT scheduling version. These overheads are dependent on the activation frequency and the number of FT threads. In the worst case scenario, a large number of FT threads execute with a high activation frequency, which in this experiment corresponds to 8 FT threads and 167 Hz (6ms period). Figure 7.10 presents a graphical representation of the CPU utilization results for such situation, considering two different compiler optimizations: O2 (maximum) and O0 (none). As it can be noticed, the optimized code reduces CPU utilization, especially in the case of AOP implementation (about 11%).

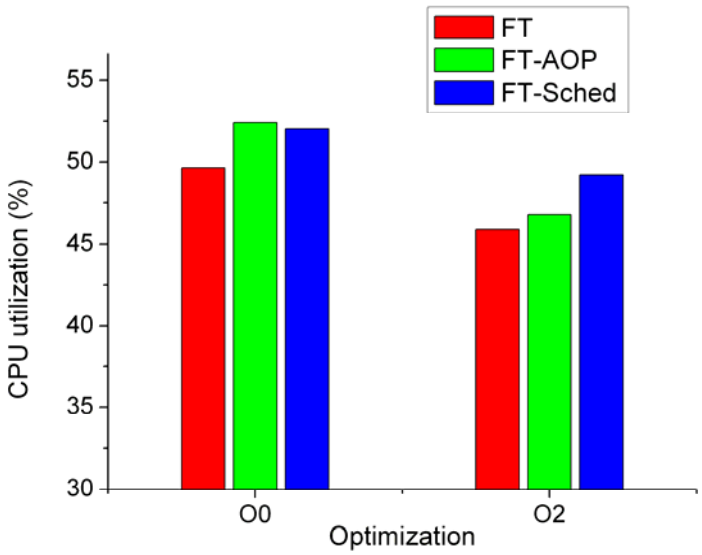

Figure 7.10: CPU utilization comparison for AOP and FT scheduling versions.

In comparison with the FT version, the FT-AOP version implies in a higher runtime overhead of 0.34% per FT thread for non-optimized programs and 0.11% per FT thread for optimized programs. This corresponds to an extra runtime of 21 and 6 microseconds respectively, in each FT thread activation.

The comparison of the version using FT scheduling with the standard FT version shows a higher runtime overhead per FT thread of 0.30% for non-optimized programs and 0.42% for optimized programs.

It should be noticed that the runtime overheads presented above are directly dependent on the activation frequency and this experiment was conducted as a worst case scenario. Consequently, real applications will probably have smaller overheads as usually lower activation frequencies are employed.

7.2.4 Memory costs

The same configuration used to measure runtime costs was applied to determine memory costs. The results were obtained using the *size* utility. Table 7.4 shows the memory footprint sizes in bytes for code (text), data and unitialized data (bss), considering the O2 compiler optimization option. Besides considering the program versions described in the previous section, this table also includes the memory footprint of the original BOSS operating system and the BOSS version integrated to the FT framework. The results for these two versions were obtained by compiling an empty application using the corresponding versions of the OS library.

Table 7.4: Memory footprint results.

version	text	data	bss	total
BOSS	53,795	3420	158,888	216,103
BOSS + FT framework	61,047	4020	183,568	248,635
Non-FT #1	57,027	3708	177,664	238,399
Non-FT #2	64,263	4428	202,504	271,195
FT	64,863	4440	202,952	272,255
FT-AOP	65,067	4440	202,968	272,475
FT-Scheduling	65,247	4440	203,080	272,767

Figure 7.11 compares total memory sizes of the two operating system versions (with and without FT framework) and their respective sorting applications (with and without FT), based on data from Table 7.4. The application footprint is much smaller

than the operating system footprint in both cases (about 10%). The inclusion of the FT framework into the OS code increases its total memory footprint in 32KB (15%). Besides, the introduction of fault tolerance into the application code increases its memory footprint in 1.3KB (6%). The total memory cost of the FT implementation in relation to the non-FT implementation in this experiment is 34KB (14.2%).

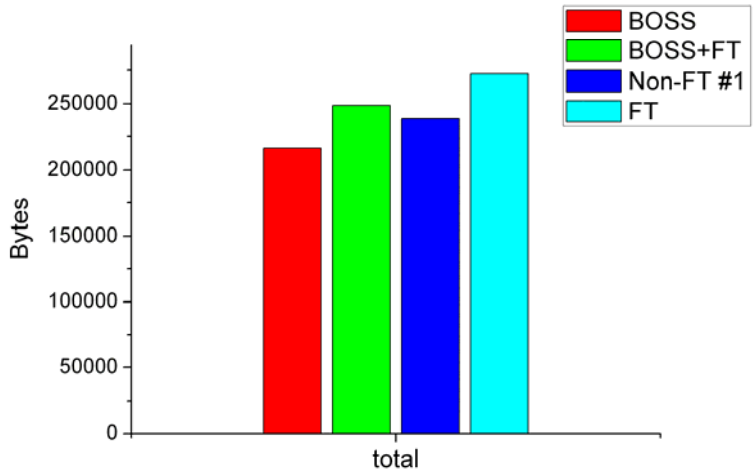

Figure 7.11: Comparison of FT and non-FT memory footprints.

Figure 7.12 presents a graphical representation of memory footprints related to the several sorting application versions described in the previous section.

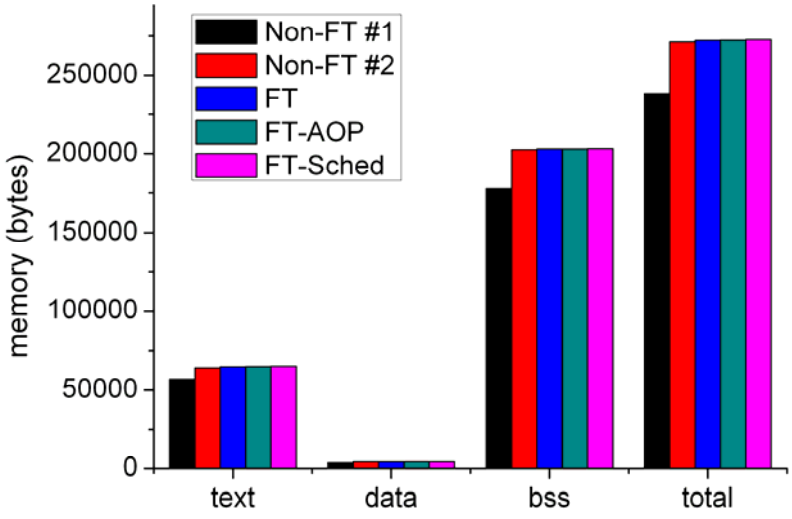

Figure 7.12: Case study I - detailed comparison of memory footprint.

In Figure 7.12 we can notice that the memory footprint differences among the versions are very small, with exception to the Non-FT #1 version, which does not include any fault tolerance at the OS and application levels. That difference is about 14.2% for the total memory size, as presented earlier.

Table 7.5 shows the additional memory costs of the AOP version and the FT scheduling version in relation to the standard FT version. The FT-AOP implementation consumes more 204 bytes of code and 16 bytes of unitialized data than the normal FT implementation. This increase in code size is caused by inlining *after* and *around* advices that make use of the AspectC++ jointpoint data structure. The extra memory for bss is related to the creation of aspect objects and pointers, and is not affected by the number of FT threads in use.

Table 7.5: Additional memory costs for AOP and FT scheduling versions.

differences	text	data	bss	total
FT-AOP to FT	204	0	16	220
FT-Scheduling to FT	384	0	128	512

The FT scheduling implementation also demands additional code and unitialized data memory: 384 and 128 bytes respectively. The extra code is due to implementation of the EDF scheduling by the MS thread. The additional data is related to the inclusion of new attributes in the *Thread* and *MiddlewareScheduler* classes. The memory cost of FT scheduling related to bss depends on the number of application threads. The results shown in Table 7.5 consider 8 FT threads. If only one FT thread is used the additional bss memory for FT scheduling reduces to 72 bytes. In contrast, the code memory cost is not affected by the number of application threads.

7.2.5 FT scheduling tests

The test configurations described in the previous sections were designed to run without deadline violations even when not applying the FT scheduling mechanism provided by the FT framework. For instance, the configuration used to measure runtime overhead described in Section 7.2.3, despite having 8 RB threads with equal

priority and high activation frequencies, was able to finish all computations within the maximum response time of 4,000 microseconds, as each FT thread had a small processing time and no errors were simulated in the primary variant.

In order to test the outcomes and benefits of FT scheduling, two special test configurations were designed. These configurations were also based on a single board target running concurrent sorting applications, as shown in Figure 7.8, but using a small number of FT threads with different processing times and deadlines.

Table 7.6 presents the settings for these configurations. The first configuration has two FT threads and the second has three FT threads. Both configurations run the RB strategy. The number of integers sorted by each thread, as well its maximum response time is shown in this table. The measured processing time for variants 1 and 2 (primary and recovery blocks) in each thread is also shown.

The configurations presented in Table 7.6 were implemented in two software versions: with and without FT scheduling. The FT scheduling version was successful in meeting the deadlines in both configurations, even when a failure in variant 1 of all threads is simulated. In contrast, the standard FT version only is able to meet the deadlines for all threads if no failures are simulated. The results are summarized in Table 7.7.

Table 7.6: FT scheduling test configurations.

Settings	Configuration # 1		Configuration # 2	
Number of RB threads	2		3	
Number of integers	200		200	
	100		100	
	-		100	
Variants 1 and 2 processing times (microseconds)	1,840	3,242	1,840	3,242
	441	885	441	885
	-	-	441	885
Maximum response time (microseconds)	10,000		14,000	
	6,000		8,000	
	-		6,000	

Table 7.7: FT scheduling test results.

Version	Failure condition	FT Thread	Config #1	Config #2
FT	No failures	1	OK	OK
		2	OK	OK
		3	-	OK
	Failure in variant 1	1	OK	OK
		2	failure	failure
		3	-	failure
FT scheduling	No failures	1	OK	OK
		2	OK	OK
		3	-	OK
	Failure in variant 1	1	OK	OK
		2	OK	OK
		3	-	OK

This test results show that the FT scheduling mechanism can be useful for eliminating deadline violations in situations where multiple FT threads with different computing times and deadlines are activated simultaneously.

7.3 Case study II: radar filtering application

The second case study developed to evaluate the application of the FT framework was a radar filtering application. Radar filtering is a real-time application commonly used in Command and Control (C^2) systems. In contrast with case study I, this case study applies single version fault tolerance techniques and FT state threads.

In this application a notebook computer simulates a radar system and periodically generates detection data of several planes. The data generation includes simulated errors in bearing and distance, typical of this kind of equipment. The radar data is sent to the target systems, which filter the planes' position, using an alpha-beta filter, and calculate the planes' course and speed.

7.3.1 Testing configurations

Three configurations were applied, as shown by the UML deployment diagrams of Figure 7.13, Figure 7.14 and Figure 7.15. The first configuration uses a single node version of the filtering application, without any fault tolerance mechanism. The other configurations implement the PSP and TMR strategies. In these figures, broadcast messages are represented by buses with the message subject on top.

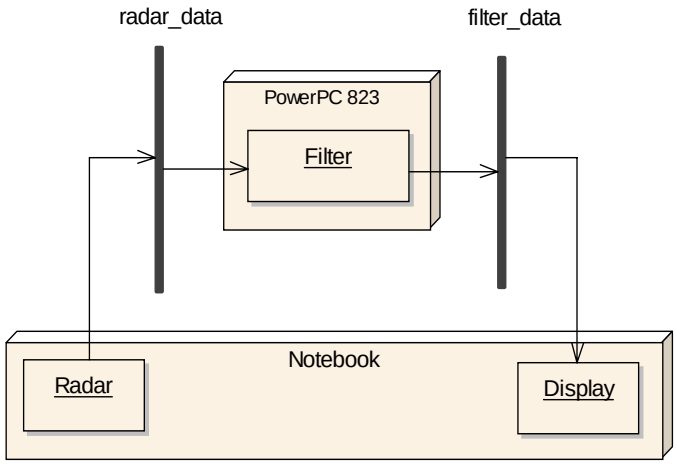

Figure 7.13: Case study II - non-FT configuration.

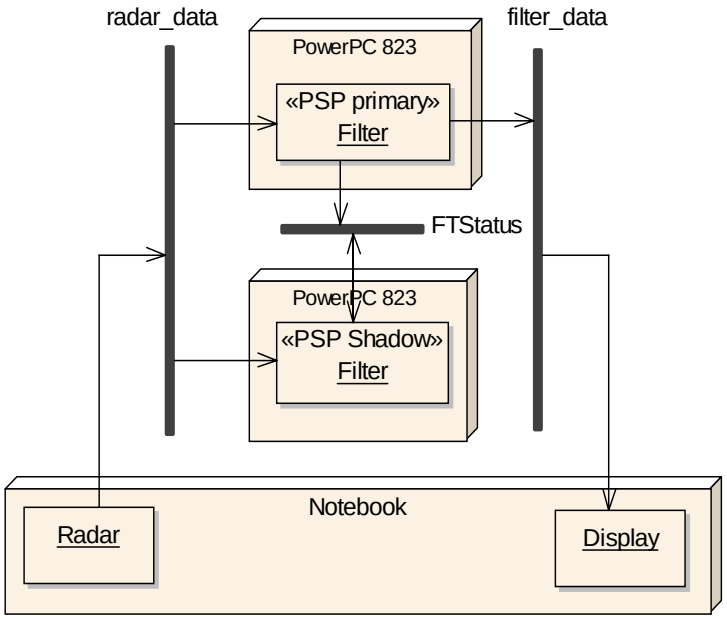

Figure 7.14: Case study II - PSP configurtion.

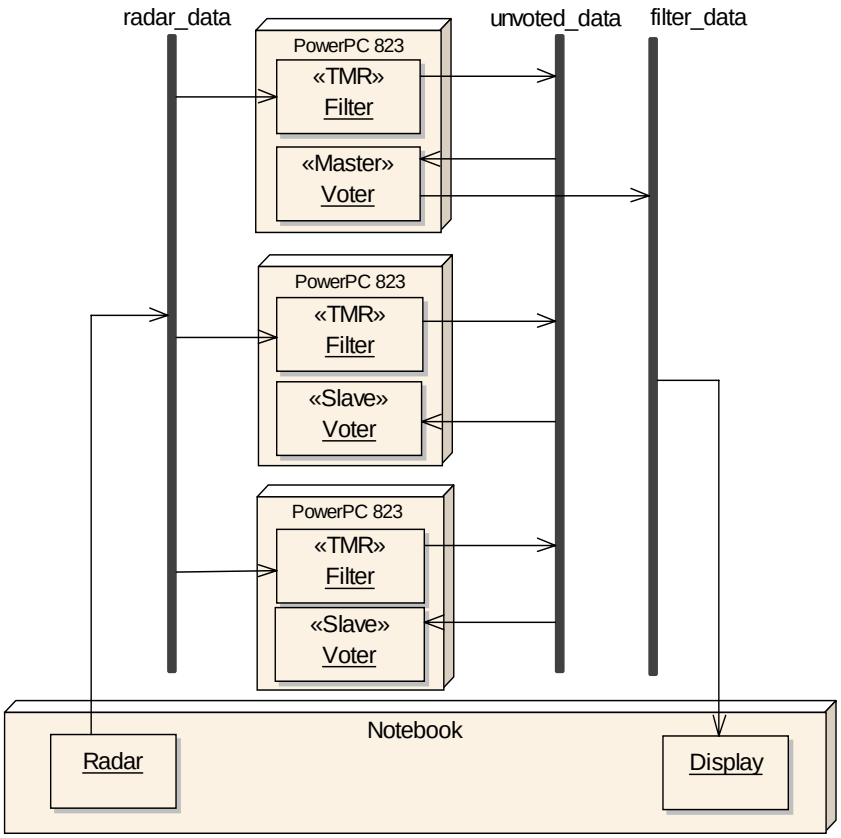

Figure 7.15: Case study II - TMR configuration.

The *Radar* thread is a BOSS thread that runs in the notebook computer. It generates radar simulated detection data (bearing and distance) of 4 planes, including typical radar measuring errors, and periodically sends this data using the string "radar_data" as subject. The period of this message depends on the selected antenna rotation period of the simulated radar. In this case study, a period of 2 seconds was selected (30 RPM). The planes have initial courses generated at random, but all have the same speed of 100 m/s. When they reach a given distance from the simulated radar its course is reverted, so as to keep them at a 10 kilometers range.

Filter is the BOSS thread that runs in a PowerPC board and filters the radar data, removing the measurement errors in plane's position and also calculating its course and speed. The filtering algorithm is an alpha-beta filter using two variable parameters that depend on the number of planes positions received previously. This case study uses a single version of the filtering algorithm, even when executing fault-tolerant configurations.

The planes' filtered position, as well as its course and speed are sent back to the notebook computer using "filter_data" as subject. Then, they are presented in the command line by the Display thread. Additionally, a graphical display program written in Java was developed. Figure 7.16 shows an example of display output for the TMR configuration. In the left part of the screen four airplanes are represented. Planes positions received from radar are plotted as small circles, while filtered positions are plotted as squares. A line associated with each filtered position indicates the plane's course (line direction) and speed (line size). The current values of course and speed are displayed on the right of each plane's position, as well as an identification number. On the right side of the screen several data are presented, as the IP numbers of nodes sending unvoted data (for TMR only) and result data. A table containing information about all planes (course, speed, bearing and distance) is also presented.

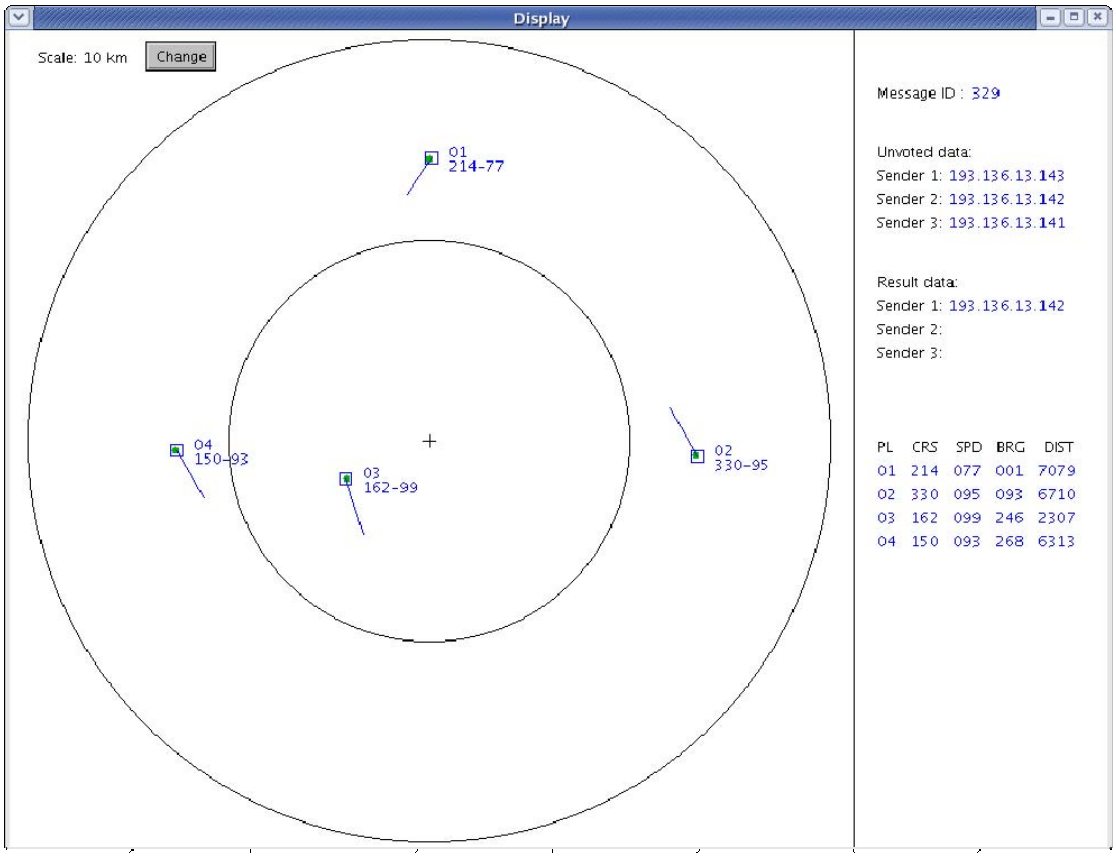

Figure 7.16: Case study II - display output example.

7.3.2 Fault tolerance implementations and testing

For the PSP configuration in Figure 7.14, both Filter threads receive the radar data and execute the computation, but only the primary thread sends its results. In the TMR configuration of Figure 7.15, all Filter threads send their results with "unvoted_data" as subject, which are received by the voter threads. In this particular configuration, coordinated voting is used and so only the master voter thread sends the final results to the Display thread. Status and coordination messages exchanged among FT threads and voters are sent with "FTStatus" as subject, as shown in Figure 7.14 (omitted in Figure 7.15).

In FT configurations, hardware faults were simulated by turning PowerPC boards off and software faults were simulated by introducing value errors in the filter calculation. In the PSP configuration, a hardware fault in a board running as primary causes a switch to primary in the other node. A software fault is detected by the acceptance test, and a rollback and retry is performed with the same algorithm. If the simulated fault is still present, the PSP thread will restart. For the TMR configuration, a hardware fault in the board with the master voter will imply in a new master voter board after the next master election. A software fault in one of the boards will be masked by the voter mechanism.

If a board is initialized, or if an FT thread is restarted, a state initialization is needed, as the filter output depends on the planes' last position and alpha-beta parameters. This initialization algorithm is performed by the corresponding *FTStrategy* object, transparently to the application program, which has only to define the *getState* and *setState* methods, as described in Section 5.3.2.

7.3.3 AOP implementations

The non-FT version of the Filter thread was modified using AOP to create the FT implementations using PSP and TMR. The definition on what version of the Filter thread (non-FT, PSP and TMR) will be applied is taken at compile time, using the same original non-FT version as the base code, and enabling the appropriate set of aspects, as described in Section 6.1.1.

AOP versions of the PSP and TMR filters have been tested in the same conditions as their respective plain object-oriented versions, performing identically and demonstrating the same functionality. A comparison of execution times between AOP and non-AOP versions led to equal outcomes.

7.3.4 Runtime costs

The radar simulation periodically sends planes' data every 2 seconds. This corresponds to the rotation period of the radar antenna. In order to test the system under more severe timing conditions and compare the runtime overhead of the test configurations the radar simulation period was reduced by factors of two. Figure 7.17 shows performance results in terms of CPU utilization for several configurations and simulation frequencies ranging from 0.5 Hz (2 seconds) to 32 Hz (31.25 ms).

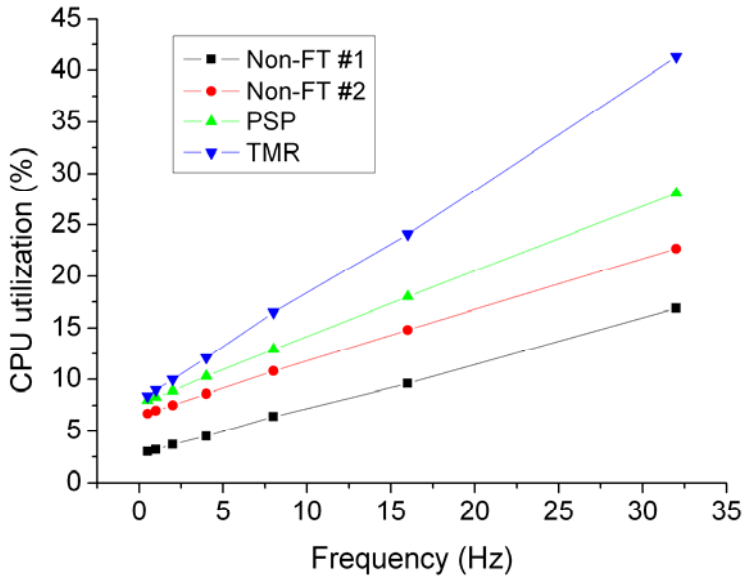

Figure 7.17: Case study II – CPU utilization results.

The curves labeled "Non-FT" are related to the non-fault tolerant single node version shown in Figure 7.13. The "Non-FT #1" version employed the original BOSS operating system with no FT framework, while in the "Non-FT #2" version the FT framework was integrated. We can notice that the utilization of the FT framework in

this case study implies a runtime cost ranging from 3.8 to 5.2%. These results are similar to the ones presented in Section 7.2.3 for case study I.

The PSP and TMR configurations resulted in higher runtime overhead than non-FT configurations as expected. The reason is the extra processing time associated with FT coordination, application-specific procedures and message communication. The TMR configuration achieved the worst results as it demands more threads for voting and more message exchanges.

7.4 Evaluation

This section evaluates the application of the FT framework, described in Chapter 5, and the AOP implementation, described in Section 6.1, based on the results obtained in case studies I and II.

7.4.1 FT framework

The utilization of the FT framework for application-level fault tolerance results in costs in time performance (execution time), runtime overhead (CPU utilization) and memory. In case study I, the execution time of non-FT configuration was compared to the execution time of several FT configurations (Section 7.2.2). The results presented in Figure 7.5 (no-failure condition) show that FT implementations have much longer execution times than their non-FT counterpart. For instance, the NVP implementation of the sorting application has a local execution time 5 times bigger. The execution time of FT configurations is affected by the coordination between the MS thread and the FT thread. Additionally, for the NVP strategy, the execution time is affected by the extra communication between NVP and voter threads.

In terms of runtime overhead, Figure 7.9 (case study I) and Figure 7.17 (case study II) show that difference of CPU utilization between non-FT and FT configurations depends linearly on the activation frequency. For low activation frequencies, the runtime overhead introduced by an FT configuration may have no significance while for high activation frequencies it may have a huge impact. The

utilization of FT scheduling also imposes a small runtime overhead, although not greater than 0.5% per FT thread.

Concerning memory costs, it was verified in case study I (Section 7.2.4) that the memory footprint of the FT version is about 15% bigger than the corresponding non-FT version. This difference is mostly due to the memory size of the FT framework rather than to the increase in memory size of the application.

We conclude that the performance penalties and resource costs of the proposed fault tolerance framework are still acceptable, considering the benefits in system dependability. However, for systems demanding very short execution times or already presenting a high CPU utilization or a reduced free memory, the introduction of fault tolerance might be a problem, and special care must be taken, including in the selection of the FT strategy.

7.4.2 Aspect-oriented implementation

The utilization of AOP for introducing fault tolerance at the application level does not increase the application execution time, as described in Section 7.3.3.

Regarding runtime overhead, the extra processing time related to the AOP implementation depends on the activation period of the FT thread. In Section 7.2.3, this overhead was measured for a high activation frequency (167 Hz – 6 ms) and it resulted in a 0.11% higher CPU utilization per FT thread for optimized programs. This overhead corresponds to an additional runtime of 6 microseconds for each FT thread activation.

The increase in memory footprint of the AOP implementation is very low. In case study I it consumed more 204 bytes of code and 16 bytes of unitialized data than the normal FT implementation, which correspond to less than 0.1% of the total memory footprint.

Based on these experiments we conclude that the utilization of AOP for application-level fault tolerance implementation does not imply a significant increase in runtime overhead or memory footprint.

7.5 Summary

This chapter presented two case studies designed to evaluate the fault tolerance introduction at the application level. The first case study was a sorting application using stateless multiple version fault tolerance. The second case study was a radar filtering system using single version fault tolerance and state threads.

Both non-fault-tolerant and fault-tolerant configurations were applied in the two case studies. Fault-tolerant configurations made use of several FT strategies, such as RB, DRB, PSP, TMR and NVP. Non-fault tolerant configurations employed operating systems versions with and without the proposed FT framework.

The performance in terms of execution time, plus the costs related to runtime overhead and memory footprint were measured for these configurations. The results show that the application of the FT framework causes significant costs, but those are still acceptable for embedded systems aiming high dependability. In contrast, the extra costs imposed by the AOP implementation proved to be negligible.

Chapter 8

Conclusions

This chapter sumarizes the objectives, contributions and conclusions of this thesis. It also proposes possible directions for future research.

8.1 Conclusions

The objective of this work is to provide fault-tolerance support for real-time embedded applications by integrating a fault tolerance framework into the operating system. Using this approach, the application software can be made fault-tolerant with a high degree of transparency regarding fault tolerance strategies and their associated mechanisms, such as state initialization and replica coordination. Special attention was taken to allow the coexistence of fault tolerance with real-time constraints, by providing an additional scheduling mechanism for FT threads.

The proposed fault tolerance framework employs the application thread as the unit of fault-tolerant computing. This solution uses a thread model which allows both state and stateless threads running in a distributed environment. Several FT strategies were implemented as RB, DRB and NVP. The inclusion of new FT strategies or the modification of the existing strategies can be easily performed by creating new *FTStrategy* classes or deriving classes from the existing ones.

As this work targets small-scale embedded systems, the proposed solution was tested using embedded PowerPC boards, similar to the previously used in the BIRD satellite. The resource costs in terms of execution time, runtime overhead and memory usage were measured and compared for several configurations in two case studies presented in Chapter 7. These case studies were selected to allow the application of a wide range of fault tolerance strategies using single and multiple version software. The results of these tests showed that this approach is feasible, but that the resource costs are significant, especially in terms of execution time and runtime overhead. However, these costs are considered acceptable for systems demanding high dependability.

The fault tolerance support described in this thesis presents several benefits. The main benefit is to simplify the application level programming because fault tolerance mechanisms are implemented at the operating system level. The application program merely has to define parameters and method implementations required by the chosen FT strategy. Other benefits include easiness of configuration and high flexibility both at compile and runtime.

In addition to the proposal and evaluation of a FT framework integrated into a real-time operating system, this work also evaluated the application of aspect-oriented techniques to the development of fault-tolerant software. In this work, AOP was applied for three different purposes: (1) integrate the FT framework into the operating system; (2) implement fault tolerance at the operating system level; and (3) modularize the fault tolerance code at the application level.

The integration of the FT framework into the operating system using AOP enables a complete separation of the FT framework from the OS code. It allows an optional integration of the framework into the operating system at weaving/compilation time. This modularization reflects in easier software maintenance and reduced memory footprint for non-FT applications.

The introduction of fault tolerance in the operating system using AOP adds fault tolerance error detection mechanisms. These mechanisms are implemented as executable assertions that verify predicates or invariants related to the OS basic functionality. This kind of FT functionality may be introduced selectively by aspects at weaving time.

The main target of AOP application was the introduction of fault tolerance at the application level. This approach was used to convey fault tolerance to existing applications without modifying their source code. The modularization of the fault tolerance code at the application level using AOP has several benefits. First, it reduces efforts and errors in making a legacy system fault-tolerant. It also simplifies system development by allowing the validation of the functional part in advance. Additionally, it facilitates the evaluation and comparison of various FT configurations, and contributes to product line development and code reuse. However, the availability of aspect-oriented weavers and tools for embedded systems development is very limited. The AspectC++ compiler used in this work is still in beta testing and has some restrictions as described in Section 6.1.2.

Regarding resource costs, implementations using application-level fault tolerance introduced by AOP were submitted to the same case studies described in Chapter 7. The results show that the extra costs imposed by AOP techniques are insignificant.

In summary, we conclude that the provision of operating system support to fault tolerance by means of an integrated FT framework is feasible and acceptable, bringing many benefits to the development of fault-tolerant embedded systems. Futhermore, our experiments indicate that the application of Aspect-Oriented Programming to introduce fault tolerance at the application level is advantageous and cost effective.

8.2 Future work

There are two possible diretions of future work regarding fault tolerance framework design: inwards or outwards the operating system. The inwards approach would be to promote a further integration between the fault tolerance framework and the operating sytem. An example of evolution regarding this approach is the modification of the operating system scheduler to include the assessment of FT thread deadlines. In the current implementation, this task is performed by the *MiddlewareScheduler* thread, and consists in a second scheduling algorithm. This work could improve the systems' real-time behavior and reduce the scheduling runtime overhead. However, as the interconnection between the OS and the FT framework increases, it would become harder to keep their development apart and just combine them, if needed, by applying AOP.

The second direction, the outward approach, would be to completely separate the FT framework from the OS. Using this approach, it could be designed a standard service interface between the OS layer and the FT framework, in order to facilitate the porting of the framework to other real-time operating systems. In this case, the operating system should be able to provide a minimum number of services to the FT framework, such as precise thread activation, thread priority management and basic communication mechanisms. The FT framework would have to implement the publisher-subscriber protocol to exchange FT related messages. This approach improves the framework portability but may have impact on real-time performance and resource costs.

Another possible future work is to include new fault tolerance strategies to the FT framework as, for instance, sequential NVP/TMR [7]. In addition, the fault tolerance

strategies currently implemented could be improved. The following improvements are suggested:

- The implementation of a mechanism for correcting state divergencies in FT threads running the NVP strategy.
- The implementation of a recovery cache for the RB and DRB strategies, using aspect-oriented techniques [10].
- The modification of the coordination behavior between the MS thread and the FT threads, in order to reduce the execution times. This can be performed by modifying the implementations of *FTStrategy* derived classes and does not depend on the *MiddlewareScheduler* class implementation.

Regarding the application of AOP in the fault tolerance domain, a possible research work is to execute more experiments with the introduction of fault tolerance at the operating system level. This work should include the definition of predicates for most operating system functionalities and the implementation of error detection mechanisms based on these predicates. The fault coverage of these mechanisms should be assessed using fault injection. It should also be evaluated if this approach is cost effective for embedded applications.

A future research may also include the application of AOP for middleware customization. In this work, the communication between the nodes employed UDP and broadcast. Other middleware versions could include point-to-point communication and different transport protocols. The configuration of what kind of middleware facility as well as other features such as fault tolerant communication can be defined selectively by aspects.

Further investigation on the combination of operating system object-oriented design, framework technologies and aspect-oriented techniques can lead to development of more customizable, evolvable and dependable embedded systems.

Bibliography

1. ACE. The Adaptive Communication Environment Douglas C. Schmidt's home page. http://www.cs.wustl.edu/~schmidt/ACE.html.

2. Afonso, F., Silva, C., Montenegro, S. and Tavares, A. Implementation of middleware fault tolerance support for real-time embedded applications. In *Proceedings of the Work-in-progress Session of the 18th Euromicro Conference on Real-Time Systems - ECRTS* (Dresden, Germany, 2006).

3. Afonso, F., Silva, C., Montenegro, S. and Tavares, A. Middleware Fault Tolerance Support for the BOSS Embedded Operating System. In *Proceeding of the International Workshop on Intelligent Solutions in Embedded Systems* (Vienna, Austria, 2006), 1-12.

4. Afonso, F., Silva, C., Montenegro, S. and Tavares, A. Applying aspects to a real-time embedded operating system. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software - ACP4IS* (Vancouver, British Columbia, Canada, 2007), ACM.

5. Afonso, F., Silva, C., Brito, N., Montenegro, S. and Tavares, A. Aspect-oriented fault tolerance for real-time embedded systems. In *Proceedings of the AOSD workshop on Aspects, components, and patterns for infrastructure software - ACP4IS* (Brussels, Belgium, 2008), ACM, 1-8.

6. Afonso, F., Silva, C., Tavares, A. and Montenegro, S. Application-level fault tolerance in real-time embedded systems. In *Proceeding of the International Symposium on Industrial Embedded Systems - SIES* (Montpelier, France, 2008), 126-133.

7. Aidemark, J., Folkesson, P. and Karlsson, J. A framework for node-level fault tolerance in distributed real-time systems. In *Proceedings of the International*

Conference on Dependable Systems and Networks (2005), IEEE Computer Society, 656-665.

8. Aksit, M., Wakita, K., Bosch, J., Bergmans, L. and Yonezawa, A. Abstracting Object Interactions Using Composition Filters. In *Proceedings of the Workshop on Object-Based Distributed Programming* (1994), Springer-Verlag, 142-184.

9. Alexandersson, R., Ohman, P. and Ivarsson, M. Aspect oriented software implemented node level fault tolerance. *Nineth IASTED International Conference on Software Engineering and Applications - SEA*, Phoenix, AZ, USA, 2005.

10. Alexandersson, R. and Ohman, P. Implementing Fault Tolerance Using Aspect Oriented Programming. In *Proceeding of the Third Latin American Symposium on Dependable Computing* (Morelia, Mexico, 2007), Springer-Verlag, 57-74.

11. Ammann, P.E. and Knight, J.C. Data diversity: an approach to software fault tolerance. *IEEE Transactions on Computers*, 37 (4): 418-425, 1988.

12. AOSD. Aspect Oriented and Fault Tolerance (aosd-discuss mailing list). http://aosd.net/pipermail/discuss_aosd.net/2004-May/000953.html.

13. Arlat, J., Fabre, J.C. and Rodriguez, M. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51 (2): 138-163, 2002.

14. AspectC++. User mailing list - Base class substitution, April 2007. http://p15111082.pureserver.info/pipermail/aspectc-user/2007-April/001146.html.

15. AspectC++. User mailing list - Base class substitution, January 2007. http://p15111082.pureserver.info/pipermail/aspectc-user/2007-January/001101.html.

16. AspectC++. http://www.aspectc.org/.

17. AspectJ. http://www.eclipse.org/aspectj/.

18. Athavale, A. Performance evaluation of hybrid voting schemes. M.S. Thesis. *Department of Computer Science*, North Carolina State University, 1990.

19. Avizienis, A. The Methodology of N-Version Programming. In Lyu, M.R. ed. *Software fault tolerance*, Wiley, 1995, 23-46.

20. Avizienis, A., Laprie, J.C., Randell, B. and Landwehr, C.A.L.C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1 (1): 11-33, 2004.

21. Barret, P.A. and Speirs, N.A. Towards an integrated approach to fault tolerance in Delta-4. In *Distributed Systems Engineering*, Institute of Physics Publishing, 1993, 59-66.

22. BeeSat. Technical University of Berlin. http://www.beesat.de.

23. Beuche, D., Guerrouat, A., Papajewski, H., Schroder-Preikschat, W., Spinczyk, O. and Spinczyk, U. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing - ISORC* (1999), 45-53.

24. Brie, K., Barwald, W., Gill, E., Kayal, H., Montenbruck, O., Montenegro, S., Halle, W., Skrbek, W., Studemund, H., Terzibaschian, T. and Venus, H. Technology demonstration by the BIRD-mission. *Acta Astronautica*, 56 (1-2): 57-63, 2005.

25. Briere, D. and Traverse, P. AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems. In *Proceedings of the Twenty-*

Third International Symposium on Fault-Tolerant Computing - FTCS-23 (Toulouse, France, 1993), 616-623.

26. Ceccato, M. and Tonella, P. Adding distribution to existing applications by means of aspect oriented programming. In *Proceedings of the fourth IEEE International Workshop on Source Code Analysis and Manipulation* (2004), 107-116.

27. Chen, L. and Avizienis, A. N-Version Programming: A fault-tolerance approach to reliability of sotware operation. In *Proceedings of FTCS-8* (Tolouse, France, 1978), 3-9.

28. Chiba, S. A metaobject protocol for C++. *ACM SIGPLAN Notices*, 30 (10): 285-299, 1995.

29. Coady, Y., Kiczales, G., Feeley, M. and Smolyn, G. Using aspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering Conference* (Vienna, Austria, 2001), ACM, 88 - 98.

30. Coady, Y. and Kiczales, G. Back to the future: a retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd international conference on aspect-oriented software development* (Boston, Massachusetts, 2003), ACM, 50-59.

31. Colyer, A., Clement, A., Bodkin, R. and Hugunin, J. Using AspectJ for component integration in middleware. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications* (Anaheim, CA, USA, 2003), ACM, 339 - 344.

32. Colyer, A. and Clement, A. Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on aspect-oriented software development* (Lancaster, UK, 2004), ACM, 56 - 65.

33. Connotech. Free Software C/C++ Cross-Compiler Suite for the Motorola MPC8xx. http://www.connotech.com/gcc_mpc8xx/powerpc_eabi_mpc850.htm.

34. Constantinides, C., Skotiniotis, T. and Stoerzer, M. AOP considered harmful. *First European Interactive Workshop on Aspect Systems - EIWAS*, 2004.

35. Daniels, F., Kim, K. and Vouk, M.A. The reliable hybrid pattern: a generalized software fault tolerant design pattern. In *Proceedings of PLOP conference* (Monticelo, Illinois, USA, 1997).

36. Dijkstra, E.W. On the role of scientific thought In *Selected writings on computing: a personal perspective*, Springer-Verlag New York, Inc., 1982, 60-66.

37. Dong, L., Melhem, R., Mosse, D., Ghosh, S., Heimerdinger, W. and Larson, A. Implementation of a transient-fault-tolerance scheme on DEOS. In *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium* (1999), IEEE Computer Society, 56-65.

38. DREAM. University of California, Irvine. http://dream.eng.uci.edu/.

39. ECOS. http://ecos.sourceware.org/.

40. Egan, A., Kutz, D., Mikulin, D., Melhem, R. and Moss, D. Fault-tolerant RT-Mach (FT-RT-Mach) and an application to real-time train control. *Software Practice and Experience*, 29 (4): 379-395, 1999.

41. ESRG. Embedded Systems Research Group, Department of Industrial Electronics, University of Minho. http://esrg.dei.uminho.pt/.

42. Filman, R.E. and Friedman, D.P. Aspect-Oriented Programming is Quantification and Obliviousness. *Technical report nº 46*, Research Institute for Advanced Computer Science (RIACS), 2000.

43. FIRST. Fraunhofer Institute for Computer Architecture and Software Technology. http://www.first.fhg.de/en/home.

44. FORTS. Fault Tolerant Real-Time Systems. University of Pittsburgh. http://www.cs.pitt.edu/FORTS/.

45. Gaisler, J. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN* (2002), 409-415.

46. Gal, A., Spinczyk, O. and Schroder-Preiskchat, W. On aspect-orientation in distributed real-time dependable systems. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems - WORDS* (2002), 261-267.

47. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

48. Gokhale, A., Natarajan, B., Schmidt, D.C. and Cross, K.C. Towards Real-Time Fault-Tolerant CORBA Middleware. *Cluster Computing*, 7 (4): 331-346, 2004.

49. Gray, J. Why do computers stop and what can be done about It? *Technical Report 85-7*, Tandem Computers, Cupertino, CA, 1985.

50. Gray, J. and Siewiorek, D.P. High-availability computer systems. *Computer*, 24 (9): 39-48, 1991.

51. Hecht, M., Agron, J., Hecht, H. and Kim, K.H. A distributed fault tolerant architecture for nuclear reactor and other critical process control applications. In *Proceedings of the 21st International Symposium of Fault-Tolerant Computing - FTCS-21* (1991), 462-498.

52. Hecht, M., Hecht, H. and Shokri, E. Adaptive fault tolerance for spacecraft. In *Proceedings of the IEEE Aerospace Conference* (2000), 521-533.

53. Herrero, J.L., Sanchez, F. and Toro, M. Fault tolerance as an aspect using JReplica. In *Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems - FTDCS* (2001), 201-207.

54. Hillman, R., Swift, G., Layton, P., Conrad, M.A.C.M., Thibodeau, C.A.T.C. and Irom, F.A.I.F. Space processor radiation mitigation and validation techniques for an 1,800 MIPS processor board. In *Proceedings of the 7th European Conference on Radiation and Its Effects on Components and Systems - RADECS* (2003), 347-352.

55. Horning, J.J., Lauer, H.C., Melliar-Smith, P.M. and Randell, B. A program structure for error detection and recovery. In *Proceedings of an International Symposium Operating Systems* (1974), Springer-Verlag, 171 - 187

56. Hursch, W. and Lopes, C. Separation of concerns, College of Computer Science, Northeastern University, 1995.

57. Jalote, P. *Fault tolerance in distributed systems*. Prentice-Hall, Inc., 1994.

58. Kantz, H. and Koza, C. The ELEKTRA railway signalling system: field experience with an actively replicated system with diversity. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing - FTCS* (1995), 453-458.

59. Kaul, D. and Gokhale, A. Middleware specialization using aspect oriented programming. In *Proceedings of the 44th annual Southeast regional conference* (Melbourne, Florida, 2006), ACM, 319 - 324.

60. Kayal, H., Baumann, F., Briess, K. and Montenegro, S. BEESAT: A Pico Satellite for the On Orbit Verification of Micro Wheels. In *Proceeding of 3rd*

International Conference in Recent Advances in Space Technology - RAST (Istanbul, Turkey, 2007), 487-502.

61. Kenneth, P.B. *Building secure and reliable network applications*. Manning Publications Co., 1997.

62. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming* (1997), Springer-Verlag, 220--242.

63. Kienzle, J. and Guerraou, R. AOP: Does It Make Sense? The Case of Concurrency and Failures. In *Lecture Notes in Computer Science*, Springer-Verlag, 2002, 113-121.

64. Kim, K.H. and Welch, H.O. Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, 38 (5): 626-636, 1989.

65. Kim, K.H. and Min, B.J. Approaches to implementation of multiple DRB stations in tightly-coupled computer networks. In *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference - COMPSAC* (1991), 550-557.

66. Kim, K.H. The distributed recovery block scheme. In Lyu, M.R. ed. *Software Fault Tolerance*, Wiley, 1995, 189-209.

67. Kim, K.H. and Subbaraman, C. Fault-tolerant real-time objects. *Communications of the ACM*, 40 (1): 75-82, 1997.

68. Kim, K.H. ROAFTS: a middleware architecture for real-time object-oriented adaptive fault tolerance support. In *Proceedings of theThird IEEE International High-Assurance Systems Engineering Symposium* (1998), 50-57.

69. Kim, K.H., Ishida, M. and Juqiang, L. An efficient middleware architecture supporting time-triggered message-triggered objects and an NT-based implementation. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing - ISORC* (1999), 54-63.

70. Kim, K.H. Toward Integration Of Major Design Techniques For Real-Time Fault-Tolerant Computer Systems. *Journal of Integrated Design & Process Science* 6(1): 83-101, 2002.

71. Kopetz, H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

72. Lala, J.H. and Harper, R.E. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82 (1): 25-40, 1994.

73. Lohmann, D., Scheler, F., Schroder-Preikschat, W. and Spinczyk, O. PURE Embedded Operating Systems - CiAO. In *International Workshop on Operating System Platforms for Embedded Real-Time Applications - OSPERT* (Dresden, Germany 2006).

74. Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O. and Schroder-Preiskchat, W. A quantitative analysis of aspects in the eCos kernel. *SIGOPS Operating Systems Review*, 40 (4): 191-204, 2006.

75. Lohmann, D., Spinczyk, O. and Schroder-Preiskchat, W. Lean and Efficient System Software Product Lines - Where Aspects Beat Objects. *Transaction of Aspect-Oriented Software Development II (TAOSD)*, Springer LNCS (4242): 227-255, 2006.

76. Lohmann, D., Streicher, J., Spinczyk, O. and Schroder-Preikschat, W. Interrupt synchronization in the CiAO operating system: experiences from implementing low-level system policies by AOP. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software - ACP4IS* (Vancouver, British Columbia, Canada, 2007), ACM, Article No. 6.

77. Maes, P. Concepts and experiments in computational reflection. *ACM SIGPLAN Notice*, 22 (12): 147-155, 1987.

78. Magnusson, J. Set and Get in AspectC++. M.S. Thesis. *Department of Computer Science and Engineering*, Chalmers University of Tecnology, Goteborg, 2006.

79. Mahrenholz, D., Spinczyk, O., Gal, A. and Schroder-Preikschat, W. An Aspect-Oriented Implementation of Interrupt Syncronization in the PURE Operating System Family. In *Proceedings of the Fifth ECOOP Workshop on Object Orientation and Operating Systems* (Malaga, Spain, 2002), 49-54.

80. Mahrenholz, D., Spinczyk, O. and Schroder-Preikschat, W. Program instrumentation for debugging and monitoring with AspectC++. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing - ISORC* (2002), 249-256.

81. Montenegro, S. and Zolzky. BOSS/EVERCONTROL OS /Middleware Target Ultra High Dependability In *Proceedings of Data Systems In Aerospace - DASIA* (Edinburgh, Scotland, 2005).

82. Montenegro, S., Briess, K. and Kayal, H. Dependable Software (BOSS) for the BEESat Pico Sattelite. In *Proceedings of the Data Systems on Aerospace Conference* (Berlin, Germany, 2006).

83. Montenegro, S. and Dittrich, L. Architecture of the SSB Core Avionics System. *Data Systems in Aerospace - DASIA* Palma de Mallorca, 2008.

84. Muñoz, F., Barais, O. and Baudry, B. Vigilant usage of Aspects. *Workshop on Aspects, Dependencies, and Interactions at ECOOP 2007*, Berlin, Germany, 2007.

85. Narasimhan, P., Dumitra, T.A., Paulos, A.M., Pertet, S.M., Reverte, C.F., Slember, J.G. and Srivastava, D. MEAD: support for Real-Time Fault-

Tolerant CORBA. *Concurrency and Computation : Practice and Experience*, 17 (12): 1527-1545, 2005.

86. Natarajan, B., Gokhale, A., Yajnik, S. and Schmidt, D.C. DOORS: towards high-performance fault tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications - DOA* (2000), 39-48.

87. Nelson, V.P. Fault-tolerant computing: fundamental concepts. *Computer*, 23 (7): 19-25, 1990.

88. OMG. CORBA Core Specification version 3.0.3 chapter 23. http://www.omg.org/cgi-bin/doc?formal/2004-03-12.

89. OMG. Real-Time CORBA specification version 1.2 http://www.omg.org/cgi-bin/doc?formal/05-01-04.

90. OMG. Object Management Group. http://www.omg.org/.

91. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15 (12): 1053-1058, 1972.

92. Powell, D. Distributed fault tolerance - lessons learnt from Delta-4. In *Papers of the workshop on hardware and software architectures for fault tolerance* (Le Mont Saint Michel, France, 1994), Springer-Verlag, 199 - 217

93. Pradhan, D.K. (ed.), *Fault-tolerant computer system design*. Prentice-Hall, Inc., 1996.

94. Pullum, L.L. *Software fault tolerance techniques and implementation*. Artech House, Inc., 2001.

95. Pure-Systems. Pure::variants. http://www.pure-systems.com/.

96. Qing, L. and Caroline, Y. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.

97. Randel, B. and Jie, X. The evolution of the recovery block concept. In Lyu, M.R. ed. *Software fault tolerance*, Wiley, 1995, 1-21.

98. Randell, B. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California, 1975), ACM, 437 - 449

99. Ren, Y., Bakken, D.E., Courtney, T., Cukier, M., Karr, D.A., Rubel, P., Sabnis, C., Sanders, W.H., Schantz, R.E. and Seri, M. AQuA: an adaptive architecture that provides dependable distributed objects. *IEEE Transactions on Computers*, 52 (1): 31-50, 2003.

100. Salles, F., Rodriguez, M., Fabre, J.C. and Arlat, J. MetaKernels and fault containment wrappers. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing* (1999), 22-29.

101. Schneider, F. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22 (4): 299-319, 1990.

102. Scott, R.K., Gault, J.W. and McAllister, D.F. Fault-Tolerant Software Reliability Modeling. *IEEE Transactions on Software Engineering*, 13 (5): 582-592, 1987.

103. Shokri, E., Crane, P., Kim, K.H. and Subbaraman, C. Architecture of ROAFTS/Solaris: a Solaris-based middleware for real-time object-oriented adaptive fault tolerance support. In *Proceedings of the 22nd Annual InternationalComputer Software and Applications Conference - COMPSAC* (1998), 90-98.

104. Siewiorek, D.P. Architecture of fault-tolerant computers: an historical perspective. *Proceedings of the IEEE*, 79 (12): 1710-1734, 1991.

105. Spinczyk, O. and Lohmann, D. Using AOP to develop architectural-neutral operating system components. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop* (Leuven, Belgium, 2004), ACM, Article No. 34

106. Spinczyk, O. and Lohmann, D. The design and implementation of AspectC++. *Knowledge Based Systems*, 20 (7): 636-651, 2007.

107. Steimann, F. The paradoxical success of aspect-oriented programming. In *Proceedings of the OOPSLA Conference* (Portland, Oregon, USA, 2006), ACM, 481 - 497

108. Storey, N. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

109. Szentivanyi, D. and Nadjm-Tehrani, S. Aspects for improvement of performance in fault-tolerant software. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing* (2004), 283-291.

110. Tokuda, H., Nakajima, T. and Rao, P. Real-time Mach: towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop* (1990), 73-82.

111. Torres, W. Software Fault Tolerance: A Tutorial, NASA Langley Technical Report Server, 2000.

112. Tourwé, T., Brichau, J. and Gybels, K. On the existence of the AOSD-evolution paradox. *Workshop on Software Engineering Properties of Languages for Aspect Tecnologies - SPLAT- AOSD*, Boston, 2003.

113. TQ. TQ Components. http://www.tqc.de/.

114. TRESE group, U.o.T. Composition Filters implementation project. University of Twente. http://trese.cs.utwente.nl/.

115. Tso, K.S., Shokri, E.H., Tai, A.T. and Dziegiel Jr., R.J. A reuse framework for software fault tolerance. In *Proceedings of the 10th AIAA Computing in Aerospace Conference* (San Antonio, TX, 1995), 490-500.

116. TXL. http://www.txl.ca/.

117. Verissimo, P. and Rodrigues, L. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

118. Vitulli, R. and Montenegro, S. High performance ultra high dependable architecture for autonomous robotics in space. *Nineth ESA Workshop on Advanced Space Technologies for Robotics and Automation*, Noordwijk, The Netherlands, 2006.

119. Xu, J., Randell, B., Rubira-Calsavara, C.M.F. and Stroud, R.J.A.S.R.J. Toward an object-oriented approach to software fault tolerance. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems* (1994), 226-233.

120. Xu, J., Randel, B. and Zorzo, A.F. Implementing Software Fault Tolerance in C++ and OpenC++: an object-oriented and reflective approach. In *Proceedings of the International Workshop on Computer-Aided Design, Test and Evalution for Dependability* (Beijing, China, 1996), pp. 224-229.

121. Xu, J., Randell, B. and Romanovsky, A. A generic approach to structuring and implementing complex fault-tolerant software. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing - ISORC* (2002), 207-214.

122. XWeaver. http://www.xweaver.org/.

123. Yacoub. Yacoub Automation GmdH. http://www.yacoub.de/e_frame.htm.

124. Zhang, C. and Jacobsen, H.A. Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14 (11): 1058-1073, 2003.