Technical Report

Cooperative Framework for Open Real-Time Systems

Cláudio Maia

HURRAY-TR-110302

Version:

Date: 03-23-2011

Cooperative Framework for Open Real-Time Systems

Cláudio Maia

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: crrm@isep.ipp.pt

http://www.hurray.isep.ipp.pt

Abstract

Embedded devices are reaching a point where society does not notice its presence; however, if suddenly taken away, everyone would notice their absence. The new, small, embedded devices used in consumer electronics, telecommunication, industrial automation, or automotive systems are the reason for their massive spread.

Influenced by this growth and pervasiveness, the scientific community is faced with new challenges in several domains. Of these, important ones are the management of the quality of the provided services and the management of the underlying resources - both interconnected to solve the problem of optimal allocation of physical resources (namely CPU, memory and network as examples), whilst providing the best possible quality to users.

Although several models have been presented in literature, a recent proposal handles resource management by using coalitions of nodes in open real-time cooperative environments, as a solution to guarantee that the application's non-functional requirements are met, and to provide the best possible quality of service to users. This proposal, the CooperatES framework, provides better models and mechanisms to handle resource management in open real-time systems, allowing resource constrained devices to collectively execute services with their neighbours, in order to fulfil the complex Quality of Service constraints imposed by users and applications.

Within the context of the CooperatES framework, the work presented in this thesis evaluates the feasibility of the implementation of the framework's Quality of Service concept within current embedded Java platforms, and proposes a solution and architecture for a specific platform: the Android operating system. To this purpose, the work provides an evaluation of the suitability of Java solutions for real-time and embedded systems, an evaluation of the Android platform for open real-time systems, as well as discusses the required extensions to Android allowing it to be used within real-time system. Furthermore, this thesis presents a prototype implementation of the CooperatES framework within the Android platform, which allows determining the suitability of the proposed platform extensions for open real-time systems applications.

© IPP Hurray! Research Group

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Cooperative Framework for Open Real-Time Systems

Cláudio Roberto Ribeiro Maia

Dissertação para a obtenção do Grau de Mestre em
Engenharia Informática

Área de especialização em **Arquitectura, Sistemas e Redes**

Orientador
Professor Doutor Luís Miguel Nogueira

Júri
XXX
XXX
XXX
XXX
XXX
XXX
XXX
XXX

Porto, Fevereiro de 2011

ACKNOWLEDGEMENTS

Since this project started back in October 2009, many people contributed by sharing experiences, guidance, knowledge and wisdom. In order to show my respect and gratitude to them, I would like to take this opportunity to do it.

First of all, I would like to thank all the people at ISEP that in some way provided me with their support. More specifically, my school collegues for keeping me sane, the teachers for clarifying me with all sort of answers, the security guards for opening the doors during the night, the cleaning ladies for their warmful "Good mornings" and the remaining people to whom I have spoken and discussed about many different topics.

I would also like to thank CISTER Research Centre for the opportunity of working in the CooperatES (Cooperative Embedded Systems) project (PTDC/EIA/71624/2006), which was the main driver to the MSc work presented in this document.

Another thankful and warm appreciation goes to my co-workers at CISTER for sharing their insights and ideas about my work; for providing their experiences, different visions and cultural perspectives; for the party nights and long discussions about my "quadratic" vision of metal versus the remaining types of music.

I am greatful to my advisors for all their support, advises, paper reviews and for showing me the correct paths of research by clarifying my thinking on several matters. Namely, to Luís Miguel Pinho for the time spent with me, support, flexibility, long discussions about my research work and for reviewing this document. To Luís Miguel Nogueira, for all support, time and effort spent reviewing my ideas, my work and for reviewing this document. Luís Nogueira, I am missing Czech Republic's beers. I cannot change the paragraph without mentioning Luís Lino Ferreira to whom I am thankful for the trust and although not being my official supervisor also helped me with his experience and knowledge. Their friendship and professional collaboration meant a lot to me.

Finally, a special thanks to my family and friends for being there in the moments where the work was at the door ringing the bell but I was busy handling other matters. Seriously, if it were not those moments of relax and joy I would most certainly have lost my mental sanity.

RESUMO ALARGADO

Actualmente, os sistemas embebidos estão presentes em toda a parte. Embora grande parte da população que os utiliza não tenha a noção da sua presença, na realidade, se repentinamente estes sistemas deixassem de existir, a sociedade iria sentir a sua falta. A sua utilização massiva deve-se ao facto de estarem practicamente incorporados em quase os todos dispositivos electrónicos de consumo, telecomunicações, automação industrial e automóvel.

Influenciada por este crescimento, a comunidade científica foi confrontada com novos problemas distribuídos por vários domínios científicos, dos quais são destacados a gestão da qualidade de serviço e gestão de recursos - domínio encarregue de resolver problemas relacionados com a alocação óptima de recursos físicos, tais como rede, memória e CPU.

Existe na literatura um vasto conjunto de modelos que propõem soluções para vários problemas apresentados no contexto destes domínios científicos. No entanto, não é possível encontrar modelos que lidem com a gestão de recursos em ambientes de execução cooperativos e abertos com restrições temporais utilizando coligações entre diferentes nós, de forma a satisfazer os requisitos não funcionais das aplicações.

Devido ao facto de estes sistemas serem dinâmicos por natureza, apresentam a característica de não ser possível conhecer, *a priori*, a quantidade de recursos necessários que uma aplicação irá requerer do sistema no qual irá ser executada. Este conhecimento só é adquirido aquando da execução da aplicação.

De modo a garantir uma gestão eficiente dos recursos disponíveis, em sistemas que apresentam um grande dinamismo na execução de tarefas com e sem restrições temporais, é necessário garantir dois aspectos fundamentais. O primeiro está relacionado com a obtenção de garantias na execução de tarefas de tempo-real. Estas devem sempre ser executadas dentro da janela temporal requirida. O segundo aspecto refere a necessidade de garantir que todos os recursos necessários à execução das tarefas são fornecidos, com o objectivo de manter os níveis de performance quer das aplicações, quer do próprio sistema.

Tendo em conta os dois aspectos acima mencionados, o projecto CooperatES foi especificado com o objectivo de permitir a dispositivos com poucos recursos uma execução colectiva de serviços com os seus vizinhos, de modo a cumprir com as complexas restrições de qualidade de serviço impostas pelos utilizadores ou pelas aplicações.

Decorrendo no contexto do projecto CooperatES, o trabalho resultante desta tese tem como principal objectivo avaliar a practicabilidade dos conceitos principais propostos no âmbito do projecto. O trabalho em causa implicou a escolha e análise de uma plataforma, a análise de requisitos, a implementação e avaliação de uma *framework* que permite a execução cooperativa de aplicações e serviços que apresentem

requisitos de qualidade de serviço.

Do trabalho desenvolvido resultaram as seguintes contribuições:

- Análise das plataformas de código aberto que possam ser utilizadas na implementação dos conceitos relacionados com o projecto CooperatES;
- Critérios que influenciaram a escolha da plataforma Android e um estudo focado na análise da plataforma sob uma perspectiva de sistemas de tempo-real;
- Experiências na implementação dos conceitos do projecto na plataforma Android;
- Avaliação da practicabilidade dos conceitos propostos no projecto CooperatES;
- Proposta de extensões que permitam incorporar características de sistemas de tempo real abertos na plataforma Android.

Keywords: Sistemas abertos de tempo-real, Execução Cooperativa, Gestão de Recursos, Gestão da Qualidade de Serviço, Android

ABSTRACT

Embedded devices are reaching a point where society does not notice its presence; however, if suddenly taken away, everyone would notice their absence. The new, small, embedded devices used in consumer electronics, telecommunication, industrial automation, or automotive systems are the reason for their massive spread.

Influenced by this growth and pervasiveness, the scientific community is faced with new challenges in several domains. Of these, important ones are the management of the quality of the provided services and the management of the underlying resources - both interconnected to solve the problem of optimal allocation of physical resources (namely CPU, memory and network as examples), whilst providing the best possible quality to users.

Although several models have been presented in literature, a recent proposal handles resource management by using coalitions of nodes in open real-time cooperative environments, as a solution to guarantee that the application's non-functional requirements are met, and to provide the best possible quality of service to users. This proposal, the CooperatES framework, provides better models and mechanisms to handle resource management in open real-time systems, allowing resource constrained devices to collectively execute services with their neighbours, in order to fulfil the complex Quality of Service constraints imposed by users and applications.

Within the context of the CooperatES framework, the work presented in this thesis evaluates the feasibility of the implementation of the framework's Quality of Service concept within current embedded Java platforms, and proposes a solution and architecture for a specific platform: the Android operating system. To this purpose, the work provides an evaluation of the suitability of Java solutions for real-time and embedded systems, an evaluation of the Android platform for open real-time systems, as well as discusses the required extensions to Android allowing it to be used within real-time system. Furthermore, this thesis presents a prototype implementation of the CooperatES framework within the Android platform, which allows determining the suitability of the proposed platform extensions for open real-time systems applications.

Keywords: Open real-time systems, Cooperative execution, Resource management, Quality of service, Android OS

CONTENTS

Acknowledgements **iii**

Resumo Alargado **v**

Abstract **vii**

Acronyms **xiii**

1 Introduction **1**

1.1 Real-Time Systems 2
1.2 Open Real-Time Systems 2
1.3 Motivation and contributions 3
1.4 Institutional Support 4
1.5 Thesis Overview 4

2 Real-Time Systems **7**

2.1 Introduction 7
2.2 Real-Time Scheduling 8
2.3 Quality of Service and Resource Management 11
2.4 Real-Time Java 15
2.5 Summary 17

3 Cooperative Embedded Systems **19**

3.1 Introduction 19
3.2 The CooperatES Framework 20
3.3 Framework Behaviour 22
3.3.1 Expressing Quality of Service 22
3.3.2 Coalition Formation 24
3.4 Scenarios 25
3.4.1 Scenario I - Campus 25

3.4.2 Scenario II - Home . . . 27
3.5 Summary . . . 29

4 Platform Evaluation **31**
4.1 Introduction . . . 31
4.2 Sun Java Real-Time System . . . 32
4.3 Aonix Perc Ultra . . . 33
4.4 JamaicaVM . . . 33
4.5 simpleRTJ . . . 34
4.6 Ovm Java Virtual Machine . . . 34
4.7 jRate . . . 35
4.8 Evaluation results . . . 35
4.9 Android Platform . . . 37
4.10 Summary . . . 39

5 Evaluating and Extending Android for Open Real-Time Environments **41**
5.1 Introduction . . . 41
5.2 Dalvik Virtual Machine . . . 42
5.3 Linux Kernel . . . 44
5.4 Resource Management . . . 46
5.5 Extending Android for Real-Time Embedded Systems . . . 46
5.6 Evaluation . . . 50
5.7 Summary . . . 51

6 Prototype Implementation **53**
6.1 Introduction . . . 53
6.2 Reverse Engineering Android . . . 54
6.3 Component Integration . . . 55
6.4 Boot Time . . . 56
6.5 Handling QoS . . . 57
6.6 Framework's Workflow . . . 61
6.7 Summary . . . 61

7 Conclusion and Future Work **63**

LIST OF FIGURES

3.1 CooperatES Framework . . . 21
3.2 Campus/Building scenario . . . 26
3.3 Home scenario . . . 28

4.1 Android platform architecture . . . 38

5.1 Zygote Heap . . . 43
5.2 Red-Black Tree example . . . 45
5.3 Android full Real-Time . . . 47
5.4 Android Extended . . . 48
5.5 Android partly Real-Time . . . 49
5.6 Android with a Real-Time Hypervisor . . . 50

6.1 CooperatES Architecture . . . 55
6.2 CooperatES Framework Workflow . . . 62

ACRONYMS

API	Application Programming Interface
CBS	Constant Bandwidth Server
CFS	Completely Fair Scheduler
CSS	Capacity Sharing and Stealing
DQM	Dynamic QoS Manager
EDF	Earliest Deadline First
FPS	Fixed priority scheduling
GARA	General-Purpose Architecture for Reservation and Allocation
GC	Garbage Collector
IETF	Internet Engineering Task Force
IP	Internet Protocol
JIT	Just-in-Time
JNI	Java Native Interface
NDK	Native Development Kit
NIST	National Institute of Standards and Technology
OS	Operating System
Q-RAM	QoS-based Resource Allocation Model
QoS	Quality Of Service

RSVP	Resource ReSerVation Protocol
RTCP	Real-Time Transfer Control Protocol
RTP	Real-Time Transport Protocol
RTSJ	Real-Time Specification for Java
SDK	Software Development Kit
SLA	Service Level Agreement
SMP	Symmetric Multiprocessing
UML	Unified Modelling Language
VM	Virtual Machine
WORA	Write Once, Run Anywhere

CHAPTER 1

INTRODUCTION

A long time has elapsed since the common use of traditional monolithic computer systems, optimally dimensioned to be used by one person, and one activity, at a time. Nowadays, almost all computer systems support multiple users, with multiple processes running concurrently and in a distributed mode, with a fair trade-off between costs and the achieved level of performance. Examples of this evolution span from the Internet, the current main vehicle of communication, to desktop computers, and to mobile phones (which are no longer phones but powerful handheld computers).

In particular, the field of embedded computing devices, *i.e.* computers embedded within other systems and that are not perceived by the user as a "computer", is increasingly important and demanding. The range of applications running on top of these systems varies from the common web browser or media player to flight control or robotic applications. Due to the decrease of the microprocessors' costs and, at the same time, reduction in size, there has been an explosion of types of systems, with an expanding diversity, which can be seen in our daily life.

Nowadays, devices are becoming smaller but, still, each one is capable of running several concurrent (and parallel) applications. However, these applications require available resources to execute, *i.e.* CPU, memory or network, which must be correctly managed. Although most of the devices come with an increasing amount of resources, these become constrained due to the, also increasing, number of applications executing in the system and the increase of requirements influencing the way applications execute.

In particular, devices with constrained resources are changing the software paradigms and the characteristics of executing applications. Most of these current applications also require that timing and service quality characteristics are supported by the underlying embedded platform. According to [Burns and Wellings, 2009], the real-time and embedded systems field is expanding its realms and it has been estimated that 99% of the worldwide production of microprocessors is used in this field. Also [Stankovic, 1996], real-time and embedded systems are becoming important and pervasive in such a way that the majority of the world-wide infrastructural services depend on them.

It is in this context that this thesis evaluates and proposes specific platforms and appropriate solutions to support the execution of applications with timing requirements on distributed embedded platforms. This chapter presents the fundamental concepts that are used throughout this document, introduces the

problem the presented work aims to solve and its main contributions. Finally, the chapter provides an outline of the rest of the document.

1.1 Real-Time Systems

A real-time system is a system in which the correctness of the system not only depends on the results it provides but also on the time instant at which the results are produced [Stankovic, 1996]. The notion of time, in this type of systems, is relative, as the response time requirements may vary from application to application, but a late action might cause an unexpected behaviour that could lead to a system failure. For instance, if a task executing in an air flight control system does not actuate at the time in which it is expected it may cause the airplane to lose control and even crash.

The time instant at which a task is expected to provide its result is denominated the *deadline*. Therefore, the task's deadline is the maximum time allowed for a task to complete its execution; this notion is what mainly distinguishes real-time systems from other systems without any timing constraints. The term task is used to refer to a unit of work that is schedulable and then executable. Schedulable means that it can be assigned to the processor unit in a particular order to be executed, *i.e.* when the selected time slot becomes available.

Not all real-time tasks may cause critical system failures. For instance, in a multimedia system, if the system fails to display a certain amount of frames belonging to a movie scene at a requested frame rate, due to a task missing its deadline, the end-user may notice it. Although this scenario does not induce catastrophic consequences, the deadline miss may clearly cause performance degradation which may affect the user's perception of that particular movie scene.

In order to distinguish the above mentioned systems, the consequences of a deadline miss are used for classification purposes. Thus, if a real-time task missing a deadline causes a performance degradation on the system, without jeopardizing the system behaviour, that task is considered a *soft* real-time task. On the other hand, if a task missing a deadline may cause a catastrophic consequence, the task is considered to be a *hard* real-time task.

When considering a real-time system as a whole, there are several important aspects that should be taken into consideration in order to ensure the timeliness of all the tasks with timing requirements. In particular, the Operating System (OS) plays a major role in the management of all the concurrent activities running on a single or multiprocessor device, both taking care of task management, through the use of scheduling mechanisms that handle the priority of each task, and managing memory allocations, by taking into account the timing requirements of the tasks. When designing a real-time system, every detail must be carefully analysed in order to make it as deterministic and predictable as possible, both in terms of time and space.

1.2 Open Real-Time Systems

Real-time systems have evolved from traditionally closed to open dynamic environments. In the traditional approach, or classical as also referred, systems are designed based on offline analysis and worst case execution time assumptions, originating a static and highly predictable behaviour, which assures the deterministic guarantees, imposed by application requirements. Examples of these systems are, for

instance, airplane control applications in the aerospace domain, electronic braking systems in automotive systems, or robot control in industrial automation.

On the other hand, open real-time systems, dynamic and heterogeneous in nature, consider that the system characteristics may not be fully known at design time, and the possibility of independent applications with timing constraints to be executed dynamically, in a local or distributed environment, together with non real-time applications.

This new approach poses new challenges to the scientific community due to its particular characteristics [Stankovic, 1996], namely, applications are allowed to run in platforms where hardware behaviour is only known at runtime, and applications' resource and timing requirements are not known beforehand, thus it is not possible to perform *a priori* analysis of the applications' tasks behaviour.

In order to overcome the characteristics of open real-time systems, there is the need of providing limited guarantees that a certain amount of resources will be available to applications when they are executed [Brandt et al., 1998a]. The efficient management of the available resources is provided by Quality Of Service (QoS) mechanisms implemented in the underlying system, which are responsible for measuring the quality of the service being provided, either to a user or an application, as well as assuring the required level of satisfaction.

Therefore, the use of resource management and reservation techniques is the basis to meet the application demands and provide stable QoS characteristics in open real-time systems, where the applications use more optimistic estimations of resource needs due to the "relaxed" timing constraints inherent to the system.

1.3 Motivation and contributions

In the context of open real-time systems, a cooperative execution of QoS-aware applications among neighbour nodes is a promising solution. The CooperatES [CooperatES, 2010] (Cooperative Embedded Systems) project has recently tackled the challenge of fulfilling the complex QoS constraints imposed by users and applications competing for the limited resources of a single device.

By allowing resource constrained devices to collectively execute services with their neighbours, forming temporary coalitions of heterogeneous nodes, whenever a particular set of QoS requirements cannot be satisfyingly answered by a single node, it is possible to deal with the QoS requirements imposed by users or applications. Therefore, CooperatES builds upon the proposition that heterogeneous, dynamic and open real-time systems can benefit from a cooperative decentralised model supported by QoS optimisation algorithms and effective scheduling mechanisms [Nogueira and Pinho, 2009].

In the context of CooperatES, this thesis aims to analyse, implement and evaluate the CooperatES Quality of Service concepts in order to validate their feasibility on current embedded platforms. The implemented framework must fulfil the non-functional requirements of the applications, by dynamically creating and managing coalitions of heterogeneous nodes, which could not be guaranteed through individual service execution.

The work presented in this thesis focuses on the QoS and resource management innovations of CooperatES (there are other issues, such as for instance code migration or resource mapping, which are not in the scope of this thesis) being the main contributions the following:

- An analysis of the available open-source platforms that can be used to implement the framework's

concepts. The analysis may be focused either on the OS or middleware, such as a Virtual Machine (VM), environments;

- An analysis of the framework requirements for each one of the specified components;
- A feasibility study of the limitations of the chosen platform for the prototype implementation;
- The implementation of the framework prototype. This step involves the implementation of the necessary extensions to the platform and must include a QoS definition scheme, framework components and its integration with the OS kernel;
- The validation of the implementation and feasibility of the framework's concepts.

Importantly, the work described in this thesis resulted in the publication of a paper entitled *Evaluating Android OS for Embedded Real-time Systems* [Maia et al., 2010c], published in the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, a highly-focused top-quality venue, satellite to the Euromicro Conference on Real-Time Systems, and the paper *Experiences on the implementation of a cooperative embedded system framework: short paper* [Maia et al., 2010a] published at the main venue for Embedded and Real-Time Java: the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems.

1.4 Institutional Support

This research work was developed in the context of the CooperatES project (PTDC/EIA/71624/2006), developed in CISTER (Research Centre in Real-Time Computing Systems).

CISTER is a top-ranked Research Unit based at the School of Engineering (ISEP) of the Polytechnic Institute of Porto (IPP), Portugal. The research unit focuses its activity in the analysis, design and implementation of real-time and embedded computing systems. CISTER was, in the 2004 evaluation process, the only research unit in Portugal, in the areas of electrical engineering and computer engineering and science, to be awarded the top-level rank of Excellent. This excellent rating was confirmed in the last evaluation process (2007), in which only one other research unit in these areas received this rating. Since it was created, the unit has grown to become one of the leading European research units in the area, contributing with seminal research works in a number of subjects.

1.5 Thesis Overview

This thesis is organized as follows:

Chapter 1 presents the main objectives behind this research work. It also provides a description of the context, motivation, contributions and information about the institutional support.

Chapter 2 introduces the core aspects that characterize real-time systems. In particular, the chapter focuses on providing the state of the art of Quality of Service management and real-time scheduling. The evolution and current state of the Real-Time Specification for Java is also presented, in order to allow the reader to fully understand the work presented in the thesis.

Chapter 3 provides a brief description of the CooperatES framework, presenting the problem description and system model as well as the supporting middleware. In order to ease the understanding the framework's concepts, two scenarios of application are also provided.

Chapter 4 presents a brief description of the characteristics and features of the technologies that were evaluated as the underlying platform for the prototype implementation of CooperatES.

Chapter 5 provides an in-depth evaluation of the support that the Android platform provides to the incorporation of real-time features. Afterwards, the chapter proposes and discusses four approaches to add real-time behaviour into the Android platform.

Chapter 6 presents an overview of the prototype implementation that was developed on the Android platform to validate the feasibility of the CooperatES framework implementation in an embedded platform.

Finally, Chapter 7 concludes this thesis and presents the future work based on the limitations found and features that can be improved.

CHAPTER 2

REAL-TIME SYSTEMS

Quality Of Service (QoS) and resource management in general have an extreme importance in the field of real-time systems. Without managing these two concepts it would not be possible to obtain guarantees on task execution and neither to obtain the required determinism and predictability of real-time tasks and real-time systems in general.

In order to present these concepts, this chapter provides a brief state of the art of quality of service and resource management, highlighting the most important works found in the literature. A special focus is also given to real-time scheduling, and, finally, to the evolution of the real-time specification for Java. The goal is to provide the reader with the necessary background about what is being done in the fields of interest to the work presented in this thesis, allowing him to better understand the performed work.

2.1 Introduction

Real-time systems were originally developed to support safety critical systems or mission critical systems, used in the automotive, aeronautics, robotic applications or manufacturing industries. As presented in the previous chapter, these systems are characterised by their timing constraints where a single deadline miss may have catastrophic consequences.

Usually, these systems are designed under worst case execution time assumptions in order to guarantee the fulfilment of its timing constraints and, therefore, that no deadline miss will occur. However, real-time systems have evolved from the traditional enclosed system to open systems, due to both the evolution of computer technology in general, and the change of requirements in applications and users.

The latter comes mostly from the fact that embedded systems nowadays are capable of executing interactive and multimedia applications that are also characterised by having timing requirements. Furthermore, these systems are more dynamic and adaptive in nature, and now need to support applications joining and leaving the system in an ad-hoc manner. Although these timing requirements can be dealt in a different manner, as the importance of a deadline miss does not have catastrophic consequences, the underlying concepts and associated methodologies are seen as an extension of the classic real-time systems.

These new systems use quality of service management to guarantee that a Service Level Agreement

(SLA) is established between each of the applications and the underlying platform. With that agreement established, the applications are assured that the amount of required resources can be provided by the system. Obviously, since (due to the dynamic nature of the applications) it is not longer possible to predict in advance the exact amount of resources required by the applications, the system must manage resources considering average requirements, but assuring that applications will not suffer from unpredictable and varying delays.

Quality of service management has a direct relationship with resource management. On one hand, QoS management is responsible for providing to end-users the tools to specify the QoS requirements and the trade-off that occurs between the achieved quality and the used resources, when certain parameters are selected. On the other hand, resource management is responsible for accepting requests from the QoS manager and to verify if there are enough resources to execute the application, and then to manage resources dynamically in order to provide the needed guarantees of execution.

Although this chapter focuses on these two concepts of real-time systems theory, it is important to emphasize that real-time systems specification must consider the whole system, *i.e.* all of its layers and components, with each one presenting its own problems that must be taken into account.

2.2 Real-Time Scheduling

Scheduling algorithms have an extreme importance in the context of real-time systems, as they are responsible for assuring that a task will terminate its execution before the requested deadline. It is the fundamental concept in these systems, and understanding its associated problems and proposed solutions is of paramount importance.

It is important to note that, in order to guarantee the needed determinism and predictability in the system, different scheduling schemes may be used. According to [Burns and Wellings, 2009], a scheduling scheme provides the following two features:

- An algorithm for assigning the system resources, *i.e.* CPU, to tasks following a particular order;
- A method of predicting the worst case behaviour of the system and therefore its temporal behaviour.

Besides the features presented above, it is also part of the scheduling function to maximize the resource that it controls, *i.e.* CPU, and minimize each task's response time.

Furthermore, the scheduling algorithm is responsible for the restrictions applied to tasks and in fact affects their categorization, *i.e.* hard real-time task vs. soft real-time task. Tasks are characterised by attributes such as the period, which is the time at which each task is released and ready for execution; the computation time, which is the time that each task consumes from the CPU; and the deadline, which is the time instant in which a task is expected to produce its result.

Scheduling can be categorized as static or dynamic. Static algorithms base their decisions on the information available before task execution while dynamic algorithms take run-time information into account. Basically, in practice this categorization affects the way the tasks characteristics are handled and therefore, how the scheduling algorithm chooses the next task to be executed.

There are two classical algorithms associated to each category presented above. Fixed priority scheduling (FPS) [Liu and Layland, 1973] is a static algorithm that assigns priorities to tasks before

the system's run-time. During run-time, the task with the highest priority, from the set of ready tasks, will be scheduled for execution. Earliest Deadline First (EDF) [Liu and Layland, 1973], is a dynamic algorithm that selects at runtime the task which has the smallest deadline from the set of tasks in the waiting queue ready to execute. The deadline in this case changes during run-time. Both of these algorithms can be used to schedule hard real-time tasks.

For typical embedded systems, hard real-time is not required as soft-real time will suffice to handle the majority of the tasks. Nevertheless, with the advent of reservation-based scheduling approaches, it was introduced the possibility of providing acceptable response for soft-real time tasks while bounding interference of hard real-time tasks [Nogueira, 2009]. The separation is achieved by assigning a share of the CPU to hard real-time tasks, based on their worst case execution time, in order to guarantee enough resources for these tasks to run, while the remaining capacity can be utilised by soft real-time tasks. Due to the loose time constraints that soft real-time tasks present, their reserves are calculated based on average execution times and not on worst case execution times.

Using average execution times to schedule soft real-time tasks may cause some undesirable effects, such as a task that takes more time to execute than the requested average time. For these cases, the system should isolate the task and avoid that the overload affects the remaining tasks executing in the system. Thus, it is desirable to isolate soft real-time tasks, not compromising hard real-time tasks in the system. The opposite, when a task takes less time than the estimated average, may also affect the system, but in this case the remaining budget can be re-used by other tasks, especially the ones that have already taken all of its average reserves.

The real-time literature is rich in scheduling schemes, either fixed or dynamic. The next paragraphs present a brief overview of the most important.

Optimal fixed priority capacity reclaiming algorithms were proposed by [Davis et al., 1993, Lehoczky and Ramos-Thuel, 1992]. These algorithms minimise the response time of soft real-time tasks while guaranteeing that the deadlines of hard real-time tasks are met. [Lehoczky and Ramos-Thuel, 1992] relies on a pre-computed table that defines the residual capacity for hard real-time tasks, while [Davis et al., 1993] determines the amount of available residual capacity at runtime, but the overheads caused by this approach make it infeasible in practice [Davis, 1993].

Bernat and Burns proposed a capacity sharing protocol for optimising non-periodic soft real-time tasks responsiveness in a fixed priority environment. Each task is handled by a dedicated server that is capable of stealing capacities from other servers in case of overloads. Servers are logical entities that are provided with a specific amount of computing time (the budget) which is used by the task while executing. When the budget is exhausted the task is suspended, until its next period (when the budget is replenished).

This protocol was extended by the HisReWri algorithm [Bernat et al., 2004]. This algorithm allows a hard real-time task to give soft-real time tasks a part of its allocated capacity. The soft real-time tasks are allowed to execute under the remaining time of hard real-time tasks. In the end of execution, if there is residual capacity available, the tasks' budgets are replenished by the amount of residual capacities that were consumed.

For dynamic priorities, a well known scheme was proposed in [Abeni and Buttazzo, 1998]. This Constant Bandwidth Server (CBS) scheduler was designed to limit the effects of tasks' overruns by handling soft real-time requests with a variable or unknown execution time using the EDF [Liu and

Layland, 1973] scheduling policy. Tasks are isolated from each other based on the category to which they belong. Therefore, hard real-time tasks are isolated from soft real-time tasks which use a reservation mechanism that prevent them to execute after the total bandwidth is used. Each soft real-time task is scheduled using a deadline, calculated as a function of the reserved bandwidth and its actual needs. If the task requires more bandwidth to execute than the one requested, the deadline will be postponed so that its reserved bandwidth is not exceeded and to avoid affecting the remaining system tasks.

CBS has been extended by several resource management schemes in order to overcome its drawback of not being able to share the computational resources of a task that finishes execution before the requested bandwidth. These extensions, namely [Caccamo et al., 2000, 2005, Lin and Brandt, 2005, Lipari and Baruah, 2000, Marzario et al., 2004], proposed to solve this problem by sharing the amount left unused by the tasks that complete its execution earlier than the requested budget in order to improve system response times.

All of the presented scheduling schemes assume that all the tasks are independent. Nevertheless, there are tasks that share system resources and therefore introduce contention issues that must be handled either by programming languages or the operating system. Synchronization mechanisms such as semaphores or protected objects can be used for assuring the correctness of the applications, however, pre-emptive real-time systems present the particularity of suffering from priority inversion [Sha et al., 1990], caused by the scheduling pre-emption process.

Priority inversion happens when a higher priority task is blocked on a shared resource owned by a lower priority task. If a medium priority task arrives to the system, pre-empting the lower priority task, it will cause an unbounded blocking delay to the higher priority task. There are two well known protocols that were proposed to handle this issue, Priority Inheritance Protocol [Sha et al., 1990] and Priority Ceiling Protocol [Chen and Lin, 1990].

Priority Inheritance Protocol dynamically modifies the priority of the tasks that are accessing a critical session so that the task inside the critical section is assigned the priority of the higher priority task waiting for the resource.

Priority Ceiling Protocol extends the previous protocol as a way to prevent deadlocks and task blocking. For that, each resource is assigned a priority equal to the priority of the highest priority task that can access it. Whenever a task executes within the resource, it will execute with its ceiling priority. However, the protocol must know the priority of the tasks that will be scheduled by the system and the resources used by them *a priori*.

The CooperatES framework uses the Capacity Sharing and Stealing (CSS) scheduling scheme, proposed in [Nogueira and Pinho, 2007], in order to solve the above mentioned problems. CSS is a dynamic server-based scheduler that supports the coexistence of guaranteed (for hard-real time tasks) and non-guaranteed servers (to handle soft real-time tasks). CSS allows servers to share residual capacity, reserved but not used, and to steal capacity from inactive servers to schedule best-effort jobs. CSS follows the CBS approach by keeping hard real-time tasks isolated from soft real-time tasks, all scheduled with EDF, but with the major difference of applying hard reservations for task replenishment.

2.3 Quality of Service and Resource Management

During the last two decades, the domain of Quality Of Service (QoS) management was the target of many research works, particularly conducted within real-time and embedded systems. Nevertheless, QoS management and the techniques associated with it have originally been developed in the context of computer networks. At that time, the intended objective was to have the possibility of managing the network bandwidth utilization and packet losses [Brandt et al., 1998a].

With the evolution of real-time systems and their paradigms, QoS management has spread from the network component into a broader context, due to the requirements of achieving system flexibility and predictability, particularly in those systems where the amount of resources required by the applications is not possible to predict in advance.

Nowadays, general purpose operating systems include mechanisms that are designed to take advantage of the applications' average performance but, at the same time, present high variability regarding applications' execution times [Colin and Petters, 2003]. This same variability applies to data intensive systems.

In order to perform an efficient QoS management, different architectural layers of the system must be addressed, particularly when considering QoS management in end-to-end systems, distributed systems or, as in the case of this thesis, open real-time systems. Although it is often approached in terms of individual resources, in order to be effective it should span and integrate all the individual resources together as a whole to accomplish the overall goal of the system [Loyall and Schantz, 2008]. Handling QoS management individually may lead to resource bottlenecks or resource contention which can be avoided if a global system perspective is taken into account.

Following the categorization provided by [Loyall and Schantz, 2008], QoS management can be divided into four categories:

- End-to-end QoS management - The management of QoS is dealt from information sources to information consumers. The notion of flow is a good example of an abstraction characterized by the production, transmission and eventual consumption of a single media source [Aurrecoechea et al., 1998];
- Multilayer QoS management - The management of QoS is performed by mapping high-level goals into lower level policies that are enforced by the resource managers;
- Aggregate QoS management - The management of QoS involves the mediation and management of resources between competing applications and users;
- Dynamic QoS management -The management of QoS involves the maintenance, improvement or graceful degradation as a response to changes in resource availability happening in the environment.

Although the categorization embraces several methodologies and technologies, for the scope of this work it is only relevant to consider QoS management for real-time operating systems, middleware and networks.

Several mechanisms have been developed for Operating System (OS) in order to support the timing requirements of soft real-time tasks. As it was referred previously, soft real-time applications are

allowed to miss deadlines. The OS support generally relies on scheduling primitives that use the information provided by the applications concerning timing requirements. These timing requirements will be interpreted and mapped into resources by the OS. When the application requirements cannot be fulfilled, the OS tends to reject additional applications or stop the running tasks with lower priorities [Mercer et al., 1994].

Other approaches use thresholds to control resource availability allowing the user to select which application to terminate [Compton and Tennenhouse, 1994]. Nevertheless, none of these approaches support the principle of graceful degradation. Approaches like the ones proposed by [Fan, 1995, Jeffay and Bennett, 1995, Jones and Regehr, 1999, Nieh and Lam, 1997] rely on the applications themselves to adjust system requirements, based on application importance or specific resources requirements criteria. However, these approaches assume that each of the applications will behave properly and also that the system is always able to work with optimal allocations.

Several efforts are being made to integrate real-time characteristics into general purpose OS, being Linux [Torvalds, 2010] the main target for modifications due to its open-source license. In general, developers of these systems create extensions to the Linux kernel that add predictability and determinism to the standard Linux kernel. For instance, RTLinux [Wind River Systems, 2010] supports Ethernet-based real-time communication [Regehr, 2008] and a lightweight TCP/IP stack for embedded systems [Deng et al., 2008]; and Linux/RK [RTMACH, 2010] which supports several multimedia and time-critical applications based on time reservations [Kim and Rosing, 2008].

LitmusRT [The University of North Carolina at Chapel Hill, 2010] is an extension to the Linux kernel focusing on multiprocessor real-time scheduling and synchronization. Its primary purpose is to provide experimental results related to real-time systems research problems. It has been used to evaluate adaptive QoS management [Block et al., 2008] and resource synchronization [Brandenburg and Anderson, 2009].

In the past two years, several efforts are being conducted in order to incorporate a new scheduling policy, in the form of a new scheduling class, for the Linux kernel. SCHED_DEADLINE [Faggioli, 2010] is the name of this new scheduling class officially maintainable in the kernel mainline. This new class implements the EDF and CBS scheduling policies in order to allow tasks to be isolated from each other, as well as to specify their deadlines and bandwidth budgets. Currently, this class supports bandwidth management and multiprocessor scheduling.

An extension to SCHED_DEADLINE and CBS itself was proposed in [Kato et al., 2010], whose objective is to enhance the QoS of the overall system by handling multiple reservations performed by multiple applications. Furthermore, this extension also supports multiprocessor environments.

On the side of commercial real-time operating systems, both VxWorks [Wind River, 2010] and QNX [QNX Software Systems, 2010] are well known as major OSs in the field.

Quality of service management is also widely considered at the networking level. Several research works have been conducted, especially focusing on end-systems or end-to-end architectures with QoS support, at the link, network and transport layers.

Scheduling algorithms were designed to support packet handling providing specific quality levels (one example of such work is [Clark et al., 1992]), while resource reservation protocols, such as Resource ReSerVation Protocol (RSVP) [Zhang et al., 2002], were also designed to support end-to-end reservation for specific network sessions. Concerning RSVP, it is integrated in DiffServ and IntServ [Braden et al., 1994], both Internet Engineering Task Force (IETF) standards. These protocols support real-time and

non real-time services in Internet Protocol (IP) networks.

Other solution for IP networks, is the Real-Time Transport Protocol (RTP) [Schulzrinne et al., 2003] which is a transport protocol that handles real-time traffic flows in this type of networks. RTP can be complemented with Real-Time Transfer Control Protocol (RTCP), which is responsible for carrying control information about RTP sessions. Although RTP does not provide the ability of performing resource reservations, it can be integrated with reservation protocols such as RSVP.

Although low-level mechanisms are important to guarantee an efficient resource management (and the references provided above are a good example of this), QoS management must combine the user's requirements for a particular application with the state and applications executing in the system. These requirements may possibly change over time and, whenever this happens, the system must adapt itself to the changes. However, adaptation cannot be performed if resources are managed independently of each other, which, in case of happening, may provoke resource bottlenecks. Therefore, in order to avoid these situations, *i.e.* conflicting demands of resources, a QoS middleware is needed to holistically manage all the resources involved in QoS provision for users and applications.

For this purpose, Jensen et al. [Jensen et al., 1985] proposed a soft real-time scheduling mechanism based on applications benefits. When using this mechanism, each application specifies a benefit curve that indicates the relative benefit to be achieved by scheduling the application at various times with respect to its deadlines. The objective is to schedule applications in order to maximise the overall system's benefit. Although complex, this work led to the adoption in several works of utility functions that represent the degree of satisfaction achieved with service modifications.

By taking into account the user's perspective in QoS management, several works focused on adaptive QoS control later appeared [Lee et al., 1996, McCanne and Jacobson, 1995, Tokuda and Kitayama, 1994]. Other works [Staehli et al., 1995, Venkatasubramanian and Nahrstedt, 1997] proposed analytical models used for measuring the quality achieved by the applications. Although still system-oriented, the mechanisms presented in these works identify the QoS parameters concerned with the user's satisfaction, the required resources and the functions that contain the relationship between both user's and resources requirements.

An important work was performed in [Rajkumar et al., 1997], proposing a QoS-based Resource Allocation Model (Q-RAM) focused on the maximisation of multiple resources' usage in order to achieve optimal system performance. Q-RAM proposed static resource allocation algorithms and these were extended to support dynamic task traffic models [Hansen et al., 2001] and to handle non-monotonic quality dimensions [Ghosh et al., 2003]. A quality dimension is used to characterize activities that refer specifically to quality of service characteristics [Hutchison et al., 1994].

On the quality perspective of specifying the user's QoS preferences, [Jones et al., 1995] proposed a model, integrated into the Rialto OS [Jones et al., 1996], to resource management by considering the user's preferences and resource allocation planning and resource management policies. The model attempts to dynamically maximize the user's perceived utility rather than specific applications. It is important to note that the preferences are specified in a qualitative manner, which impairs the analysis of QoS measures.

Focusing more on the precise specification of QoS requirements, [Abdelzaher et al., 2000] proposed a QoS negotiation mechanism that ensures graceful degradation of the applications. This approach was applied in operating systems and middleware implementations [Abdelzaher and Shin, 1998, 1999], where

the graceful degradation happens in case of system overloads, failures or violation of pre-runtime assumptions. [Khan, 1998] followed a similar approach.

A slightly different approach was followed by [Fan, 1995] and [Brandt et al., 1998b]. In the first case, an architecture was proposed that used a QoS manager to manage resource allocations based on pre-negotiated ranges between the architecture and the applications. As for the second case, the Dynamic QoS Manager (DQM) was proposed as a mediation mechanism to handle resource allocations taking into account maximum processor usage and users' benefit as measures for QoS levels. DQM uses execution level information and the state of the system to determine the appropriate resource allocations for the applications.

Still focusing on architectures that implemented concepts of system adaptation, Hola-QoS [Valls et al., 2003] is a framework for managing QoS and resources. It follows a layered perspective where each of the layers is responsible for a particular aspect of QoS. Lower layers provide resource management operations while high level layers provide service configuration and quality control.

A General-Purpose Architecture for Reservation and Allocation (GARA) was proposed by [Foster et al., 2004]. This generic architecture was designed to handle resource reservation and allocation for specific specifications related to flows. Furthermore, the architecture is capable of performing online monitoring and control of individual and heterogeneous resources.

The framework presented by [Palopoli et al., 2005] introduces an architecture for adaptive management of multiple resources on a general purpose OS. The management is done recurring to feedback mechanisms that allow applications to share access to resources by specifying fractions of usage, which are then used by the architecture for adaptation purposes. This architecture is an extension of the one proposed in [Cucinotta et al., 2004].

From the end-to-end QoS perspective, several works have been proposed. From those, [Aurrecoechea et al., 1998] conducted a survey of QoS architectures. This work then proposes a generic framework for distributed multimedia systems focused on the end-to-end perspective. This generic framework is based on software engineering principles that also affect how QoS is handled in a system, from its overall perspective. Principles such as transparency, integration, separation and performance are presented, as well as how the specification of the QoS requirements should be performed. From the several concepts presented in the survey, some have an extreme importance in the QoS management field:

- Level of Service - Expresses a degree of certainty that the QoS levels required by an application will be fulfilled;

- QoS Mapping - The mechanism responsible for performing the automatic translation of the different representation of QoS into the amount of physical resources.

- Admission Control - It is the decision of accepting an application to execute based on the comparison of the requested values against the available system resources. It can be used to perform reservations if the decision is positive;

- QoS degradation - It is a direct response to the current system state. If the system fails to maintain the current QoS levels, the system can adapt itself by reducing (degrading) the quality levels of certain applications in order to be possible to execute them at reduced levels of service.

The survey also presents several architectures focused both on network and end-systems domains. Although important for the evolution of QoS systems and the field in general, those architectures are considered to be out of scope of this thesis and therefore are not covered here. The reader is referred to [Aurrecoechea et al., 1998] for additional information.

CooperatES is a QoS-aware architecture that relies on the concepts of computation offloading and anytime computation. These concepts are not explored in this thesis, but some basic notions are important for the reader in order to understand some aspects of the work.

Concerning computation offloading, several studies propose task partition and allocation schemes that allow the computation to be offloaded, either entirely or partially, from resource constrained devices to more powerful neighbours [Gu et al., 2004, Li et al., 2002, Wang and Li, 2004]. The conclusion from these works is that the efficiency of an application execution can be improved if the workload is carefully partitioned between a device and a neighbour. The goal is the reduction of the needed computation time and energy consumption and it is limited to partitions where exists one constrained device and a neighbour [Chen et al., 2004, Kremer et al., 2003, Li et al., 2001, Othman and Hailes, 1998, Rudenko et al., 1998].

As for the concept of anytime computation [Liu et al., 1994], it is motivated by the fact that for many real-time applications it is preferable to have approximate results, on timely basis, of a poorer but acceptable quality than having results with the desired quality but that appear too late. The approximate results can be often produced with less processing power than optimal results. Anytime computation produces the result through several iterations, each one producing a more approximate value. The objective is that the system may use this technique for those cases where the optimal result may have an unbounded time to produce.

2.4 Real-Time Java

Although several different languages and programming frameworks are used and considered in real-time systems, one of the main currently being addressed, and that is relevant to this thesis, is the real-time/embedded version of Java.

Java is considered a successful language, one of the main programming languages in many projects implemented nowadays. Much of this success is due to its advantages, from a software engineering perspective, over other programming languages and models, but also to its open source nature. However, its concurrency model and support for real-time systems is very limited.

Several efforts were made in the past to extend the language in order to make it more appropriate for real-time systems. From all of those efforts, two were the precursors of the current existent solutions: [Miyoshi et al., 1997] proposed the Real-Time Java Threads and [Nilsen, 1998] proposed the PERC real-time API which aimed at solving the drawbacks of the language concerning real-time systems, such as synchronization mechanisms and garbage collection.

The Real-Time Specification for Java (RTSJ) [Gosling and Bollella, 2000, RTSJ, 2010] started later from a proposal made by Sun Microsystems and IBM to a call conducted by the U.S. National Institute of Standards and Technology (NIST) and several other organizations and companies. The objective was to have an extension to the Java platform that was capable of integrating all the current real-time practices and advances in real-time research [Wellings and Burns, 2008].

According to [Wellings and Burns, 2008] the main guiding principles behind the RTSJ were the following:

- RTSJ must be backward compatible with the standard Java programs;
- Support the Write Once, Run Anywhere (WORA) principle, but not at the cost of predictability;
- The priority must be the predictable execution of real-time programs;
- RTSJ must not introduce syntactic extensions to the Java standard programming language.

The idea in that time (late 1990s) was to have an extension, in the form of an Application Programming Interface (API), which supported the timeliness required by real-time programs, with hard real-time and soft real-time constraints, and non real-time programs in uniprocessor systems.

In the following paragraphs the enhancements that were introduced by the RTSJ are described briefly. For a more detailed analysis of the RTSJ specification the reader is referred to [Gosling and Bollella, 2000, RTSJ, 2010, Wellings, 2004]

Due to the fact that Java relies on the operating system to perform scheduling operations, these are not guaranteed to execute in a predictive manner. Java only supports 10 priority levels, while operating systems may support a higher or lower number of priority levels which affects how the thread mapping is done, especially in more restrictive operating systems. Furthermore, Java threads are competing against the operating system threads for the resources.

The approach followed by the RTSJ is to incorporate the notion of a *schedulable* object (*i.e.* RealtimeThreads or AsynEventHandler) instead of assuming the concept of threads. These objects have as attributes the same attributes that characterize a real-time task, namely, period or deadline, release time and priority, among several others. As for the scheduling policy, RTSJ specifies a pre-emptive priority-based scheduler, with 28 priority levels supported, that guarantees that the schedulable object with the highest priority is the one that is running. This scheduler can handle periodic tasks and sporadic tasks, *i.e.* asynchronous event handlers.

Memory management may be considered as one of the most important aspects that need attention in embedded systems, where the amount of resources is very limited. Thus, how memory is allocated is of extreme importance in embedded systems. Standard Java uses dynamic memory management allocated in the heap and garbage collection to deallocate memory that it is not in use by any of the running programs. Furthermore, as the reader may know, memory management in standard Java is the cause of many problems and application delays.

Real-time systems require all the operations to be bounded in time, which means that deallocation operations performed by a garbage collector may impact the timing requirements imposed by real-time applications. In order to solve this problem, RTSJ handles memory management using a model based on the concept of memory regions. The following memory regions are considered by the specification:

- HeapMemory - Allows objects to allocate memory in the heap, as it happens in the standard Java;
- ImmortalMemory - This memory region allows that all the memory allocated within it is never deallocated throughout the program execution. It is shared among all the program threads and it is deallocated when the program terminates, thus is never subject to garbage collection;

- ScopedMemory - This region can be utilised in scoped execution of instructions. All of the objects allocated within the scope will be deallocated when the program leaves that specific scope. Thus, this region has a limited lifetime.

Besides the memory regions, RTSJ provides extensions that allow programs to direct access the memory through the use of special classes. Also it provides an object named NoHeapRealtimeThread, that is not allowed to allocate objects on the heap and therefore, is independent of the garbage collector. Furthermore, the RTSJ garbage collector has the ability of being pre-emptable, meaning that when a higher priority thread (which can only be of the class NoHeapRealtimeThread) arrives for execution, the garbage collector can be pre-empted, leaving the system in a safe state.

Regarding synchronization, standard Java suffers from the priority inversion problem as all mutual exclusion mechanisms. RTSJ requires the implementations to support the priority inheritance or priority ceiling protocols. An alternative mechanism to the implementation of the protocols is to use nonblocking communication, and for this purpose RTSJ specifies several classes.

Other important features that deserve to be mentioned are the following:

- Asynchronous Transfer of Control - RTSJ allows threads to interrupt other threads, *i.e.* a response to an event generated by any task or even hardware, leaving shared data in a consistent state;

- Resource Management has not yet been address, however the recommendation is to negotiate resources via the operating system API;

- High resolution timers and a real-time clock.

2.5 Summary

In the context of real-time systems there are several issues that need to be considered, in order to provide the timing guarantees that applications are expecting. This chapter, presented the key concepts for the development of real-time systems that can provide timing guarantees and predictability to applications.

The first concept presented was that of real-time scheduling, which is important as it provides the mechanisms that assign the applications' tasks to the CPU, a fundamental issue in real-time systems. Several mechanisms were presented targeting not only traditional scheduling schemes but also new schemes, using resource reservation, that are more appropriate to be used in open real-time systems.

Afterwards, Quality of Service and Resource management were introduced, which have an extreme relevance in open real-time systems as they are responsible for providing applications with the required levels of quality and resource reservation to achieve the applications requirements.

Finally, the chapter presented the Real-Time Specification for Java (RTSJ), not only because the trend is that future embedded systems will run in virtual machine environments, being the RTSJ an important example, but also because it was important for the development of this thesis.

CHAPTER 3

COOPERATIVE EMBEDDED SYSTEMS

As the complexity of embedded systems increases, multiple services and applications have to compete for the limited resources of a single device. This situation is particularly critical for small embedded devices used in consumer electronics, telecommunication, or automotive systems. In fact, managing resources in these environments is complex and the problem becomes even more complex when considering open real-time systems.

This chapter focuses on the complex demands imposed to open real-time systems. It provides an overview of the problem of resource management in these systems, as well as a framework that intends to solve the problem of resource management in constrained devices. Finally, the chapter presents possible scenarios where the framework may be applied.

3.1 Introduction

Due to the characteristics of the computational load present in open real-time systems, where applications may dynamically enter or leave the system, it is not possible to predict the amount of resources needed by applications beforehand. This amount can only be calculated during run-time, and even so it is only valid at the instant it is calculated. Nevertheless, the desired level of performance required by an application must be managed, as well as its timeliness, otherwise resource overloads may happen, causing the system to enter in an unpredictable state or suffer from uncontrolled performance degradation.

Open real-time systems are characterised by their dynamic nature, which requires them to support mechanisms to manage the Quality Of Service (QoS) requirements imposed by users and applications. These mechanisms are responsible for handling the service level negotiation, service admission control and resource allocations, according to the requested requirements imposed by the applications. These sub-processes generally entail a decision making process that is performed based in a multi-criteria analysis where the objective is to maximise the global performance of the system.

The decision on accepting an application for execution is made based on the result of an admission control process, which evaluates available resources, considering the application's average expected values for resources. If the evaluation result is successful, the application is accepted and allowed to execute, meaning that the system has enough resources to fulfil the application's QoS requirements. On

the other hand, if the evaluation result states that there are not enough resources, the application is not allowed to execute, as accepting it for execution would cause uncontrolled system degradation. Other approaches can be considered that accept the new application, with a controlled degradation of service to other applications, if the overall system utility is considered to be higher.

In order to manage these issues and provide a better response to applications, from a system's perspective, a cooperative framework was proposed in [Nogueira and Pinho, 2009]. The CooperatES framework's objective is to facilitate the cooperation among neighbour nodes when a particular set of QoS requirements cannot be satisfied by a single node. Thus, the framework addresses the problem of resource management in cooperative environments, where the amount of resources available is scarce and diverse, and the applications' QoS requirements impose a distributed execution instead of only a local execution, that would not respond successfully to the applications' requests.

It is important to note that the work presented in this thesis is focused on the Framework's specification of the QoS requirements and the formation of coalitions of nodes, in order to determine the feasibility of providing such concepts in a current embedded platform. The Framework itself provides much more issues (such as for instance code migration, resource mapping, ...), which are not in the scope of this thesis.

3.2 The CooperatES Framework

The CooperatES framework is primarily focused in open and dynamic environments where new applications, named *services*, can appear while others are being executed. The processing of those services has associated real-time execution constraints, and their execution can be performed by a coalition of neighbour nodes. Nodes may cooperate either because they cannot deal alone with the resource allocation demands imposed by users and services, or because they can reduce the associated cost of execution by working together.

The framework is composed by several components, as depicted in Figure 3.1. It is not the objective of this thesis to detail the framework model presented in the figure, as it has already been done in [Nogueira and Pinho, 2009]. Nevertheless, an overview is here presented for the reader to understand the Framework's main components and behaviour.

When a QoS-aware service arrives to the system, it requests execution to the framework through the *Application Interface*, thus providing explicit admission control, while abstracting the underlying middleware and operating system.

Users provide a single specification of their own range of QoS preferences for a complete service, ranging from a desired QoS level to the maximum tolerable service degradation, specified by a minimum acceptable QoS level, without having to understand the individual tasks that make up the service. It is assumed that a service can be executed at varying levels of QoS to achieve an efficient resource usage, which constantly adapts to the devices' specific constraints, nature of executing tasks and dynamically changing system conditions.

The service request will then be handled by the *QoS Provider*, which in turn is composed by the *Local Provider* and *Coalition Organiser* components. The *Local Provider* is responsible for determining if a local execution of the new service is possible within the user's accepted QoS range, by executing a local gradient descent QoS optimisation algorithm. The goal is to maximise the satisfaction of the

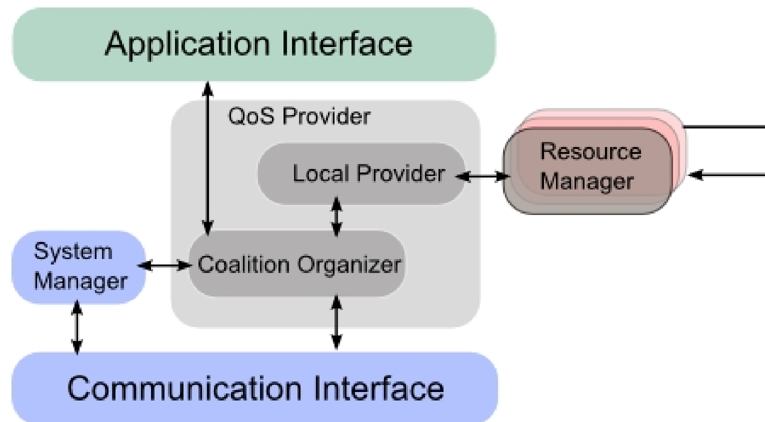

Figure 3.1: CooperatES Framework

new service's QoS constraints, while minimising the impact on the current QoS of previously accepted services [Nogueira and Pinho, 2009].

Rather than reserving local resources directly, it contacts the *Resource Managers* to grant the specific resource amounts requested by the service. Each *Resource Manager* is a module that manages a particular resource, and interfaces with the actual implementation in a particular system of the resource controller, such as the network's device driver, the CPU scheduler, or even with the module that manages other types of resources, *i.e.* memory.

The CooperatES framework differs from other QoS-aware frameworks by considering, due to the increasing size and complexity of distributed embedded real-time systems, the needed trade-off between the level of optimisation and the usefulness of an optimal runtime system's adaptation behaviour [Nogueira and Pinho, 2009]. Searching for an optimal resource allocation with respect to a particular goal has always been one of the fundamental problems in QoS management. However, as the complexity of open distributed systems increases, it is also increasingly difficult to achieve an optimal resource allocation that deals with both users' and nodes' constraints within a useful and bounded time. Note that if the system adapts too late to the new resource requirements, it may not be useful and may even be disadvantageous.

This idea has been formalised using the concepts of anytime QoS optimisation algorithms [Nogueira and Pinho, 2009], in which there are a range of acceptable solutions with varying qualities, adapting the distributed service allocation to the available deliberation time that is dynamically imposed as a result of emerging environmental conditions [Nogueira and Pinho, 2007].

If the resource demand imposed by the user's QoS preferences cannot be locally satisfied, the coalition formation process is started by the *Coalition Organiser*. This component is responsible for broadcasting the service's description, the user's quality preferences, evaluating the received service proposals, and deciding which nodes will be part of the coalition.

Thus, there will be a set of independent blocks to be collectively executed, resulting from partitioning a resource intensive service. Correct decisions on service partitioning must be made at runtime when sufficient information about workload and communication requirements become available [Wang and Li, 2004], since they may change with different execution instances and users' QoS preferences.

The *Coalition Organiser* interacts directly with the *System Manager* to know which nodes are able to participate in the coalition formation process. The *System Manager* is responsible for maintaining the

overall system configuration, detecting QoS-aware nodes entering and leaving the network, and managing the coalition's operation and dissolution.

3.3 Framework Behaviour

In order to address a cooperative execution, dynamic scenarios are considered the primary concern. In these scenarios, tasks may appear while others are being executed, and the computation of those tasks has real-time execution constraints. Furthermore, service execution may be performed by a coalition of neighbour nodes, as with these characteristics, resource availability is highly dynamic. The neighbour nodes may be heterogeneous regarding the architecture and specific set of resources as well as, constrained regarding the type and size of services they can execute.

For execution purposes, each service is partitioned into a set of independent tasks. This partitioning can only be made at runtime since the workload of the system may vary with different execution instances and users' QoS preferences. Furthermore, each service can be executed at varying levels of QoS to achieve an efficient resource usage that can adapt to system conditions. Nevertheless, the specification of the service's tasks that can be partitioned is part of the service description.

3.3.1 Expressing Quality of Service

In the context of distributed cooperative systems, QoS specification is an important issue due to the heterogeneity of the systems involved. The goal is to have a common understanding of the QoS specification in order to be possible for each of the systems involved in the cooperation process to understand and map the QoS requirements into system resources.

Thus, a generic QoS scheme must be provided in order to guarantee information consistency and compatibility. Consistency is achieved when all the nodes involved in the service execution understand the QoS requirements in the same manner. Compatibility is achieved when the same requirements have the same meaning for each of the nodes. Furthermore, this generic scheme must be flexible in order to provide the end-user with enough freedom to add or remove parameters, according to his needs.

In the CooperatES framework, the service's QoS requirements are specified in the form of parameters that are used by the system for adaptation purposes. Each subset of parameters that relates to a single aspect of service quality is called a QoS dimension. Each dimension has a direct relation to attributes and each attribute a direct relation to an associated value. The following structure is used to model the mentioned relations:

$$QoS = \{Dim, Attr, Val, DAr, AVr, Deps\}$$

, where

- Dim is the set of QoS dimensions;
- $Attr$ is the set of attributes that will map to one or more dimensions;
- Val is the set of attribute's values;
- DAr represents the set of relationships between dimensions and attributes;
- AVr represents the set of relationships between attributes and values;

- $Deps$ represents the set of dependencies between attributes.

Furthermore, each value has an associated type and domain. The attribute's type specifies the type of data that the system may expect when reading a particular value, *i.e.* integer, float or string; the attribute domain specifies the intervals and variations that the system may use, when performing the optimisation of the QoS levels, *i.e.* continuous or discrete.

In order to illustrate the proposed model, an example of a video streaming application is provided. In this example it is intended to demonstrate how the QoS requirements for such an application domain could be specified:

- Dim = {Video Quality, Audio Quality}
- Attr = {compression index, color depth, frame size, frame rate, sampling rate, sample bits}
- Val = {{1,integer,discrete},{3,integer,discrete}, ..., {[1,30],integer,continuous},...}

- DA Video Quality = {image quality, color depth, frame size, frame rate}
- DA Audio Quality = {sampling rate, sample bits}

- AV compression index = {[0,100]}
- AV frame size = {SQCIF,QCIF,CIF,4CIF,16CIF}
- AV color depth (bits) = {1,3,8,16,24,...}
- AV frame rate (per second) = {[1,30]}
- AV sampling rate (kHz) = {8,11,32,44,88}
- AV sample bits (bits) = {4,8,16,24}

By specifying and characterising a particular application domain, following the above format, user's and service providers are able to reach an agreement on service provisioning. Furthermore, with the requirements being specified in such a format, the system is also able to map them to resources and perform quality trade-offs, taking into account what is the best for the global system level, according to the user's request and the system's perspective.

Each user is allowed to provide their own range of QoS preferences for each service, ranging from the desired QoS level to the maximum tolerable service degradation, independently of the service internals. As a result, the user is able to express acceptable compromises in the desired QoS and assign utility values to QoS levels. One should note that the system will try to map the specified levels into resource allocations in a decoupled manner (note that QoS is not specified in terms of resource usage) and, according to the input QoS dimensions.

The user shall have an interface that allows him to understand the quality trade-offs between what he is selecting, in terms of QoS requirements, and the quality implications associated with each of the requirements. A typical example of the impact that different perspectives of quality may have for different

users is lip synchronisation. While listening to a music clip a user that prefers music quality instead of video quality, will require from the system the best audio quality the system can provide. On the other hand, another user may prefer to have better quality on video display instead of the audio itself. In this case, the audio rate would be a better candidate for degradation when the system resources are scarce.

In order to ease the life of the user, the user is able to specify the quality requirements based on a qualitative perspective and in a decreasing order of importance, as a way to remove the need of quantifying the quality trade-off with absolute values. The user is also able to provide an indication to the system of the level of importance of a quality dimension or even an attribute.

Choosing a remote video surveillance system as a different example to demonstrate this concept, it is clear that in this case video quality is more important than audio quality. Thus, the image's quality and the attributes associated with it, should be displayed with the highest quality possible than the audio playback. For this particular case, a user could specify its QoS constraints in the following manner:

- 1. Video Quality
- (a) compression index : {[0,20]}
- (a) frame rate : {[5,1]}
- (b) color depth : {3, 1}

- 2. Audio Quality
- (a) sampling rate : 11,8
- (b) sample bits : 8,4

3.3.2 Coalition Formation

For each service and given the range of user's QoS levels, the coalition formation process [Nogueira and Pinho, 2009] can be described as:

> Given a set of neighbour nodes, and a resource allocation demand enforced by the user's QoS preferences, if the resource demand cannot be satisfyingly answered by a single node, neighbour nodes should cooperate to fulfil such resource demand.
>
> The selection of a subset of nodes to cooperatively execute the service should be influenced by either the maximisation of the QoS constraints, associated with the service, and by the minimisation of the impact on the current QoS of the previously accepted services, caused by the arrival of the new service.

Instead of searching for an optimal resource allocation concerning a particular goal, it is proposed to establish an initial, sub-optimal, solution according to the set of QoS constraints that have to be satisfied. The initial solution is iteratively refined until it reaches its optimal value or the available (bounded) decision time expires.

At each iteration, a new solution is found with an increasing utility to the user's request under negotiation, but these successive adjustments get smaller as the QoS optimisation process progresses. Note

that if the system adapts too late to the new resource requirements, it may not be useful and may even be disadvantageous for the end-user.

In order to respond to a cooperation request, the nodes must have the same service's code blocks as the node performing the cooperation request. Each node must be capable of sharing resources with other nodes and executing tasks in an autonomous manner. Furthermore, a service can be executed by a single node or a group of nodes, depending on the users' node available resources and the service quality requirements. It is important to note that the service is processed in a transparent way for the user in order to avoid that users know the exact distribution used in cooperation scenarios.

3.4 Scenarios

In order to illustrate the practical applications of a cooperative framework such as the CooperatES framework, and to provide the reader with a better understanding of its behaviour, two example scenario are provided.

3.4.1 Scenario I - Campus

Within this scenario Figure 3.2 a person carrying a mobile device, for instance a PDA, enters a campus (or a building) and needs to navigate in 3D the campus/building map, while listening to direction indications, and information (for instance turn indications or information of what office is in a particular room the person is looking at in the 3D map).

This service is provided by a campus-provided application, through the campus/building wireless network. This could also be on top of 3G communication. The system uses the network infrastructure for issues of communication and communication mobility, which are not considered part of the system. The application is built incorporating the QoS model presented in section 3.3 of this chapter.

When the person requests to execute the application (service) in the mobile device, the application contacts the system middleware, depicted in Figure 3.1 of section 3.2, executing in the mobile device, requesting the allocation of computing resources (amount of memory, processor cycles and network bandwidth) necessary for the execution of the application. However, the mobile device may not have the computing requirements necessary for this. Or, for instance, the 3D video could have a frame size larger than the display of the device, therefore it needs to be compressed, by a task with larger requirements of computing power.

In this case, the system middleware implements the overall model of section 3.2. First, it will contact neighbour computing devices (campus/building computers and servers), called nodes, to request them to collaborate in executing the application. In this request, there is a description of the application computing requirements, as shown in section 3.2, and the different tasks that can be executed by different nodes.

In this case, for instance, this service description (user selected from the available application range) could be:

- Quality Dimensions = {Video Quality, Audio Quality}
- Video Quality = {color depth, frame size, update rate}

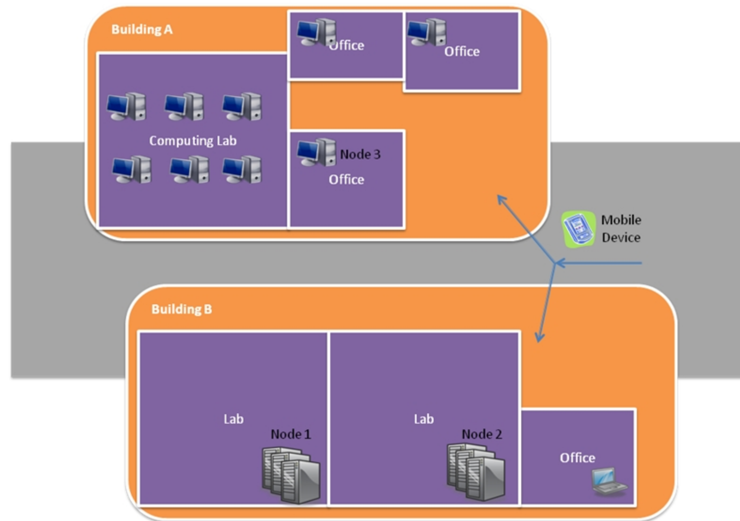

Figure 3.2: Campus/Building scenario

- Audio Quality = {sampling rate, sample bits}

- Frame size (pixels) = {80x40, 240x180, 320x240}
- Color depth (bits) = {8, 16, 24}
- Map update rate (updates per second) = {[15,30]}
- Sampling rate (kHz) = {32, 44, 88}
- Sample bits (bits) = {16, 24}
- Service tasks (called sub-services) = {Video compressing, Audio re-sampling}

Some of these nodes will have the system middleware installed and thus are prepared to collaborate. These nodes will reply to the mobile device with a proposal of the best service they can offer. They can propose to execute all, or only a subset of the tasks. The system middleware uses an anytime approach which means that the longer they calculate the proposal, a better proposal will be provided.

In order to assure the timeliness of the tasks, these neighbour nodes will use the underlying processor scheduling approach of Capacity Sharing and Stealing (CSS) [Nogueira and Pinho, 2007], in order to guarantee that the execution of the tasks does not negatively impact the applications already executing in them.

Example proposals could be:

Node 1:

- Proposes to execute = Video compressing and Audio re-sampling
- Frame size (pixels) = 240x180
- Color depth (bits) = 24

- Map update rate (updates per second) = 20
- Sampling rate (kHz) = 88
- Sample bits (bits) = 16

Node 2:

- Proposes to execute = Video compressing and Audio re-sampling
- Frame size (pixels) = 320x240
- Color depth (bits) = 24
- Map update rate (updates per second) = 30
- Sampling rate (kHz) = 44
- Sample bits (bits) = 16

Node 3:

- Proposes to execute = Audio re-sampling
- Sampling rate (kHz) = 88
- Sample bits (bits) = 24

After receiving the proposals from the neighbour nodes, the mobile device will choose the best set of nodes (coalition) to execute the service tasks. For instance, in this example, node 2 will execute the video compression and node 3 will execute the audio re-sampling. Therefore, the person will navigate the campus, following the 3D map, provided by node 3 and listening to the directions and information provided by node 3.

During this navigation, it can occur that one of these nodes will need to change the quality it is providing, because of the start or end of other applications it is executing. For instance, due to a higher load, node 2 may change the map update rate from 30 to 20 updates per second. This is still in the acceptable range of the application.

In this case, the system can be optimized by decreasing the amount of computation reserved in node 3 for audio (in order to maintain the same audio quality, node 3 will not need much processor capacity, memory and bandwidth because updates will be slower). Later on, node 2 can once again have more free resources (lower load), thus it can get back to the setting that provides a better quality to the applications it is executing.

3.4.2 Scenario II - Home

Within this scenario Figure 3.3, at home, a person watching a movie in the TV, intends to continue to see the movie in a different house room, using, for instance, a netbook. This service is provided by connecting to a Wi-Fi enabled house media centre. The video player application is built incorporating the QoS model presented in section 3.3.

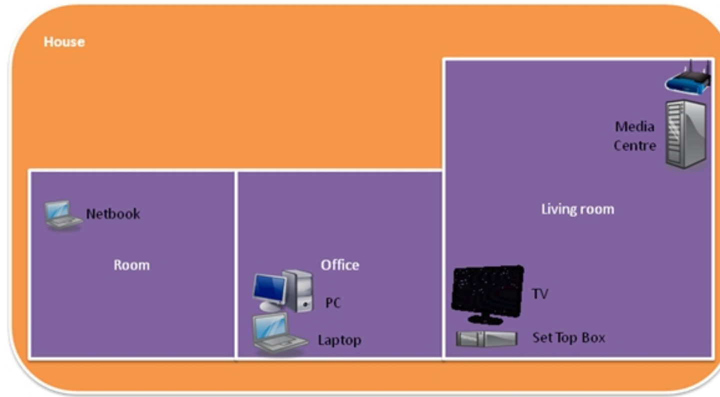

Figure 3.3: Home scenario

When the person requests to execute the video player application, in the netbook, it contacts the system middleware executing it in the device, requesting the allocation of computing resources (amount of memory, processor cycles and network bandwidth) necessary for the execution of the application. However, the device may not have the computing requirements necessary for this. For instance, the video is in high definition, and needs to be downgraded to a lower quality video.

In this case, the system middleware will contact computing devices in the house (PC, laptops, TV set top boxes, etc.), to request them to collaborate in executing the video application. In this request, there is a description of the computing requirements.

In this case, for instance, this service description (user selected from the available application range) could be:

- Quality Dimensions = {Video Quality}
- Video Quality = {color depth, frame size, frame rate}

- Frame size (pixels) = {640x480, 1024x768, 1280x800}
- Color depth (bits) = {16, 32}
- Frame rate (updates per second) = {[24,30]}
- Service tasks (called sub-services) = {Video downgrade}

Some of these nodes will have the system middleware installed and thus are prepared to collaborate, by replying to the netbook with a proposal of the best service they can offer. In this case there is only one task (sub-service to execute). These algorithms use a anytime approach which means that the longer they execute, a better proposal will be provided.

In order to execute these algorithms, these nodes will also use the underlying processor scheduling approach of CSS [Nogueira and Pinho, 2007] to guarantee that there is no negative impact in the applications already executing in them. For instance, the set top box needs to continue to provide the TV channel being seen in the TV with the same quality as before. These algorithms also guarantee that a more occupied node will have less time to execute therefore providing a lesser quality proposal. Therefore, load balancing is achieved.

Example proposals could be:

PC:

- Frame size (pixels) = 1024x768
- Color depth (bits) = 32
- Frame rate (updates per second) = 24

Laptop:

- Frame size (pixels) = 1024x768
- Color depth (bits) = 16
- Frame rate (updates per second) = 30

Media Centre:

- Does not reply. It is not able to provide a proposal.

Set top box:

- Frame size (pixels) = 1280x800
- Color depth (bits) = 32
- Frame rate (updates per second) = 30

After receiving the proposals from the nodes, the mobile device will choose the best set (in this case only one) to execute the service tasks: the set top box will execute the video downgrade.

During the movie, it can occur that the set top box will need to change the quality it is providing, because of other application requirements (e.g. it is no longer providing only a TV channel, but it is also recording). In this case, it could switch to a 1024x768 video frame size, which is still acceptable.

Later on, the set top box has once again lower load, thus it can give a better quality to the video. Note that, the user stability requirements are also taken into consideration. The user may decide that it is better to keep in a lower quality for more time, instead of frequent quality upgrades and downgrades.

3.5 Summary

Resource management is a complex problem to handle in embedded systems, where the amount of resources is limited. As an add-on to this complexity, resources must be shared by different types of applications presenting their own execution requirements. Although, local execution may be possible for some applications, others may not execute, due to the lack of available resources.

In order to solve this problem, the CooperatES framework was specified as a solution that takes advantage from QoS and resource management to solve the problem of resources' scarcity. The framework allows services to be executed in the neighbour nodes, in a cooperative fashion, by allowing services to form coalitions among those nodes and thus, providing the resources needed to applications.

This chapter presented the CooperatES framework and its importance in today's embedded systems. Two example scenarios were presented for a better understanding of the framework's behaviour.

CHAPTER 4

PLATFORM EVALUATION

The CooperatES framework proposes a new approach for the Quality Of Service (QoS) and resource management problems. This approach takes advantage of a distributed environment with a certain amount of heterogeneous nodes, which may cooperate between them, by creating coalitions of nodes, in order to fulfil the quality requirements imposed by applications and users.

Since there are no current platforms that support the framework's concepts out-of-the-box, a suitable platform should be chosen to evaluate the framework's feasibility. Therefore, this document presents an analysis of several platforms that could fit the framework's needs or, at least, that could provide the foundations that could be reused throughout the prototype implementation phase.

This chapter presents some details about each of the studied platforms, finishing with an evaluation section where the reasons behind the selection of the platform are presented.

4.1 Introduction

Being CooperatES a novelty in its domain of knowledge, the choice of a possible platform to implement the framework's concepts presented several implications. These implications may particularly affect the introduction of other concepts of the framework, resulting of the continuation of the research activities.

Not only the target platform should incorporate the foundation concepts that are part of CooperatES, such as offering support for network primitives, especially wireless ones, but also be a platform that is actively maintained, with a reasonable community support, and well documented. Furthermore, it should follow software engineering principles in order to be possible to take advantage of component reuse (and also to extend it).

Considering only the above mentioned requirements, it would be possible to assume that several easy alternatives would exist, particularly considering the actual state of software engineering and platforms publicly available. Nevertheless, real-time systems introduce extra requirements, thus, the potential platforms had to be narrowed down by following different criteria than the general ones presented above.

The main objective behind this analysis was to evaluate existing off-the-shelf solutions that could be used as a foundation for the CooperatES framework's prototype.

A narrowing scope for platforms was also established as a decision was made to focus on Java-based platforms. Java offers certain features that make it a strong language and environment, as for instance, platform independence, portability, or the object oriented model, which are not often found in other platforms for embedded real-time systems. Furthermore, there is a current evaluation process taking place on the suitability of real-time and embedded Java, and the result of this analysis could be used as input to this evaluation. For the specific case of CooperatES, Java advantages would have to be combined with real-time characteristics due to the real-time constraints imposed by the framework itself.

Following the above requirements and in order to neither broaden nor tighten the number of platforms *a priori*, the following criteria were used for evaluation purposes:

- (i) suitable to use with the Real-Time Specification for Java (RTSJ) [RTSJ, 2010];
- (ii) an operating system appropriate for real-time/embedded systems;
- (iii) the type of software license associated to the solution. The openness of the solution was an important decision factor as it would affect the extensibility of the framework;
- (iv) market trends, current status of the platforms and future work;
- (v) mobility support due to the distributed nature of the framework demonstrated by the coalition process.

Two major conclusions were reached. The first was that most of the existent solutions have proprietary licenses, meaning that it is not possible to extend them in order to implement the adaptations needed by the framework. The second conclusion is that each different solution tackles a single purpose, *i.e.* embedded real-time solutions vs. mobile device specific solutions. Thus, if one of these solutions would be chosen, this would entail the integration of different solutions into one, in order to implement all the proposed requirements of the CooperatES framework, *i.e.* real-time features and coalitions being performed by multiple heterogeneous nodes, as examples.

In the next sections, the evaluated platforms are presented, in a concise manner but with enough information about their features, in order to be possible to understand what influenced the decision process.

4.2 Sun Java Real-Time System

The Sun Java Real-Time System (Java RTS) [Oracle Corporation, 2010] is the implementation of the RTSJ provided by Oracle (previously Sun). Oracle added the RTSJ extensions to Java Standard Edition in order to make the Java technology more predictable and deterministic and, thus, to enable the possibility of using it in mission-critical real-time applications in areas such as financial trading, telecommunications infrastructure and industrial automation.

Concerning the features offered, this implementation provides full compatibility with the Java Standard Edition, meaning that standard Java applications can be run in this implementation. Furthermore, it supports all the extensions specified in RTSJ 1.0.2 [RTSJ, 2010], RealTimeThreads, NoHeapRealtimeThread, the ScopedMemory and ImmortalMemory models, priority-based scheduling, real-time garbage collection and synchronization mechanisms that avoid priority inversion situations.

According to Oracle, Java RTS presents a maximum latency of 20 microseconds and a maximum jitter of 10 microseconds [Oracle Corporation, 2010].

Java RTS needs a real-time Operating System (OS) in order to be possible to achieve the low latencies typically required by real-time applications. However, as Java RTS depends much on the OS running underneath it, all of the offered features are limited to the support and the features offered by the OS itself.

The supported OSs are Solaris 10, SUSE Linux Enterprise Real Time 10 and Red Hat Enterprise MRG 1.1. In terms of architectures, Java RTS supports both SPARC and x86-based architectures. The minimum recommended for the architecture is a dual core or dual CPU system with 512 MegaBytes of memory.

Regarding its license, Oracle provides a ninety day evaluation license for evaluation purposes. As for the commercial license, the pricing starts at $6500 per socket or CPU and it can be used for development or internal deployment purposes. For commercial external and volume deployments the price needs to be negotiated.

4.3 Aonix Perc Ultra

Aonix Perc Ultra is the Virtual Machine (VM) provided by Atego [Atego, 2010]. The implementation provided is compatible with the Java Standard Edition and it specifically targets real-time and embedded systems.

The Aonix Perc Ultra VM supports RTSJ 1.1 with a patented garbage collection technology; offers a direct memory Application Programming Interface (API) which offers compiler-optimized access to buffers and memory-mapped I/O. Besides this features, it also supports ahead-of-time compilation and building Read Only Memory images (ROM).

Regarding the supported OSs, it supports real-time OSs such as RTEMS, QNX, VxWorks and non real-time OSs such as Linux (although in this case no real-time guarantees are provided). In terms of architectures, the VM can execute in ARMs, PowerPC, MIPS and x86.

Atego only offers commercial licenses or 30-day trial licences.

4.4 JamaicaVM

The Jamaica Virtual Machine (JamaicaVM) [Aicas, 2010] is an implementation of the Java VM specification provided by aicas GmbH. It enables the execution of applications written for the Java 6 Standard Edition and it has been design for real-time and embedded systems.

Among all the supported features, the most important in the scope of this analysis are the hard real-time support, the minimal footprint, the dynamic memory management, the native code support. This VM is certifiable for the DO-178C and IEC 61508 standards used in safety critical systems.

Concerning the RTSJ version supported, at the time of writing the version 1.0.2 was the one supported. The VM itself occupies less than 1 MegaByte of memory depending on the target platform, which means that typically small applications will fit into a few MegaBytes of memory as most of the space needed is due to the libraries or resources used by the application.

The JamaicaVM executes on non real-time operating systems such as Linux, Solaris and Windows;

and real-time operating systems such as RTEMS, QNX or Linux/RT, among others. Regarding the CPUs, it executes on ARM, x86, PowerPC or Sparc, among others.

In terms of licensing, the provider grants a non-exclusive license to install and use the VM for evaluation purposes.

4.5 simpleRTJ

The simpleRTJ [RTJComputing, 2010] is a Java VM that has been specifically designed for embedded and control systems with a small amount of system memory. It runs on 8, 16 and 32 bit embedded system microcontrollers with a required average of 18 to 24 KyloBytes of code memory to run. It supports linear memory addressing of up to 16 MegaBytes of Random Access Memory.

Among the core features offered, the following are of particular interest: memory allocation, heap management, multi-threading, exception handling and garbage collection. Furthermore, this VM does not need an underlying real-time OS to execute. Most of these features are configurable in order to be possible to adapt the VM to particular environments. Concerning performance, it has the advantage of not needing dynamic class loading due to the pre-linked staged performed during the compilation of the Java code.

In order to be possible to run this VM, a host capable of running Java is required together with the target processor development tools. The application code and the simpleRTJ code will be compiled and *.class* files will be generated and pre-linked. The result will be a Java application binary image that will be used by the target processor cross-compiler and linker to generate the executable that will execute on the target platform. The Java application files are linked on the target platform.

In terms of limitations, it does not support all Java standard libraries and objects as for instance *java.lang.Class* object. Java Native Interface is not supported either and the target architecture must provide a timer interrupt in order to be used by the VM for thread scheduling.

This VM targets devices used in factory automation, robotic controllers and other various consumer electronic devices as well as smart cards and smart card readers or writers.

As far as the author of this thesis knows, this VM does not comply with the RTSJ.

Regarding its license, an evaluation version of simpleRTJ is provided for private, educational and evaluation purposes. The same applies for the source code. For commercial purposes, a commercial license must be acquired.

4.6 Ovm Java Virtual Machine

Ovm Java VM [Purdue University, 2010] is a generic framework for building virtual machines with different features, created to allow rapid prototyping of new VM features and new implementation techniques. The entity responsible for this implementation is Purdue University.

Most of the OVM's code is written in Java and it uses ahead-of-time compilation to perform the analysis of the program and the VM itself in order to translate the entire code into C/C++. The generated code is then processed by the GCC compiler to generate the executable code to a specific architecture. Currently, the supported architectures are RTEMS/LEON and real-time Linux with x86 processors. Since May of 2008, Ovm supports native scheduling using *pthreads* which by itself enables Symmetric

Multiprocessing (SMP).

Regarding its real-time support, the current implementation is capable of executing production code compliant with version 1.0 of the RTSJ, in the following areas [Baker et al., 2006]:

- Real-time threads and priority scheduler, is the basic mechanism for scheduling defined by RTSJ;
- Priority inheritance monitors for supporting priority inheritance protocols;
- General asynchronous event handler used to handle system or application events;
- Memory management, Scoped Memory and NoHeapRealtimeThread objects are supported.

Several efforts are being made in order to include new features and improve older ones in Ovm. As such, now it includes a configurable real-time garbage collection framework named Minuteman [Kalibera et al., 2009], which supports different garbage collection features especially targeting real-time systems, as well as a new scheduling policy, the slack-based scheduling policy.

As a curiosity, Ovm was used in the DARPA PCES project where autonomous navigation capabilities were demonstrated on ScanEagle, an unmanned air vehicle. The experiences and results obtained from that work can be found in [Armbruster et al., 2007].

The Ovm framework and tools are released under the new BSD license.

4.7 jRate

jRate [Corsaro, 2010] is an open-source Java real-time extension to the GNU GCJ compiler front-end and runtime system. It provides ahead-of-time compilation in order to create real-time applications. The result of the compilation is a native code application executable integrated with the GCJ runtime that runs directly on top of the operating system. The main drawback from having the code compiled using ahead-of-time compilation is portability, meaning that applications need to be recompiled for each target platform of execution. This extension supports most of the features required by the RTSJ, such as real-time scheduling, real-time threads and garbage collection.

It is important to note that the developers of this real-time extension have stopped the development of new features in 2006.

4.8 Evaluation results

The platforms presented in the previous sections were analysed following the criteria presented in section 4.1. It should be now clear for the reader that CooperatES heavily relies under temporal constraints assumptions as well as QoS and resource management.

Mobility also plays an important role in the coalition process occurring among the participating nodes. Thus, this was the first criterion to be taken into account. All of the presented platforms do not provide mobility features by default, mainly because all platforms were designed for small and embedded systems that run in isolated environments.

Thus, in order to incorporate mobility, extensions would need to be integrated as well as support for mobility in the host devices. If this path was chosen, this would probably entail an undetermined number of integration issues as there would be many independent components working together. It is important

to note that the mobility extensions would interface with the VM and the VM itself would have to be integrated with the real-time operating system, which would probably cause higher latency responses.

Another disadvantage found in the analysis was the licensing and access to the source code. Concerning this, the only platforms that are open-source are jRate and Ovm. The others only provide evaluative versions with limited features in some cases, *i.e.* simpleRTJ, or limited time evaluation periods, *i.e.* JamaicaVM, Aonix Perc Ultra and Sun Java RTS. Furthermore, their source code cannot be freely downloaded from the Internet. For the purpose of CooperatES this was a major limitation as the implementation of the framework's concepts heavily relies on integration either at the VM layer or operating system kernel layer.

Finally, concerning the compliance with the RTSJ specification, all of the platforms are compliant with some version of the specification, with exception of simpleRTJ, which does not follow it, and jRate which is not totally compliant.

After analysing the available platforms the conclusion was that there was no appropriate solution for this specific purpose. The project required an embedded and real-time platform designed with mobility capabilities and with an open source model.

Still, most of the above platforms have advantages, being the most obvious the out-of-the-box support for real-time systems which would mean less implementation effort, as the support for real-time would only depend on the features provided by the underlying OS.

Nevertheless, outweighing the advantages and disadvantages, a decision was made to broaden the selection criteria and other platforms were assessed. After a brief analysis on the new mobile platforms becoming available in the market, Android [Android, 2010] was considered the best platform for evaluation due to its VM environment, open source nature and Linux environment.

One of the main criteria was definitely mobility. Android offers mobility out-of-the-box, meaning that the coalition process operations could be easily implemented. Although not in the focus of this thesis, the potential for code migration was also an important factor.

The second aspect, also very important, concerns the availability of the source-code. Android's source code is open-source, licensed under Apache Software License 2.0 for the majority of the software and GPLv2 license for the Linux kernel.

The main Android's limitation, concerning the framework's concepts, is that it does not offer real-time features. However, there are a few patches available (or even the new SCHED_DEADLINE policy [Faggioli, 2010]) that introduces support in Linux for real-time systems.

In practice, this would mean more implementation effort and due to the lack of documentation regarding the integration between the VM and the Linux kernel, hours and hours of reverse engineering. On the other hand, it has the advantage of running in heterogeneous systems, *i.e.* mobile devices and x86 architectures, which is useful to prove the heterogeneity of the CooperatES framework.

Another important aspect about Android that needs to be considered is that most of the VM's work is being delegated to the Linux kernel, which means that it is possible to apply real-time patches to the Linux kernel and then provide the needed real-time features to the VM environment.

Thus, after evaluating all of the platforms, including Android, the possible scenarios and CooperatES framework requirements, it was decided to use Android for implementing the CooperatES prototype. This was mostly a strategic decision due to the challenges involved in the implementation of the project's concepts as well as the use of the platform in future research activities. Therefore, the following section

provides a more detailed description of the Android platform.

4.9 Android Platform

Android [Android, 2010] is an open-source software architecture provided by the Open Handset Alliance [Open Handset Alliance, 2010], a group of 79 technology and mobile companies whose objective is to provide a mobile software stack, although being initially promoted (and still developed) by Google. It was made publicly available during the fall of 2008. Being considered a fairly new technology, due to the fact that it is still being substantially improved and upgraded either in terms of features or firmware, Android is gaining strength both in the mobile industry and in other industries with different hardware architectures (such as the ones presented in [Android-x86, 2010] and [Macario et al., 2009]). The increasing interest from the industry arises from two core aspects: its open-source nature and its architectural model.

Being an open-source project allows Android to be fully analysed and understood, which enables feature comprehension, bug fixing, further improvements regarding new functionalities and, finally, porting to new hardware. On the other hand, its Linux kernel-based architecture model also adds the use of Linux to the mobile industry, taking advantage of the knowledge and features offered by Linux. Both of these aspects make Android an appealing target to be used in other type of environments. Another aspect that is important to consider when using Android is its own VM environment - Dalvik VM. Android applications are Java-based and this factor entails the use of a VM environment, with both its advantages and known problems.

Android platform includes an operating system, middleware and out of the box applications that can be used by the end user. It comes with a Software Development Kit (SDK) and a Native Development Kit (NDK), which provide the tools and Application Programming Interfaces (APIs) needed by the developers, in order to develop new applications for the platform. The applications can be developed using the Java programming language, if the developer uses the SDK or, on as an alternative, the C/C++ programming languages in the case of using the NDK. In terms of features, Android incorporates the common features found nowadays in any mobile platform, such as: application framework reusing, integrated browser, optimised graphics, media support, network technologies, etc.

The Android architecture, depicted in Figure 4.1, is composed by the following layers: Applications, Application Framework, Libraries, Android Runtime and finally the Linux kernel.

Starting with the uppermost layer, the Applications layer provides the core set of applications that are commonly shipped out of the box with any mobile device, *i.e.* Browser, Contacts, Phone, among others.

The next layer, the Application Framework, provides the Android framework's APIs used by the applications to interact with the platform. Besides the APIs, there is a set of services that enable the access to the Android's core features such as graphical components, information exchange managers, event managers and activity managers, as examples.

Below the Application Framework layer, there is another layer containing two important components: the Libraries and the Android Runtime. The Libraries provide core features to the applications. Among all the libraries provided, the most important are *libc*, the standard C system library tuned for embedded Linux-based devices; the media libraries, which support playback and recording of several audio and video formats; graphics engines; fonts; a lightweight relational database engine and 3D libraries based

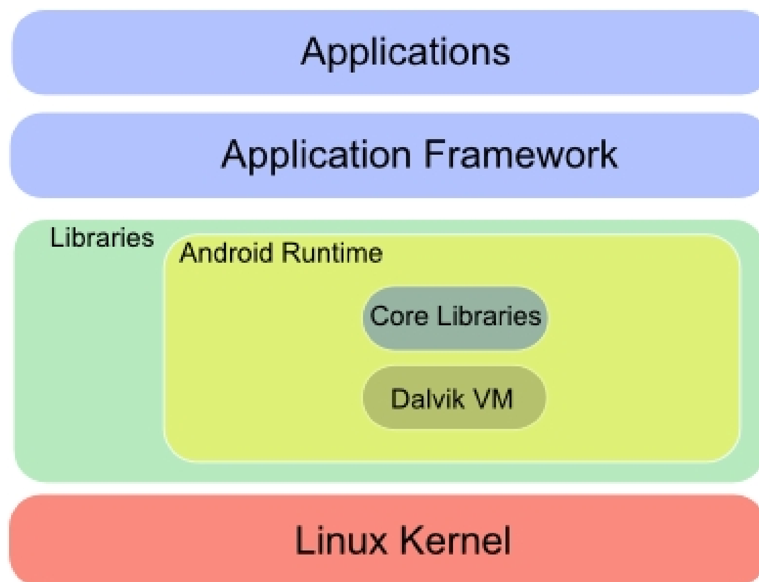

Figure 4.1: Android platform architecture

on OpenGL ES.

Concerning the Android Runtime, besides the internal core libraries, Android provides its own VM, as previously referred, which is named Dalvik [Bornstein, 2010]. Dalvik was designed from scratch and it is specifically targeted for memory-constrained and CPU-constrained devices. It runs Java applications on top of it and, unlike the standard Java VMs, which are stack-based, Dalvik is an infinite register-based machine. Being a register-machine, it presents two advantages when compared to stack-based machines. Namely, it requires 30% less instructions to perform the same computation as a typical stack machine, causing the reduction of instruction dispatch and memory access; and less computation time, which is also derived from the elimination of common expressions from the instructions. Nevertheless, Dalvik presents 35% more bytes in the instruction stream than a typical stack-machine. This drawback is compensated by the reading of two bytes at a time when decoding the instructions.

Dalvik uses its own byte-code format name Dalvik Executable (*.dex*), with the ability to include multiple classes in a single file. It is also able to perform several optimisations during *dex* generation concerning the internal storage of types and constants by using principles such as minimal repetition; per-type pools; and implicit labelling. By applying these principles, it is possible to have *dex* files smaller than a typical Java archive (*jar*) file. During install time, each *dex* file is verified and optimisations such as byte-swapping and padding, static-linking and method in-lining are performed in order to minimise the runtime evaluations and at the same time to avoid code security violations.

Dalvik maps the applications running on the uppermost layer to its own processes, each one with an instance of the virtual machine. This means that each application has its own address space, by running in a separate process. Dalvik is capable of executing multiple instances of the virtual machines and relies on the Linux kernel for features such as thread management and memory management.

The Linux kernel, version 2.6.29, is the bottommost layer and act as a hardware abstraction layer, which enables the interaction of the upper layers with the hardware via device drivers, also providing fundamental system services such as security, memory management, process management and the network stack.

4.10 Summary

Choosing a platform for the evaluation of the CooperatES framework is not an easy task as it may seem at the first glance. CooperatES presents unique requirements much because of its novelty in the field of real-time systems, particularly concerning QoS and resource management. As a framework that targets the system as a whole, the platform chosen for the development would need to provide this possibility as well as to satisfy the remaining requirements of the framework.

This chapter presented the features of several platforms that were analysed in order to find the most suitable to be used in the implementation of the CooperatES prototype. Although none of the initially considered platforms fit the requirements, after some deliberation and reasoning, Android was selected as the target platform. Besides providing the platforms features, this chapter also presents the ideas resulting from the deliberation process and the main ideas behind the selection of Android.

As the team did not have a deep knowledge about the Android platform, an in-depth evaluation was conducted concerning several aspects related to real-time systems. However, the study is mainly focused in Android's VM environment and Linux kernel. The results of this study are provided in the next chapter.

CHAPTER 5

EVALUATING AND EXTENDING ANDROID FOR OPEN REAL-TIME ENVIRONMENTS

The reasons for selecting Android as the platform to be used in the CooperatES prototype implementation were presented in the previous chapter. Nevertheless, an in-depth study of the current capabilities of the platform regarding several aspects of real-time systems and QoS support is needed. Therefore, this chapter discusses the suitability of Android for open embedded real-time systems, analyses its architecture internals and points out its current limitations. Android is evaluated considering the following topics: its Virtual Machine (VM) environment, the underlying Linux kernel, and its resource management capabilities. Furthermore, the chapter presents extensions to the Android platform in order to incorporate the real-time behaviour needed for CooperatES.

5.1 Introduction

Android is not a real-time platform neither was built with that purpose in mind. The companies involved in the project have the major goal of providing a mobile software stack that can fit the user's needs in the era of smartphones. Nevertheless, as Android is a Linux-based architecture, this fact was taken into consideration and was the main motivation behind the evaluation of its real-time capabilities.

Taking into consideration works made in the past such as [RTMACH, 2010] and [Corsaro, 2010], either concerning the Linux kernel or VM environments, there is the possibility of introducing temporal guarantees allied with Quality Of Service (QoS) guarantees in these layers (Operating System (OS) kernel and VM), in a way that a possible integration may be achieved, fulfilling the temporal constraints imposed by the applications.

An integration between the two layers, OS kernel and VM environment, may definitely bring advantages for applications that present soft real-time constraints or require resources to be reserved in advance, in order to execute properly. A good example of applications that fit these characteristics is

multimedia applications, which more often present real-time constraints in order to satisfy the user's needs. Thus, assuming that the Android platform may provide such real-time capabilities and resource optimisation in an out-of-box fashion, the end-users would largely benefit from this.

In the scope of the project, after selecting Android as the platform to be used in the prototype implementation, there was the need of evaluating its real-time capabilities in order to clearly understand its limitations and moreover to identify what modifications were needed in the platform in order to incorporate real-time behaviour and therefore, make it a better platform for the prototype implementation.

As a result of the conducted evaluation, the work presented in [Maia et al., 2010c] discusses the potential of Android and the implementation directions that can be adopted in order to make it possible to be used in Open Real-Time environments. It is important to note that most of the contents written in this chapter are taken from [Maia et al., 2010c] and [Maia et al., 2010a].

This chapter starts by presenting the characteristics of Android's, version 1.6, VM environment with a real-time system's perspective in mind. Following this section, the same work was applied to the Linux kernel and resource management mechanisms. Furthermore, possible extensions to the platform are presented taking into consideration the characteristics and limitations of the platform. These extensions provide the needed modifications to make Android a real-time platform and are provided with different perspectives in order to fit specific usage needs.

5.2 Dalvik Virtual Machine

Dalvik VM is capable of executing multiple independent processes, each one in a separate address space and memory. Therefore, each Android application is mapped to a Linux process and able to use an interprocess communication mechanism, based on Open-Binder [PalmSource Inc., 2010], to communicate with other processes in the system.

The ability of separating each process is provided by Android's architectural model. During the device's boot time, there is a process responsible for starting up the Android's runtime, the Zygote. This process is responsible for the pre-initialisation and pre-loading of the common Android's classes that will be used by most of the applications.

Afterwards, the Zygote opens a socket that accepts commands from the application framework whenever a new Android application is started. This will cause the Zygote to be forked and create a child process which will then become the target application. Zygote has its own heap and a set of libraries that are shared among all processes, whereas each process has its own set of libraries and classes that are independent from the other processes. This model is presented in Figure 5.1. The approach is beneficial for the system as, with it, it is possible to save RAM and to speed up each application startup process.

Android applications provide the common synchronisation mechanisms known to the Java community. Technically speaking, each VM instance has at least one main thread and may have several other threads executing concurrently. The threads belonging to the same VM instance may interact and synchronise with each other by the means of shared objects and monitors.

The API also allows the use of synchronised methods and the creation of thread groups in order to ease the manipulation of several thread operations. It is also possible to assign priorities to each thread. For example, when a programmer modifies the priority of a thread, with only 10 priority levels being permitted, the VM maps each of the values to Linux *nice* values, where lower values indicate a higher

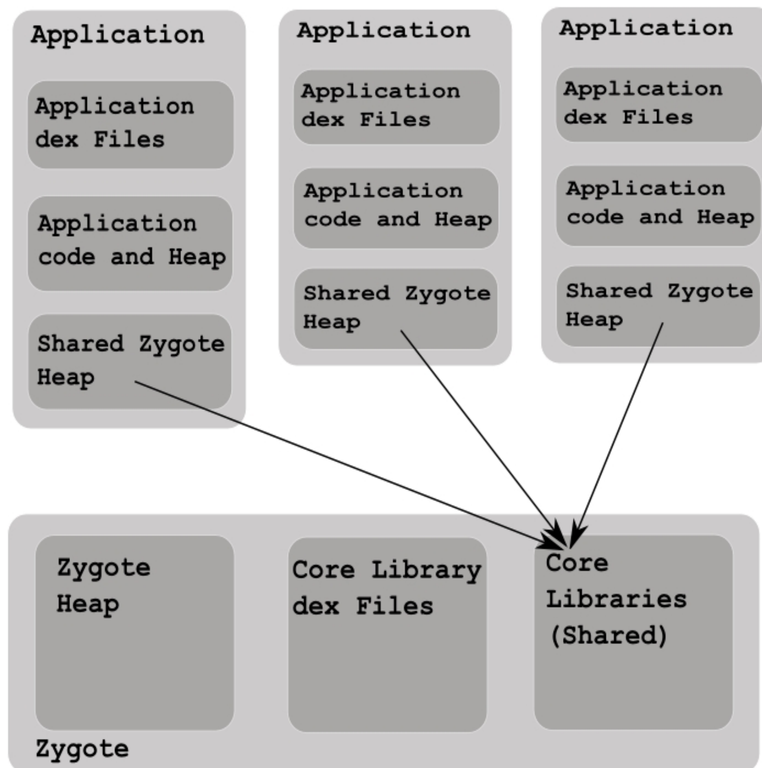

Figure 5.1: Zygote Heap

priority.

Dalvik follows the *pthread* model where all the threads are treated as native *pthreads*. Internal VM threads belong to one thread group and all other application threads belong to another group. According to source code analysis, Android does not provide any mechanisms to prevent priority inversion, neither allows threads to use Linux's real-time priorities within Dalvik.

Threads may suspend themselves or be suspended either by the Garbage Collector (GC), debugger or the signal monitor thread. The VM controls all the threads through the use of an internal structure where all the created threads are mapped. The GC will only execute when all the threads referring to each of the processes are suspended, in order to avoid inconsistent states.

The GCs have the difficult task of handling dynamic memory management, as they are responsible for deallocating the memory allocated to objects that are no longer needed by the applications. Concerning Android's garbage collection process, as the processes execute separately from other processes and each process has its own heap and a shared heap - the Zygote's heap -, Android executes separate instances of GCs in order to collect memory that is not being used anymore. Thus, each process heap is garbage collected independently, through the use of parallel mark bits that sign which objects shall be removed by the GC. This mechanism is particularly useful in Android due to the Zygote's shared heap, which in this case is kept untouched by the GC and allows a better use of the memory.

Android uses the mark-sweep algorithm to perform garbage collection. The main advantage provided by the platform is that there will be a GC executing per process, which wipes all the objects allocated from the application heap of a specific process. This way, GCs belonging to other processes will not impact the GC executing for a specific process. The main disadvantage arises from the algorithm used. As this algorithm implies the suspension of all the threads belonging to an application, this means that

no predictability can be achieved as that specific process will be frozen for an unbounded interval while being garbage collected.

Android's VM relies on the Linux kernel to perform all the scheduling operations. This means that all the threads executing on top of the VM will be, by default, scheduled with SCHED_OTHER, and as such will be translated into the fair scheme provided by the kernel. Therefore, it is not possible to indicate that a particular task needs to be scheduled using a different scheduling scheme.

Interrupt/event handling plays another important role when concerning real-time systems, as it may lead to inconsistent states if not handled properly. Currently, Android relies on the Linux kernel to dispatch the interrupt/event via device drivers. After an interrupt, the Java code responsible for the event handling will be notified in order to perform the respective operation. The communication path respects the architecture layers and inter-process communication may be used to notify the upper event handlers.

As of Android's version 1.6, Dalvik did not support Just-in-Time (JIT) compilation. The newer Android versions already support JIT compilation. Similarly, as of version 1.6, other features were also being considered as improvements, such as: a compact and more precise garbage collector and the use of ahead-of-time compilation for specific pieces of code. However, no further evaluations have been conducted since version 1.6.

5.3 Linux Kernel

The Linux kernel provides mechanisms that allow a programmer to take advantage of a basic pre-emptive fixed priority scheduling policy. However, when using this type of scheduling policy it is not possible to achieve real-time behaviour. Efforts have been made in the implementation of dynamic scheduling schemes which, instead of using fixed priorities for scheduling, use the concept of dynamic deadlines. These dynamic scheduling schemes have the advantage of achieving full CPU utilisation bound, but at the same time, they present an unpredictable behaviour when facing system overloads.

Since version 2.6.23, the standard Linux kernel uses the Completely Fair Scheduler (CFS), which applies fairness in the way that CPU time is assigned to tasks. This balance guarantees that all the tasks will have the same CPU share and that, each time that unfairness is verified, the algorithm assures that task re-balancing is performed. Although fairness is guaranteed, this algorithm does not provide any temporal guarantees to tasks, and therefore, neither Android does it, as its scheduling operations are delegated to the Linux kernel.

Android relies on the Linux kernel for features such as memory management, process management and security. As such, all the scheduling activities are delegated by the VM to the kernel.

Android uses the same scheduler as Linux. CFS has the objective of providing balance between tasks assigned to a processor. For that, it uses a red-black binary tree, as presented in Figure 5.2, with self-balancing capabilities, meaning that the longest path in the tree is no more than twice as long as the shortest path. Other important aspect is the efficiency of these types of trees, which present a complexity of $O(logn)$, where n represents the number of elements in the tree. As the tree is being used for scheduling purposes, the balance factor is the amount of time provided to a given task. This factor has been named virtual runtime. The higher the task's virtual runtime value, the lower is the need for the processor.

In terms of execution, the algorithm works as follows: the tasks with lower virtual runtime are placed

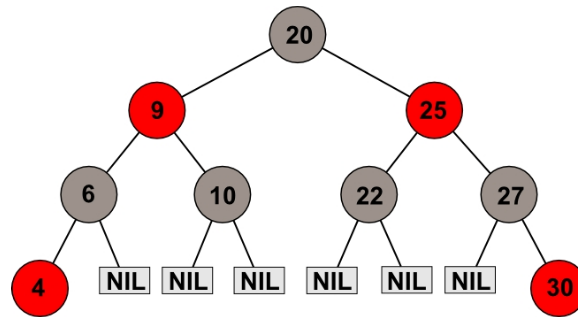

Figure 5.2: Red-Black Tree example

on the left side of the tree, and the tasks with the higher virtual runtime are placed on the right. This means that the tasks with the highest need for the processor will always be stored on the left side of the tree. Then, the scheduler picks the left-most node of the tree to be scheduled. Each task is responsible for accounting the CPU time taken during execution and adding this value to the previous virtual runtime value. Then, it is inserted back into the tree, if it has not finished yet. With this pattern of execution, it is guaranteed that the tasks contend the CPU time in a fair manner.

Another aspect of the fairness of the algorithm is the adjustments that it performs when the tasks are waiting for an I/O device. In this case, the tasks are compensated with the amount of time taken to receive the information they needed to complete its objective.

Since the introduction of the CFS, the concept of scheduling classes was also introduced. Basically, these classes provide the connection between the main generic scheduler functionalities and the specific scheduler classes that implement the scheduling algorithms. This concept allows several tasks to be scheduled differently by using different algorithms for this purpose.

Regarding the main scheduler, it is periodic and pre-emptive. Its periodicity is activated by the frequency of the CPU clock. It allows pre-emption either when a high priority task needs CPU time or when an interrupt exists. As for task priorities, these can be dynamically modified with the *nice* command and currently the kernel supports 140 priorities, where the values ranging from 0 to 99 are reserved for real-time processes and the values ranging from 100 to 139 are reserved for normal processes.

Currently, the Linux kernel supports two scheduling real-time classes, as part of the compliance with the POSIX standard [IEEE, 2010], SCHED_RR and SCHED_FIFO. SCHED_RR may be used for a round robin scheduling policy and SCHED_FIFO for a first-in, first-out policy. Both policies have a high impact on the system's performance if bad programming applies. However, most of the tasks are scheduled with SCHED_OTHER class, which is a non real-time policy.

Task scheduling plays one of the most important roles concerning the real-time features presented by a particular system. Currently, Linux's real-time implementation is limited to two scheduling real-time classes, both based on priority scheduling. Another important aspect to be considered in the evaluation is that most of the tasks are scheduled by CFS. Although CFS tries to optimise the time a task is waiting for CPU time, this effort is not enough as it is not capable of providing guaranteed response times.

One important aspect that should be remarked is that although the Linux kernel supports the real-time classes aforementioned, these classes are only available for native[1] Android applications. Normal Android applications can only take advantage of the synchronisation mechanisms described earlier in this paper.

[1]A native application in Android is an application that can execute on top of the Linux kernel without the need of the VM.

Regarding synchronisation, Android uses its own implementation of *libc* - named *bionic*. *bionic* has its own implementation of the *pthread* library and it does not support process-shared mutexes and condition variables. However, thread mutexes and thread condition variables are supported in a limited manner. Currently, inter-process communication is handled by Open-Binder. In terms of real-time limitations, the mechanisms provided by the architecture do not solve the old problems related with priority inversion. Therefore, synchronisation protocols such as priority ceiling and inheritance are not implemented.

In terms of interrupt/event handling, these are performed by the kernel via device drivers. Afterwards, the kernel is notified and then is responsible for notifying the application waiting for that specific interrupt/event. None of the parts involved in the handling has a notion of the time restrictions available to perform its operations. This behaviour becomes more serious when considering interrupts. In Linux the interrupts are the highest priority tasks, and therefore, this means that a high priority task can be interrupted by the arrival of an interrupt. This is considered a big drawback, as it is not possible to make the system totally predictable.

5.4 Resource Management

Resource management implies its accounting, reclamation, allocation, and negotiation [Higuera-Toledano and Issarny, 2000]. Concerning resource management conducted at the VM level, CPU time is controlled by the scheduling algorithms, whereas memory can be controlled either by the VM, if we consider the heaps and its memory management, or by the operating system kernel. Regarding memory, operations such as accounting, allocation and reallocation can be performed. All these operations suffer from an unbounded and non-deterministic behaviour, which means that it is not possible to define and measure the time allowed for these operations. The network is out of scope of our analysis and thus was not evaluated.

At the kernel level, with the exception of the CPU and memory, all the remaining system's hardware is accessed via device drivers, in order to perform its operations and control the resources' status.

Nevertheless, a global manager that has a complete knowledge of the applications' needs and system's status is missing. The arbitration of resources among applications requires proper control mechanisms if real-time guarantees are going to be provided. Each application has a resource demand associated to each quality level it can provide. However, under limited resources not all applications will be able to deliver their maximum quality level. As such, a global resource manager is able to allocate resources to competing applications so that a global optimisation goal of the system is achieved [Nogueira and Pinho, 2009].

5.5 Extending Android for Real-Time Embedded Systems

This section discusses four possible directions to incorporate the desired real-time behaviour into the Android architecture.

- The first approach considers the replacement of the Linux operating system by one that provides real-time features and, at the same time, it considers the inclusion of a real-time VM;
- The second approach respects the Android standard architecture by proposing the extension of

Dalvik as well as the substitution of the standard operating system by a real-time Linux-based operating system;

- The third approach simply replaces the Linux operating system for a Linux real-time version and real-time applications use the kernel directly;

- the fourth approach proposes the addition of a real-time hypervisor that supports the parallel execution of the Android platform in one partition while the other partition is dedicated to the real-time applications.

Regarding the first approach, depicted in Figure 5.3, this approach replaces the standard Linux kernel with a real-time operating system. This modification introduces predictability and determinism in the Android architecture. Therefore, it is possible to introduce new dynamic real-time scheduling policies through the use of scheduling classes; predict priority inversion and to have better resource management strategies.

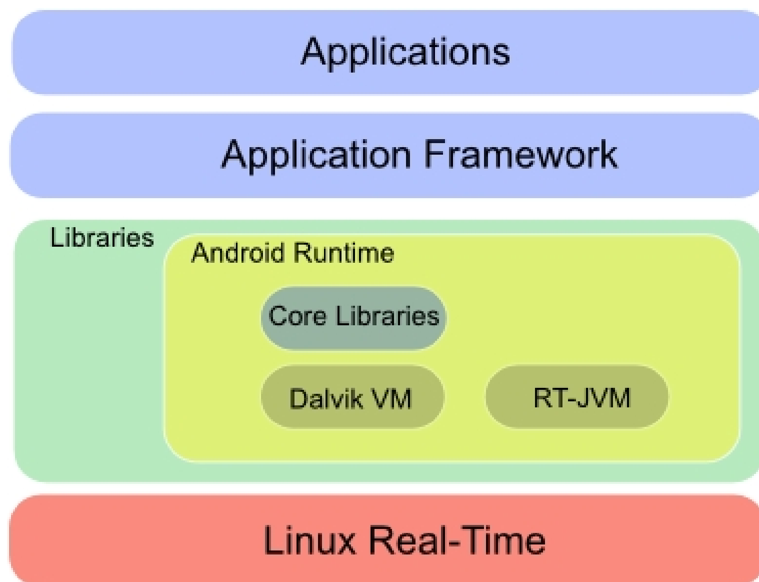

Figure 5.3: Android full Real-Time

However, this modification entails that all the device drivers supported natively need to be implemented in the operating system with predictability in mind. This task can be painful, especially during the integration phase. Nevertheless, this approach also leaves space for the implementation of the required real-time features in the Linux kernel. Implementing the features in the standard Linux kernel requires time, but it has the advantage of providing a more seamless integration with the remaining components belonging to the architectures involved.

The second modification proposed, within the first approach, is the inclusion of a real-time Java VM. This modification is considered advantageous as, with it, it is possible to have bounded memory management; real-time scheduling within the VM, depending on the adopted solution; better synchronisation mechanisms and finally to avoid priority inversion. These improvements are considered the most influential in achieving the intended deterministic behaviour at the VM level. It is important to note that the real-time VM interacts directly with the operating system's kernel for features such as task scheduling or bounded memory management.

As an example, if one considers task scheduling, the real-time VM is capable of mapping each task natively on the operating system where it will be scheduled. If the operating system supports other types of scheduling policies besides the fixed priority-based scheduler, the VM may use them to schedule its tasks. This means that most of the operations provided by real-time Java VMs are limited to the integration between the VM's supported features and the supported operating system's features.

Other advantage from this approach is that it is not necessary to keep up with the release cycles of Android, although some integration issues may arise between the VM and the kernel. The impact of introducing a new VM in the system is related to the fact that all the Android specificities must be implemented as well as *dex* support in the interpreter. Besides this disadvantage, other challenges may pose such as the integration between both VMs. This integration possibly entails the formulation of new algorithms to optimize scheduling and memory management in order to be possible to have an optimal integrated system as a whole and also to treat real-time applications in the correct manner.

The second proposed approach, presented in Figure 5.4, also introduces modifications in the architecture both in the operating system and virtual machine environments. As for the operating system layer, the advantages and disadvantages presented in the first approach are considered equal, as the principle behind it is the same. The major difference lies on the extension of Dalvik with real-time capabilities based on the Real-Time Specification for Java (RTSJ) [RTSJ, 2010].

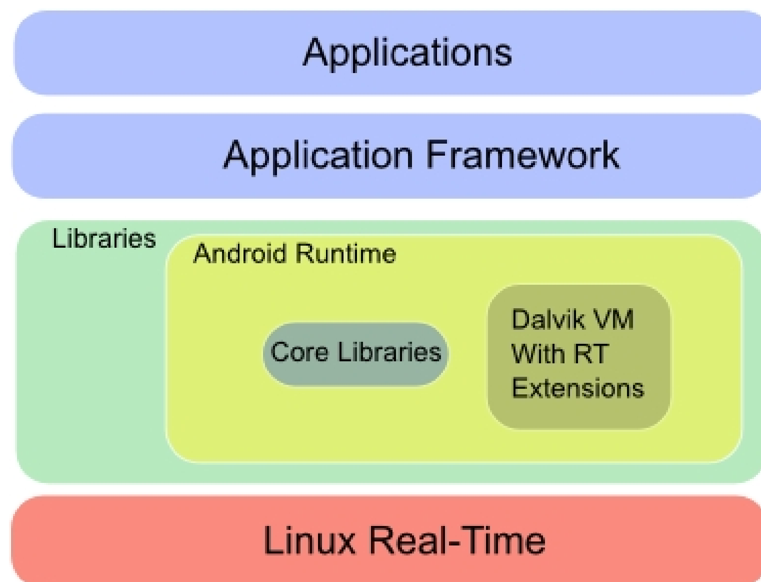

Figure 5.4: Android Extended

By extending Dalvik with RTSJ features we are referring to the addition of the following API classes: *RealTimeThread*, *NoHeapRealTimeThread*, as well as the implementation of generic objects related to real-time scheduling and memory management such as *Scheduler* and *MemoryAreas*. All of these objects will enable the implementation of real-time garbage collection algorithms, synchronization algorithms and finally, asynchronous event handling algorithms.

All of these features are specifically related to the RTSJ and must be considered in order to be possible to have determinism and predictability. However, its implementation only depends on the extent one wishes to have, meaning that a full compliant implementation may be achieved if the necessary implementation effort is applied in the VM extensions and the operating system's supported features.

This extension is beneficial for the system as with it, it is possible to incorporate a more deterministic behaviour at the VM level without the need of concerning about the particularities of Dalvik. Nevertheless, this approach has the disadvantage of having to keep up with the release cycles of the Android, more specially the VM itself, if one wants to add these extensions to all the available versions of the platform.

Two examples of this direction are [Guerra et al., 2010] and [Maia et al., 2010b]. The work in [Guerra et al., 2010] states that the implementation of a resource management framework is possible in the Android platform with some modifications in the platform. Although the results presented in this work are based on the CFS scheduler, work is being done to update the scheduler to a slightly modified version of EDF [Liu and Layland, 1973], that incorporates reservation-based scheduling algorithms as presented in [Faggioli et al., 2009].

The work reported in [Maia et al., 2010b] is being conducted in the scope of CooperatES project [CooperatES, 2010], where a proof of concept of a QoS-aware framework for cooperative embedded real-time systems has already been developed for the Android platform. Other important aspect of this work is the implementation of a new dynamic scheduling strategy named CSS [Nogueira and Pinho, 2007] in the Android platform.

Both works show that it is possible to propose new approaches based on the standard Linux and Android architectures and add real-time behaviour to them in order to take advantage of resource reservation and real-time task scheduling. With both of these features, any of these systems is capable of guaranteeing resource bandwidth to applications, within an interval of time, without jeopardising the system.

The third proposed approach, depicted in Figure 5.5, is also based in Linux real-time. This approach takes advantage of the native environment, where it is possible to deploy real-time applications directly over the operating system. This can be advantageous for applications that do not need the VM environment, which means that a minimal effort will be needed for integration, while having the same intended behaviour. On the other hand, applications that need a VM environment will not benefit from the real-time capabilities of the underlying operating system.

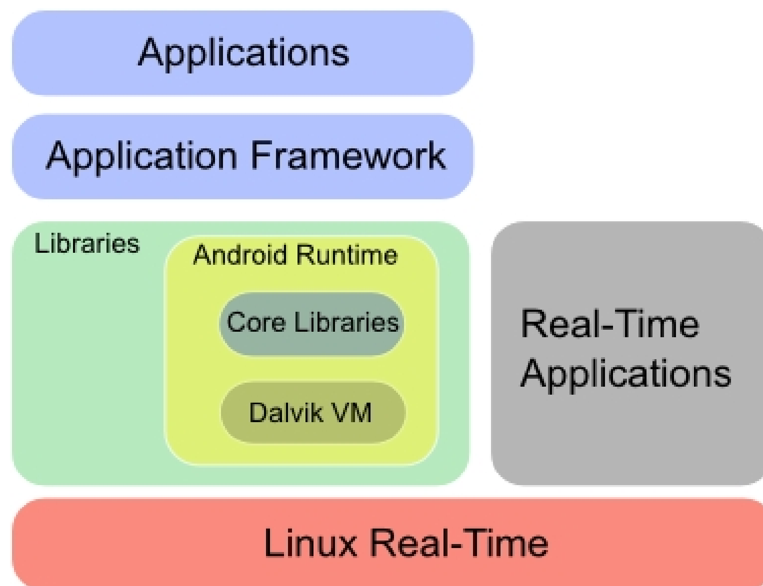

Figure 5.5: Android partly Real-Time

Finally, the fourth approach, depicted in Figure 5.6, employs a real-time hypervisor that is capable of executing Android as a guest operating system in one of the partitions and real-time applications in another partition, in a parallel manner.

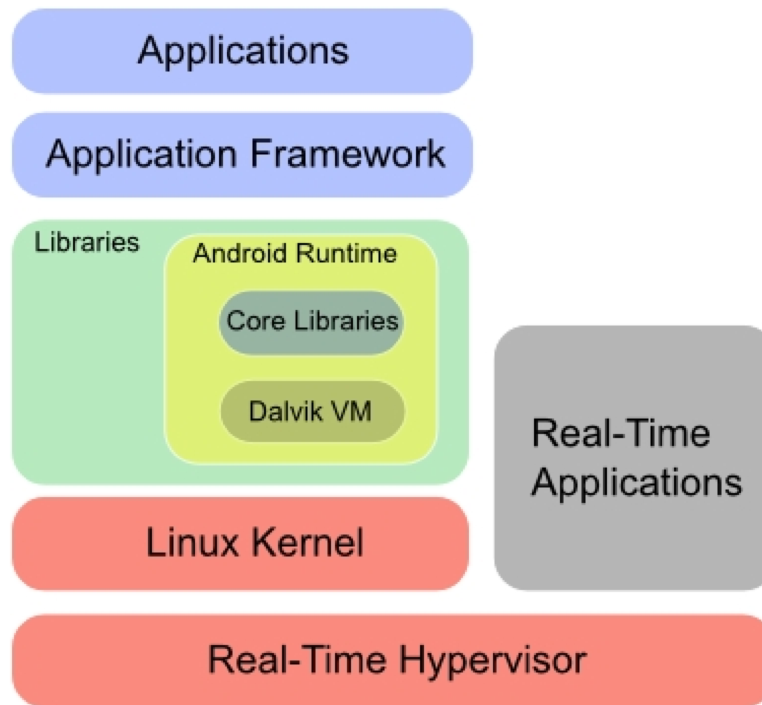

Figure 5.6: Android with a Real-Time Hypervisor

This approach is similar to the approach taken by the majority of the current real-time Linux solutions, such as RTLinux [Wind River Systems, 2010] or RTAI [Dipartimento di Ingegneria Aerospaziale, 2010]. These systems are able to execute real-time applications in parallel to the Linux kernel, where the real-time tasks have higher priority than the Linux kernel tasks, which means that hard real-time can be used.

On the other hand, the Linux partition tasks are scheduled using the spare time remaining from the CPU allocation. The main drawback from this approach is that real-time applications are limited to the features offered by the real-time hypervisor, meaning that they cannot use Dalvik or even most of the Linux services. Other limitation known lies on the fact that if a real-time application hangs, all the system may also hang.

5.6 Evaluation

In the previous sections, it was discussed the suitability of Android for open embedded real-time systems, its architecture internals were analysed, its current limitations pointed out, and four possible directions to incorporate real-time behaviour into the Android platform were proposed.

The acquired experiences with the directions proposed in [Maia et al., 2010c] allows us to conclude that the first direction is the one that causes less impact in the system as a whole. It allows the possibility of having Dalvik VM serving the needs of any native Android application, while at the same time, the real-time VM can handle the specific requests made by any real-time application. Nevertheless, while the inclusion of this second VM brings the desired real-time behaviour to the Android platform, it also

brings important challenges that should be considered. One may think of how the scheduling operations between both VMs are mapped into the OS, how the memory management operations will be managed in order to take advantage from the system's resources, and finally, how to handle thread synchronisation and asynchronous events in this dual VM environment.

Regarding scheduling, it must be assured by the OS that all the real-time tasks have higher priority than the normal Android tasks. This can be achieved by having a mechanism that maps each of the real-time tasks to a higher priority OS task. Then, the OS scheduler is responsible for assuring that these tasks are dispatched earlier than the remaining tasks. Thus, at a lower end limit a simple mapping mechanism must exist to perform this operation.

As for the memory management, one possible solution to consider would be to have a memory management abstraction layer that handles all the memory operations requested by both VMs, *i.e.* allocation and deallocation through the use of a smart garbage collector. The main benefit from this layer would come from the fact that it would be possible to have a single heap where all the objects would be managed and thus, the system's resources to deal with the dual VM environment would be optimised. The disadvantage lies in the way that Dalvik performs. Each Android application executes on its own Linux process with its own VM and garbage collector instances. Also, there is a part of the heap that is shared among all the processes. This *modus operandi* entails the need to, at least, integrate Dalvik with the abstraction layer and at the same time to modify its behaviour related to the per-process garbage collector instances.

Regarding thread synchronisation, as long as the real-time threads do not have the need to communicate with Dalvik threads, it is assured that this will not pose any kind of problems. However, if this communication is desired, a protection mechanism must be implemented in order to assure that a real-time thread will not block on a Dalvik thread and that priority inversion does not happen.

In terms of asynchronous events, a mapping mechanism must be sufficient to assure that the task that is waiting for the event will receive it in a bounded time interval. This mechanism must be implemented at the OS level in order to forward the events to the correct VM. Both VMs just need to implement the handlers for the events.

Although the scope of the presented work does not include changing the scheduling algorithms in the operating system or the creation of a dual VM environment, this strategy is already identified for future work.

5.7 Summary

This chapter presented the evaluation results conducted to the Android platform concerning its suitability for open embedded real-time systems. As the reader may have noticed Android has several limitations regarding these types of systems and none of the layers provides determinism and predictability in a way that it is possible to execute tasks in a bounded interval of time. However, by extending the platform and enhancing it with well known real-time mechanisms, particularly at the Linux kernel and VM layers, it is possible to obtain a platform that can provide enough determinism to real-time tasks.

With these extensions in mind, it is possible to implement a prototype where tasks can share resources and be executed by coalitions of heterogeneous nodes and, at the same time, ensure that the amount of time in which the tasks must execute is guaranteed.

The next chapter describes the details of a prototype implementation of the CooperatES in the Android platform.

CHAPTER 6

PROTOTYPE IMPLEMENTATION

As described in the previous chapters, the support for real-time behaviour is important in the Cooperates platform, but it is not the only issue involved. In order to have a Quality Of Service (QoS) aware framework, supporting the cooperative execution of services throughout several nodes, like the one proposed by CooperatES, services for managing QoS and cooperation must be also provided.

Therefore, this chapter is dedicated to the prototype implementation of the QoS and coalition support of CooperatES. Specific details are presented relating to the implementation of the framework components' in the Android platform, in order to allow the execution of applications in a distributed setting with quality requirements.

6.1 Introduction

As well as being a framework requiring real-time behaviour, CooperatES proposes its own individual components outlining an overall architecture, which is able to respond to the quality requirements being requested by the applications or users. By combining the real-time extensions proposed in the previous chapter and the CooperatES framework's components, it is possible to handle QoS and resource management with timing guarantees.

A detailed description of the framework's components was presented in section 3.2 of Chapter 3. However, a brief description of each component is provided next, for the sake of clarity and the understanding of the prototype implementation. The framework presents five major components:

- QoS Provider - Responsible for integrating the Local Provider and Coalition Organiser components;
- Local Provider - Responsible for evaluating if the local execution of a service is feasible;
- Coalition Organiser - Responsible for the coalition formation process;
- System Manager - Responsible for the overall system maintenance and availability of network nodes;

- Resource Manager - Responsible for managing a particular resource. Each node has several resource managers.

These components combined provide QoS and resource management and the possibility to create coalitions of nodes. Note that the framework relies on the underlying kernel layer to obtain the timing guarantees.

This chapter presents and describes the prototype implementation of the CooperatES framework in the Android platform, detailing the needed extensions to the Android's architecture to handle the formation of a coalition of cooperative nodes as a response to the environment conditions imposed by the end-user's QoS requirements.

It is important to note that most of the chapter contents are the result of reverse engineering, especially conducted for the lower level layers of the Android platform, version 1.6, i.e. Linux Kernel and Android's Virtual Machine (VM) environment. Nevertheless, a description of the steps taken to reverse engineer the platform is provided in the next section. Furthermore, this chapter only focuses on the integration of the CooperatES framework and Android.

6.2 Reverse Engineering Android

As stated in Chapter 4, there was lack of knowledge with the Android internals, particularly considering that the Android components and their interactions is not well documented. Both facts led to the need to conduct reverse engineering activities on the platform. Therefore, before presenting the prototype implementation of the CooperatES framework, it is important to describe the conducted work.

Much of the platform's internals knowledge stated from the analysis of the main device's log file, where each one of the log lines present information concerning specific operations.

The reverse engineering process entailed the analysis of each one of those lines in order to understand the following aspects:

- (i) The origin (which component) of the message in the source code;
- (ii) The importance of that component in the whole platform, *i.e.* device driver, core component, application, *etc.*;
- (iii) The boot process and the core components responsible for this process;
- (iv) How could a framework, such as CooperatES, be integrated seamlessly in the platform.

By understanding (i) and (ii), the main components of the platform were identified and their importance categorized in the form of the impact that have in the scope of the work presented in this thesis.

The aspect (iii) and (iv) were definitely the most important in the scope of the prototype implementation. Understanding the boot process (iii) enabled the comprehension of all of the components involved in it and allowed for a decision to choose where to implement the CooperatES framework's components (iv).

In fact, part of the reverse engineer process applied was to check each of the log messages and then verify where that specific log message was being generated in the source code. The other part of that process entailed understanding the source code around that specific message and infer specific

information about that component. Such information could be related to flow of execution, specific conditions used, special validations performed, in resume, all the information that could be gathered would be considered useful.

It is important to note that without reverse engineering the platform, it would not be possible to have a prototype of the CooperatES framework. The approach was proven advantageous as it provided the required context by analysing the source code. Furthermore, it is also important to mention that in spite of the importance of this work, a description of all of the aspects of the analysed components would be cumbersome and useless for the understanding of the work described in this thesis. Therefore, only the Android's components that were severely modified, *i.e.* changes in the usual information flow, are described.

The result of that reverse engineering work was integrated and applied in the CooperatES prototype implementation. This work is presented in the next sections.

6.3 Component Integration

The proposed implementation of the CooperatES framework into the standard Android's architecture is depicted in Figure 6.1. The real-time features must be provided by the Linux kernel, where CooperatES proposes the CSS scheduling algorithm [Nogueira and Pinho, 2007]. Since Android also does not provide support to QoS and resource management these are the main components to add to the architecture.

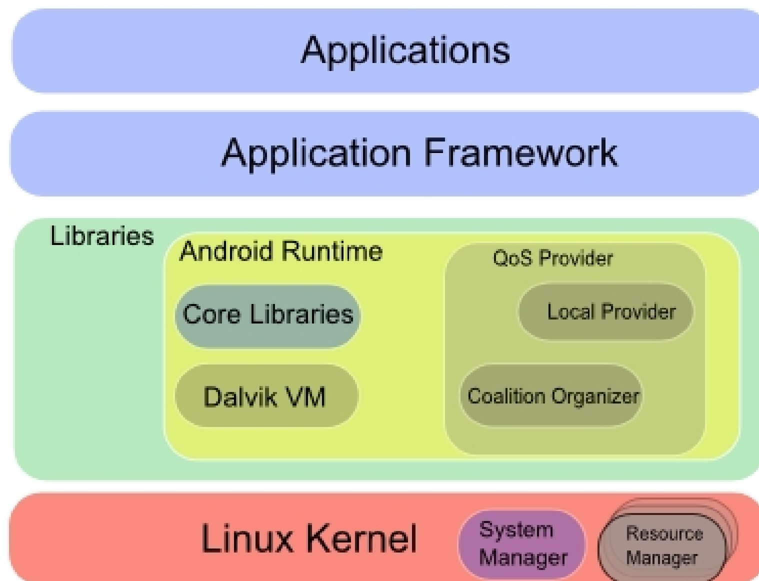

Figure 6.1: CooperatES Architecture

The Android's architecture does not provide mechanisms to control the resource demands imposed by applications. Therefore, and to overcome this limitation, *Resource Managers* are implemented at the Kernel layer for handling singular resources, i.e. memory, CPU, network, disk, etc. Due to the dynamic nature of the framework, the amount of available resources has a direct impact on the decisions taken by the *QoS Provider*, which will decide whether to form a coalition of cooperative nodes or not, based on actual amount of resources allocated.

In order to detect nodes entering and leaving the network, maintaining the overall system configuration and manage the coalitions' operations, namely their creation and dissolution, the *System Manager*

component was included into the prototype. This component is extremely important as it is the one responsible for the all the operations concerning the cooperation process.

All the implemented components provide a socket interface, enabling the communication between components located at different layers in the Android's stack. Without following this approach, it would not be possible to have a direct communication path between the *QoS Provider* and either the *Resource Managers* or *System Manager*, due to the internal design of the Android's architecture.

The *QoS Provider* runs seamlessly integrated in the Android Runtime as a natural part of its architecture. The main advantage provided by this integration is the ability of having a natural support for QoS in the architecture. There are two execution options available: (i) local execution or (ii) cooperative distributed execution. In either case, applications are processed in a transparent way for the user, as users are not aware of the exact distribution used to solve the computationally expensive applications.

6.4 Boot Time

During the device's boot time, the Linux kernel boots, as well as the *Resource Managers* and *System Manager*. These will run as kernel services and handle all the requests through a socket interface. It is important for the framework to be able to control resource usage through the *Resource Managers* and also to be able to maintain the overall system configuration with the *System Manager* component. As such, it was decided to include these components as near the device drivers as possible. The communication between the components will be socket-based, as previously mentioned.

Afterwards, the Android Runtime is started via a process named Zygote. The Zygote is responsible for the pre-initialisation of the Dalvik VM instance, which is used later on to spawn new VM instances with a partially initialised state. Whenever a new application is started, the main VM instance is forked via the Linux kernel and a new process is created. This new process is managed by the first created instance of the VM and it is assigned to the new application.

It is important to mention that Zygote is responsible for the initialisation of the core Java classes existing at the Application Framework level, which handle all the basic needs of the Android applications. The use of Zygote reduces the application's startup time, as most of the Android's core Java classes are already in a running state when a user desires to start an application.

Just before starting the VM, the runtime starts the *QoS Provider*, through the *startQoSProvider()* method, with its internal components attached – the *Local Provider* and the *Coalition Organiser*. The code listing for this step is presented in Listing 6.1.

The *QoS Provider* component executes as if natively belonging to the runtime itself. This is a strategic step for the CooperatES framework. The intention is to provide QoS management to applications, where each application is mapped into an instance of the Dalvik VM and, at the same time, to a Linux process. This way, it is possible to provide the application with an instance of a *QoS Provider* that is capable of handling each application's QoS constraints.

Listing 6.1: QoS Provider instantiation

```
if(startQoSProvider() != 0)
{
        LOGE("Unable to start QoS Provider\n");
        goto bail;
```

```
}

/* start the virtual machine */
if (startVm(&mJavaVM, &env) != 0)
        goto bail;
...
int AndroidRuntime::startQoSProvider()
{
    return createQoSProvider();
}

int createQoSProvider() {
    pthread_t threadID;

    LOGI("createQoSProvider␣was␣called\n");
    /* Create client thread */
    pthread_detach(pthread_self());
    if (pthread_create(&threadID, NULL, putSocketUpAndListen, NULL) != 0)
        LOGI("QoSProvider␣pthread_create()␣failed");
    return 0;
}
```

Zygote continues and starts up all the remaining components, beginning with Dalvik VM, then the core classes on the Application Framework and finally, the phone's applications. It is important to note that Android applications are able to use either the regular Android stack, if no QoS management is required, or use the proposed framework to satisfy their QoS requirements.

6.5 Handling QoS

Users or applications can specify their requirements by using the *AndroidManifest.xml*. By default, this file contains the definition of an application, e.g. the declaration of the libraries, components, and specific permissions required or handled by the application. In order to add support for QoS, it was naturally decided to use XML and the *AndroidManifest.xml* as the proper means for this purpose. Listing 6.2 shows an example of this file, considering the specification of the application's QoS requirements.

Basically, each application specifies its QoS requirements through a set of quality dimensions, as proposed in [Nogueira and Pinho, 2009] and presented in section 3.2 of Chapter 3. Each quality dimension must specify a name, and a set of attributes associated to the quality dimension being specified. Each attribute also has an identifier, and a set of quality values. These values have a data type associated- *float*, *integer* or *string*; a domain - discrete or continuous - and finally, the attribute values that will be used as a reference by the *QoS Provider*. Each application may define an infinite set of dimensions according to its QoS needs.

Listing 6.2: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
  <application ...>
    <activity ...></activity>
  </application>
```

```
<qos-parameters>
  <dimensions>
    <dimension>
      <dimension-name>Video Quality</dimension-name>
        <attributes>
          <attribute>
            <attribute-name>color depth</attribute-name>
              <attribute-values
                domain="discrete"
                datatype="integer">1,3,8,16,24
              </attribute-values>
              </attribute>
            <attribute-name>sampling rate</attribute-name>
                <attribute-values
                  domain="discrete"
                  datatype="string">SQCIF,QCIF,CIF
                </attribute-values>
            </attribute>
           <attribute>...</attribute>
        </attributes>
     </dimension>
     <dimension>...</dimension>
  </dimensions>
</qos-parameters>
</manifest>
```

By allowing the definition of the QoS requirements in an XML format and validating this definition in the Android code, quality information consistency among heterogeneous nodes is achieved and, therefore, there is a common understanding of the QoS requirements by all the nodes involved in the cooperative execution. This is a major advantage offered by the Android architecture, and was also taken into consideration in the framework's implementation. With this XML format, it is also possible to map and form hierarchies between quality dimensions and their attributes.

The application QoS parameters are parsed by the Package Parser, as presented in Listing 6.3. Here, the *parseQoS()* method is called to load the parameters into memory by creating Java objects which will then be used by Android to handle the application's workflow. Each dimension is also parsed and added to a list where all the dimensions and their attributes are mapped. Note that all the application's properties are also loaded into memory, from the manifest file, during boot time.

Listing 6.3: QoS parameter parsing in PackageParser.java

```
else if (tagName.equals("qos-parameters"))
{
        if (!parseQoS(pkg, res, parser, attrs, flags, outError))
        {
                return null;
        }
        else
        {
                pkg.usesQoS = true;
                pkg.applicationInfo.setQoS(pkg. qosDimensions);
                pkg.applicationInfo.usesQoS = true;
```

```
        }
...
}

private boolean parseQoS(...)
{
        if (tagName.equals("dimensions"))
        {
                ArrayList <QoSDimension> dimArray = parseDimension(...);
        }
}
...
private ArrayList <QoSDimension> parseDimension(...)
{
        ArrayList <QoSDimension> dimensionArray = new ArrayList<QoSDimension>();
        QoSDimension tmpDimension;

        ...

        // each TAG is parsed and if everything is looks good
        // add the dimension to the array
        dimensionArray.add(tmpDimension);
...
}
```

Afterwards, the parameters are sent to the *QoS Provider* component via a Java Native Interface (JNI) call to Zygote (Listing 6.4). There is a decision path based on the existence of QoS parameters: if the application presents QoS constraints, the method *Zygote.forkAndSpecializeQoS()* is called, otherwise, the standard *Zygote.forkAndSpecialize()* Android method for forking the applications will be called. The listing is related to the *ZygoteConnection.java* file.

Listing 6.4: JNI Call to Dalvik VM

```
...
if(parsedArgs.qosArgs != null)
{
        qosArgs = parsedArgs.qosArgs.toArray(stringArray2d);

        pid = Zygote.forkAndSpecializeQoS(parsedArgs.uid, parsedArgs.gid,
        parsedArgs.gids, parsedArgs.debugFlags, rlimits, qosArgs);
}
else
{
        pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,
          parsedArgs.gids, parsedArgs.debugFlags, rlimits);
}
...
```

This JNI call is handled by *dalvik_system_Zygote.c*, which belongs to the VM. The *Zygote.forkAndSpecializeQoS()* method presented in Listing 6.5 is responsible for handling the application QoS parameters and sending them via a socket communication to the *QoS Provider* which is listening for requests.

Each request is delivered to the *Local Provider*, which based on the resource allocation levels sent

by each *Resource Manager*, determines if there are enough resources to locally execute the application. If this is the case, the VM will be forked and a new process is assigned to the application, following the typical flow of Android. On the other hand, the *Coalition Organiser* is invoked whenever the imposed QoS constraints cannot be locally satisfied.

Listing 6.5: Zygote before forking

```
static void Dalvik_dalvik_system_Zygote_forkAndSpecializeQoS(
        const u4* args,
    JValue* pResult)
{
pid_t pid;
ArrayObject *qosArgs = (ArrayObject *) args[5];

if (qosArgs == NULL)
{
        LOGI("Zygote Args are NULL\n");
}
else
{
  // socket code was removed

  u4 i, j;

  ArrayObject** tuples = (ArrayObject **) (qosArgs->contents);

  for (i = 0; i < qosArgs->length; i++)
  {
    u4 count = tuples[i]->length;
        StringObject** qos_tuple = (StringObject**) tuples[i]->contents;

        for (j = 0; j < count; j++)
        {
                StringObject* contents = (StringObject*) qos_tuple[j];
                char* tmpString = dvmCreateCstrFromString(contents);

                LOGI("Sendind QoS parms to QoSProvider -> %s \n", tmpString);

                // send params to QoS Provider
                n = write(sockfd, tmpString, strlen(tmpString));

                memset(&buffer, 0, sizeof (buffer));
        }
}

// call standard function for fork
pid = forkAndSpecializeCommon(args);

RETURN_INT(pid);
}
```

The *Coalition Organiser* is responsible for partitioning the service in a set of blocks; requesting service proposals from neighbour nodes (detected by the *System Manager*) for each of those blocks;

deciding which nodes will form the coalition based on their service proposals and user's QoS requirements. Currently, only services that can be divided in sets of independent blocks are supported, although algorithms for coordinating the execution of interdependent blocks were already proposed in [Nogueira et al., 2009]. A directed graph, describing the inputs and outputs of each block is dynamically formed when the service is partitioned.

Whenever the coalition partners receive an application's block sent by the requesting node, a new VM instance is spawned by forking the main Dalvik VM instance, dealing with blocks as if they were locally started by a user.

6.6 Framework's Workflow

Figure 6.2 presents the framework's workflow in the form of a Unified Modelling Language (UML) activity diagram. The diagram presents a high level overview of the framework workflow and its objective is to provide an overview of the interactions presented in the previous sections, in order to summarize the information provided.

As a resume of what was previously presented, whenever an application is started, the Package Parser component will be invoked in order to verify if the application specified any QoS requirements. If the application did not specify any QoS parameters, the usual Android flow will be followed, meaning that the VM will be forked and the new application will execute. On the other hand, if the application requires QoS support, the *QoS Provider* will be invoked. This invocation will trigger the *Local Provider* that will verify with each one of the *Resource Managers* the availability of resources. If the amount of resources is sufficient to run the application locally, the VM will be forked and the new application will execute. If the number of resources is insufficient, the *Coalition Organiser* will be invoked in order to form a coalition of nodes. The *Coalition Organiser* will contact the *System Manager* for this purpose.

6.7 Summary

This chapter presented the implementation details of the CooperatES main components in the Android platform. This work required the analysis of the platform (including reverse engineering), the analysis of the CooperatES framework requirements, and the integration between the theoretical model and the Android platform.

The objective was to have a seamless integration between both in order to provide the end users with an easy to use framework, which could be used in the specification of their applications' quality requirements. The framework allows such a specification and the application of resource management with timing requirements, by using coalitions of nodes to solve resource constraints that may happen during the normal execution of applications.

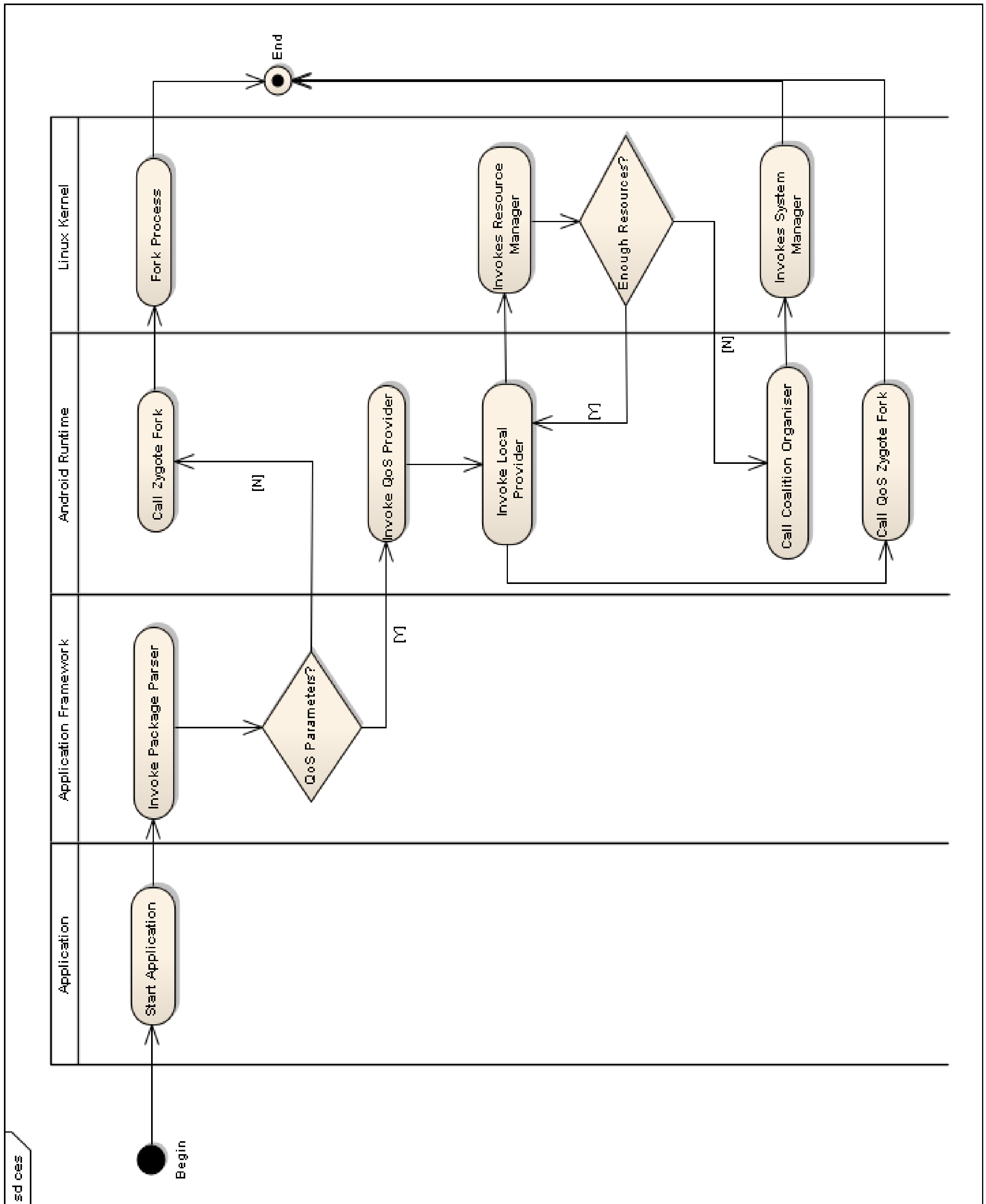

Figure 6.2: CooperatES Framework Workflow

CHAPTER 7

CONCLUSION AND FUTURE WORK

As the complexity of embedded systems increases, multiple applications have to compete for the limited resources of a single device. This situation is particularly critical for small embedded devices used in consumer electronics, telecommunication, industrial automation, or automotive systems. In fact, in order to satisfy a set of constraints related to weight, space, and energy consumption, these systems are typically built using microprocessors with lower processing power and limited resources. In this context, resource constrained devices may need to collectively execute services with neighbour nodes in order to fulfil the complex Quality of Service (QoS) constraints imposed by users and applications.

The CooperatES framework is a QoS-aware framework addressing the increasing demands on resources and performance by allowing applications (named services) to be executed by temporary coalitions of nodes. Users encode their own relative importance of the different QoS parameters for each service and the framework uses this information to determine the distributed resource allocation that maximises the satisfaction of those constraints.

In this context, the goal of this thesis was to analyse, implement and evaluate the CooperatES main QoS/resource management and coalition concepts in order to validate the feasibility of such concepts in a current embedded platform.

In order to accomplish those objectives, a state of art analysis was conducted. The topics surveyed, presented in Chapter 2, focused on Quality of Service management, real-time scheduling and Real-Time Java. The objective behind this analysis was to gather knowledge in order to be possible to understand the specific concepts approached in CooperatES.

The CooperatES framework is the basis of this work, and therefore Chapter 3 introduces its objectives and the problems it tries to solve, and the main components of the framework's architecture. The work conducted entailed the analysis of the framework itself as well as its requirements. These requirements were taken into account for the feasibility analysis.

Before implementing the framework's concepts, a platform had to be chosen. Several platforms were analysed, both real-time Java platforms and generic embedded platforms as Android. The results of this analysis is presented in Chapter 4. In that chapter, it was also discussed the main reasons for selecting Android as the testbed for demonstrating the feasibility of the CooperatES framework.

Android is not a real-time platform by default and therefore this led, in Chapter 5, to an extensive

analysis of its limitations and the proposal of several different approaches to extend its architecture for real-time systems. These extensions had in mind the mobile and cooperative execution concept together with real-time issues in a seamless way and as a natural ability of the architecture to satisfy the needs of the industry or the most demanding users.

There is no feasibility analysis without an implementation of a prototype that can demonstrate the main issues being proposed. This prototype implementation is presented in Chapter 6. The chapter presents the needed modifications, implementation oriented, that were performed so that Android provides support to the QoS and node coalition features of CooperatES.

Android can be seen as a potential platform for real-time environments and, as such, numerous industry domains would benefit from an architecture with such capabilities. The feasibility results are positive, meaning that with some effort, as demonstrated in this thesis, it is possible to have the desired real-time behaviour on any Android device. This behaviour may suit specific applications or components by providing them the ability of taking advantage of temporal guarantees, and therefore, to behave in a more predictable manner.

One difficulty however may impair that development. Android is a well documented platform in what concerns its higher layers, but the lower layers of the platform are not so well documented. This had some impact in the work presented, since it was necessary to reverse engineer several components of the Android platform.

Nevertheless, the knowledge gained and the implemented prototype allow Android to be used for the continuation of the CooperatES research activities:

- Integration of separate layers and concepts (such as code mobility, resource mapping, adaptability, *etc.*) in Android in order to have a complete CooperatES prototype;
- Afterwards, the implementation of applications allowing exploring the scenarios presented in Chapter 3, as well as the specification of new scenarios that may take advantage of the framework's concepts;
- Exploration of the virtual machine environment capabilities in order to make it compliant with the RTSJ.

In particular, and what concerns this last point, the introduction of a double virtual machine environment in the Android platform (as identified in Chapter 5) is considered an important, but challenging, research topic, which will be explored in the future.

BIBLIOGRAPHY

T. Abdelzaher and K. Shin. End-host architecture for qos-adaptive communication. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, pages 121–, Washington, DC, USA, 1998. IEEE Computer Society.

Tarek F. Abdelzaher and Kang G. Shin. Qos provisioning with qcontracts in web and multimedia servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 44–, Washington, DC, USA, 1999. IEEE Computer Society.

T.F. Abdelzaher, E.M. Atkins, and K.G. Shin. Qos negotiation in real-time systems and its application to automated flight control. *Computers, IEEE Transactions on*, 49(11):1170 – 1183, November 2000.

L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, page 4, Washington, DC, USA, 1998. IEEE Computer Society.

Aicas. Jamaicavm, March 2010. URL `http://www.aicas.com/jamaica.html`.

Android. Home page, January 2010. URL `http://www.android.com/`.

Android-x86. Android-x86 project, January 2010. URL `http://www.android-x86.org/`.

Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst.*, 7:5:1–5:49, December 2007.

Atego. Perc products, December 2010. URL `http://www.aonix.com/perc.html`.

Cristina Aurrecoechea, Andrew T. Campbell, and Linda Hauw. A survey of qos architectures. *Multimedia Syst.*, 6:138–151, May 1998.

Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbruster, Edward Pla, and David Holmes. A real-time java virtual machine for avionics - an experience report. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 384–396, Washington, DC, USA, 2006. IEEE Computer Society.

G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 328–335, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2247-5.

A. Block, B. Brandenburg, J.H. Anderson, and S. Quint. An adaptive framework for multiprocessor real-time system. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 23 –33, July 2008.

Dan Bornstein. Dalvik vm internals, March 2010. URL `http://sites.google.com/site/io/dalvik-vm-internals`.

R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview, 1994.

B.B. Brandenburg and J.H. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 184 –193, July 2009.

S. Brandt, G. Nutt, T. Berk, and M. Humphrey. Soft real-time application execution with dynamic quality of service assurance. In *Quality of Service, 1998. (IWQoS 98) 1998 Sixth International Workshop on*, pages 154 –163, May 1998a.

S. Brandt, G. Nutt, T. Berk, and J. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 307–, Washington, DC, USA, 1998b. IEEE Computer Society.

Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009. ISBN 0321417453, 9780321417459.

Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Proceedings of the 21st IEEE conference on Real-time systems symposium*, RTSS'10, pages 295–304, Washington, DC, USA, 2000. IEEE Computer Society.

Marco Caccamo, Giorgio C. Buttazzo, and Deepu C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Trans. Comput.*, 54:198–213, February 2005.

Guangyu Chen, Byung-Tae Kang, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. Studying energy trade offs in offloading computation/compilation in java-enabled mobile devices. *IEEE Trans. Parallel Distrib. Syst.*, 15:795–809, September 2004.

Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2:325–346, 1990. 10.1007/BF01995676.

David D. Clark, Scott Shenker, and Lixia Zhang. Supporting real-time applications in an integrated services packet network: architecture and mechanism. *SIGCOMM Comput. Commun. Rev.*, 22(4): 14–26, October 1992.

A. Colin and S.M. Petters. Experimental evaluation of code properties for wcet analysis. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 190 – 199, December 2003.

C.L. Compton and D.L. Tennenhouse. Collaborative load shedding for media-based applications. In *Multimedia Computing and Systems, 1994., Proceedings of the International Conference on*, pages 496 –501, May 1994.

CooperatES. Home page, January 2010. URL `http://www.cister.isep.ipp.pt/projects/cooperates/`.

Angelo Corsaro. jrate home page, March 2010. URL `http://jrate.sourceforge.net/`.

T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni. Adaptive reservations in a linux environment. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:238, 2004.

R.I. Davis. Approximate slack stealing algorithms for fixed priority pre-emptive systems. Technical report, Department of Computer Science, University of York, York, United Kingdom, November 1993.

R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 222 –231, December 1993.

Gan Deng, Douglas C. Schmidt, Christopher D. Gill, and Nanbor Wang. *Handbook of Real-Time and Embedded Systems*, chapter QoS-Enabled Component Middleware for Distributed Real-Time and Embedded Systems. CRC Press, 2008. ISBN: 1584886781.

Politecnico di Milano Dipartimento di Ingegneria Aerospaziale. Realtime application interface for linux, June 2010. URL `https://www.rtai.org/`.

Dario Faggioli. Sched_deadline scheduling policy, December 2010. URL `http://www.evidence.eu.com/content/view/313/390/`.

Dario Faggioli, Michael Trimarchi, and Fabio Checconi. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC09)*, pages 1984–1989, New York, NY, USA, 2009. ACM.

Changpeng Fan. Realizing a soft real-time framework for supporting distributed multimedia applications. In *Distributed Computing Systems, 1995., Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of*, pages 128 –134, August 1995.

Ian Foster, Markus Fidler, Alain Roy, Volker Sander, and Linda Winkler. End-to-end quality of service for high-end applications. *Elsevier Computer Communications*, 27(14):1375–1388, 2004. Network Support for Grid Computing.

Sourav Ghosh, Ragunathan (Raj) Rajkumar, Jeffery Hansen, and John Lehoczky. Scalable resource allocation for multi-processor qos optimization. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, page 174, Washington, DC, USA, 2003. IEEE Computer Society.

James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201703238.

Xiaohui Gu, Alan Messer, Ira Greenberg, Dejan Milojicic, and Klara Nahrstedt. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, 3:66–73, July 2004.

Raphael Guerra, Stefan Schorr, and Gerhard Fohler. Adaptive resource management for mobile terminals - the actors approach. In *Proceedings of 1st Workshop on Adaptive Resource Management (WARM10)*, Stockholm, Sweden, April 2010.

Jeffery P. Hansen, John P. Lehoczky, and Ragunathan Rajkumar. Optimization of quality of service in dynamic systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, page 95, Washington, DC, USA, 2001. IEEE Computer Society.

M. Teresa Higuera-Toledano and Valerie Issarny. Java embedded real-time systems: An overview of existing solutions. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 392–399, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0607-0.

David Hutchison, Geoff Coulson, Andrew Campbell, and Gordon S. Blair. *Quality of service management in distributed systems*, pages 273–302. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994. ISBN 0-201-62745-0.

IEEE. Ieee standard 1003.1, March 2010. URL `http://www.opengroup.org/onlinepubs/009695399/`.

Kevin Jeffay and David Bennett. A rate-based execution abstraction for multimedia computing. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV '95, pages 64–75, London, UK, 1995. Springer-Verlag.

E. Douglas Jensen, C. Douglas Locke, and Hideyuki Tokuda. A time-driven scheduling model for real-time operating systems. In *IEEE Real-Time Systems Symposium*, pages 112–122, December 1985.

M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, page 12, Washington, DC, USA, 1995. IEEE Computer Society.

Michael B. Jones and John Regehr. Cpu reservations and time constraints: implementation experience on windows nt. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*, page 10, Berkeley, CA, USA, 1999. USENIX Association.

Michael B. Jones, Daniel L. McCulley, Alessandro Forin, Paul J. Leach, Daniela Roşu, and Daniel L. Roberts. An overview of the rialto real-time architecture. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, EW 7, pages 249–256, New York, NY, USA, 1996. ACM.

T. Kalibera, F. Pizlo, A.L. Hosking, and J. Vitek. Scheduling hard real-time garbage collection. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 81 –92, 2009.

S. Kato, R. Rajkumar, and Y. Ishikawa. Airs: Supporting interactive real-time applications on multicore platforms. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 47 –56, July 2010.

Md. Shahadatullah Khan. *Quality adaptation in a multisession multimedia system: model, algorithms, and architecture*. PhD thesis, University of Victoria, Victoria, B.C., Canada, Canada, 1998. AAINQ36645.

Jihong Kim and Tajana Simunic Rosing. *Handbook of Real-Time and Embedded Systems*, chapter Power-Aware Resource Management Techniques for Low-Power Embedded Systems. CRC Press, 2008. ISBN: 1584886781.

Ulrich Kremer, Jamey Hicks, and James Rehg. A compilation framework for power and energy management on mobile computers. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 115–131, Berlin, Heidelberg, 2003. Springer-Verlag.

Chen Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time mach. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 220 –229, June 1996.

J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Real-Time Systems Symposium, 1992*, pages 110 –123, December 1992.

Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '01, pages 238–246, New York, NY, USA, 2001. ACM.

Zhiyuan Li, Cheng Wang, and Rong Xu. Task allocation for distributed multimedia processing on wirelessly networked handheld devices. *Parallel and Distributed Processing Symposium, International*, 1: 79–84, 2002.

Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 410–421, Washington, DC, USA, 2005. IEEE Computer Society.

Giuseppe Lipari and Sanjoy Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. *Real-Time Systems, Euromicro Conference on*, 0:193, 2000.

C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

Jane W. S. Liu, Kwei-Jay Lin, Riccardo Bettati, David Hull, and Albert Yu. Use of imprecise computation to enhance dependability of real-time systems. In Gary M. Koob and Clifford G. Lau, editors, *Foundations of Dependable Computing*, volume 284 of *The Kluwer International Series in Engineering and Computer Science*, pages 157–182. Springer US, 1994. ISBN 978-0-585-27316-7.

Joseph P. Loyall and Richard E. Schantz. *Handbook of Real-Time and Embedded Systems*, chapter Dynamic QoS Management in Distributed Real-Time Embedded Systems. CRC Press, 2008. ISBN: 1584886781.

Gianpaolo Macario, Marco Torchiano, and Massimo Violante. An in-vehicle infotainment software architecture based on google android. In *SIES*, pages 257–260, Lausanne, Switzerland, July 2009. IEEE. ISBN 978-1-4244-4110-5.

Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho. Experiences on the implementation of a cooperative embedded system framework: short paper. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 70–72, New York, NY, USA, 2010a. ACM.

Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho. Experiences on the implementation of a cooperative embedded system framework. Technical report, CISTER Research Centre, Porto, Portugal, June 2010b.

Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho. Evaluating android os for embedded real-time systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Brussels, Belgium, July 2010c.

Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:211, 2004.

Steven McCanne and Van Jacobson. vic: a flexible framework for packet video. In *Proceedings of the third ACM international conference on Multimedia*, MULTIMEDIA '95, pages 511–522, New York, NY, USA, 1995. ACM.

C.W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: operating system support for multimedia applications. In *Multimedia Computing and Systems, 1994., Proceedings of the International Conference on*, pages 90–99, May 1994.

A. Miyoshi, T. Kitayama, and H. Tokuda. Implementation and evaluation of real-time java threads. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 166 –175, December 1997.

Jason Nieh and Monica S. Lam. The design, implementation and evaluation of smart: a scheduler for multimedia applications. *SIGOPS Operating Systems Review*, 31:184–197, October 1997.

Kelvin Nilsen. Adding real-time capabilities to java. *Communications of the ACM*, 41:49–56, June 1998.

Luís Nogueira. *Time-Bounded Adaptive Quality of Service Management for Cooperative Embedded Real-Time Systems*. PhD thesis, Faculdade de Ciências da Universidade do Porto, 2009.

Luís Nogueira and Luís Miguel Pinho. Capacity sharing and stealing in dynamic server-based real-time systems. In *Proceedings of the 21th IEEE International Parallel and Distributed Processing Symposium*, page 153, Long Beach,CA,USA, March 2007.

Luís Nogueira and Luís Miguel Pinho. Time-bounded distributed qos-aware service configuration in heterogeneous cooperative environments. *Journal of Parallel and Distributed Computing*, 69(6):491–507, June 2009.

Luís Nogueira, Luís Miguel Pinho, and Jorge Coelho. Coordinated runtime adaptations in cooperative open real-time systems. In *Proceedings of the 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Vancouver, Canada, August 2009.

Open Handset Alliance. Home page, June 2010. URL http://www.openhandsetalliance.com/.

Oracle Corporation. Sun java real-time system, November 2010. URL http://java.sun.com/javase/technologies/realtime/index.jsp/.

Mazliza Othman and Stephen Hailes. Power conservation strategy for mobile computers using load sharing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2:44–51, January 1998.

PalmSource Inc. Openbinder 1.0, March 2010. URL http://www.angryredplanet.com/~hackbod/openbinder/.

L. Palopoli, P. Valente, T. Cucinotta, L. Marzario, and A. Mancina. A unified framework for multiple type resource scheduling with qos guarantees. In *Proceedings of the 1th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 67–75, Mallorca, Spain, July 2005.

Purdue University. Ovm project, November 2010. URL http://www.cs.purdue.edu/homes/jv/soft/ovm/index.html.

QNX Software Systems. Qnx, December 2010. URL http://www.qnx.com/.

R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, page 298, Washington, DC, USA, 1997. IEEE Computer Society.

John Regehr. *Handbook of Real-Time and Embedded Systems*, chapter Safe and Structured Use of Interrupts in Real-Time and Embedded Software. CRC Press, 2008. ISBN: 1584886781.

RTJComputing. simplertj, March 2010. URL http://www.rtjcom.com/main.php?p=home.

RTMACH. Linux/rk, March 2010. URL http://www.cs.cmu.edu/~rajkumar/linux-rk.html.

RTSJ. Real-time specification for java 1.0.2, January 2010. URL http://www.rtsj.org/specjavadoc/book_index.html.

Alexey Rudenko, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Saving portable computer battery power through remote process execution. *Mobile Computing and Communications Review*, 2, 1998.

H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications, 2003.

L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39:1175–1185, September 1990.

Richard Staehli, Jonathan Walpole, and David Maier. A quality-of-service specification for multimedia presentations. *Multimedia Syst.*, 3:251–263, November 1995.

John A. Stankovic. Strategic directions in real-time and embedded systems. *ACM Comput. Surv.*, 28(4): 751–763, December 1996.

The University of North Carolina at Chapel Hill. Litmusrt, December 2010. URL `http://www.cs.unc.edu/~anderson/litmus-rt/`.

Hide Tokuda and Takuro Kitayama. Dynamic qos control based on real-time threads. In Doug Shepherd, Gordon Blair, Geoff Coulson, Nigel Davies, and Francisco Garcia, editors, *Network and Operating System Support for Digital Audio and Video*, volume 846 of *Lecture Notes in Computer Science*, pages 114–123. Springer Berlin / Heidelberg, 1994. 10.1007/3-540-58404-8_11.

Linus Torvalds. Linux kernel, December 2010. URL `http://www.kernel.org/`.

Marisol Garcia Valls, Alejandro Alonso, Jose Ruiz, and Angel Groba. An architecture of a quality of service resource manager middleware for flexible embedded multimedia systems. In *Proceedings of the 3rd international conference on Software engineering and middleware*, SEM'02, pages 36–55, Berlin, Heidelberg, 2003. Springer-Verlag.

Nalini Venkatasubramanian and Klara Nahrstedt. An integrated metric for video qos. In *Proceedings of the fifth ACM international conference on Multimedia*, MULTIMEDIA '97, pages 371–380, New York, NY, USA, 1997. ACM.

Cheng Wang and Zhiyuan Li. Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 119–130. ACM Press, 2004.

Andrew Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley & Sons, 2004. ISBN 047084437X.

Andy Wellings and Alan Burns. *Handbook of Real-Time and Embedded Systems*, chapter Real-Time Java. CRC Press, 2008. ISBN: 1584886781.

Wind River. Vxworks, December 2010. URL `http://www.windriver.com/products/vxworks/`.

Wind River Systems. Real-time linux, June 2010. URL `http://www.rtlinuxfree.com/`.

Lixia Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. Rsvp: a new resource reservation protocol. *Communications Magazine, IEEE*, 40(5):116 –127, May 2002.