Luís Miguel Pinho Nogueira

Time-Bounded Adaptive Quality of Service Management for Cooperative Embedded Real-Time Systems

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2009

Luís Miguel Pinho Nogueira

Time-Bounded Adaptive Quality of Service Management for Cooperative Embedded Real-Time Systems

Tese submetida à Faculdade de Ciências da Universidade do Porto
para obtenção do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2009

This thesis is dedicated to my family and friends, without whom none of this would have been even possible

Acknowledgements

First and foremost, I would like to thank my advisor Prof. Luís Miguel Pinho for his guidance, support, and dedication. This work would not be possible without his extraordinary expertise in real-time systems, his impressive ability to solve problems, and our friendship. I also wish to express my gratitude to my co-advisor Prof. Miguel Filgueiras for his support throughout these years, valuable suggestions, and patiently and meticulously reading of the draft version of this thesis. Needless to say, I alone bear the responsibility for any errors in this current version.

I have benefited from many and varied interactions with fellows at the IPP-HURRAY! research group. Eduardo Tovar, Mário Alves, Filipe Pacheco, Nuno Pereira, Björn Andersson, Paulo Baltarejo, and Luís Ferreira all showed me how to conduct real-time systems research from various angles and provided valuable personal and institutional support which will always be treasured.

Jorge Coelho, long time friend and fellow at ISEP, has always offered stimulating discussions on issues of interest to this work.

I also owe a special thanks to Sandra Almeida at IPP-HURRAY who kept things run smoothly and who cheerfully and efficiently made all the administrative hassles went way.

I must also acknowledge the valuable support given by the Portuguese Fundation for Science and Technology (FCT) through the PhD grant SFRH/BD/30818/2006.

Finally, I am forever indebted to my parents and my brother for their love, understanding, endless patience, and encouragement when it was most required.

Abstract

As an increasing number of users runs both real-time and non-real-time applications in an embedded system, the issue of how to provide an efficient resource utilisation in dynamic, open, and heterogeneous environments becomes very important. The need arises from the fact that independently developed services can enter and leave the system at any time, without any previous knowledge about their real execution requirements and tasks' inter-arrival times but, nevertheless, response to events still has to be provided within timing constraints in order to guarantee a desired level of performance. Within this context, this thesis proposes a cooperative QoS-aware framework which allows resource constrained devices to collectively execute services in cooperation with more powerful neighbours.

The proposed framework allows devices to collectively execute services in order to meet non-functional requirements that otherwise would not be met by an individual execution. Devices dynamically group themselves into coalitions, establishing initial service configurations which maximise the satisfaction of each users' QoS preferences associated with the new services and minimise the impact on the global QoS caused by the new services' arrival. At coalitions' runtime, the dynamic QoS arbitration among competing services is done under the users' control, extending each user's influence not only to a coalition's formation phase but also to its operation.

The traditional QoS optimisation approach, mainly concentrated on finding single optimal or with a fixed sub-optimality bound solutions, is reformulated as a heuristic-based anytime optimisation that can be interrupted at any time and still able to provide a service solution, even when services exhibit unrestricted local and distributed QoS inter-dependencies among their tasks. The proposed anytime approach is able to quickly find a good initial service solution and effectively optimise the rate at which the quality of the determined solution improves at each iteration of the algorithms.

Autonomous individual runtime adaptations are coordinated through an one-step decentralised model based on an effective feedback mechanism, able to reduce the

complexity of the needed interactions among nodes until a collective adaptation behaviour is determined. Positive feedback is used to reinforce the selection of the new desired global service solution, while negative feedback discourages nodes to act in a greedy fashion as this adversely impacts on the provided service levels at neighbouring nodes. By exchanging feedback on the desired self-adaptive actions nodes converge towards a global solution, even if that means not supplying their individually best solutions. As a result, each node, although autonomous, is influenced by, and can influence, the behaviour of other nodes in a coalition.

The dynamic changes of services' requirements is handled in a predictable fashion, enforcing timing constraints with a certain degree of flexibility, aiming to achieve the desired tradeoff between predictable performance and an efficient use of resources. The proposed CSS (Capacity Sharing and Stealing) dynamic server-based scheduler supports the coexistence of guaranteed and non-guaranteed bandwidth servers to efficiently handle soft-tasks' overloads by making additional capacity available from two sources: (i) residual capacity allocated but unused when jobs complete in less than their budgeted execution time; (ii) stealing capacity from inactive non-isolated servers used to schedule aperiodic best-effort framework's management tasks.

The effectiveness and reduced complexity of CSS in managing unused reserved capacities without any previous complete knowledge about the tasks' runtime behaviour, makes it appropriate to be used as the basis of a more powerful scheduler able to handle dependent tasks sets which share access to some of the system's resources and exhibit precedence constraints. The proposed CXP (Capacity Exchange Protocol) integrates the concept of bandwidth inheritance with the greedy capacity sharing and stealing policy of CSS. Rather than trying to account borrowed capacities and exchanging them later in the exact same amount, CXP focus on greedily exchanging extra capacities as early, and not necessarily as fairly, as possible in order to effectively minimises the degree of deviation from the ideal system's behaviour caused by inter-application blocking.

Resumo

À medida que um número crescente de utilizadores pretende executar simultâneamente aplicações com e sem restrições de tempo real no mesmo dispositivo computacional a questão de como fornecer uma utilização eficiente de recursos num tal ambiente dinâmico, aberto e partilhado é cada vez mais importante. A necessidade surge do facto de aplicações independentemente desenvolvidas poderem entrar e sair do sistema a qualquer momento sem que exista um conhecimento prévio das suas reais necessidades em termos de recursos e periodicidade de chegada das suas tarefas mas, mesmo assim, ser necessário responder aos eventos dentro de determinadas restrições temporais de modo a garantir um nível desejado de performance. Dentro deste contexto, esta tese propõe uma *framework* cooperativa de gestão da qualidade de serviço (QoS), denominada *CooperatES* (*Cooperative Embedded Systems*), que permite que dispositivos com restrições computacionais possam executar colectivamente serviços com os seus vizinhos.

A *framework* proposta permite satisfazer requisitos não funcionais que de outra forma não poderiam ser garantidos através de uma execução individual. Os dispositivos agrupam-se dinamicamente em novas coligações, definindo configuração iniciais de serviço que maximiza a satisfação das restrições de qualidade impostas pelo utilizador para o novo serviço e minimizam o impacto da chegada desse serviço no nível de qualidade global. Durante a execução da coligação, a gestão dinâmica da qualidade de serviço dos diversos serviços que concorrem entre si pelos recursos do sistema é feita sob o controlo do utilizador, estendendo a sua influência não só à formulação da coligação mas também à sua operação.

A abordagem tradicional de optimização da QoS, centrada na procura de uma única solução óptima ou com um nível de qualidade não óptimo pré-definido, é reformulada como uma optimização *anytime* baseada em heurísticas que pode ser interrompida a qualquer momento e, mesmo assim, capaz de devolver uma solução, mesmo quando quando os serviços exibem relações de interdependência entre as suas tarefas. A abordagem *anytime* proposta é capaz de encontrar rapidamente uma boa solução

inicial e optimiza o incremento da qualidade dessa solução a cada iteração, com um custo que pode ser negligenciado quando comparado com os benefícios introduzidos.

As adaptações autónomas de cada dispositivo durante a execução são coordenadas por um modelo descentralizado baseado num mecanismo de *feedback* para reduzir a complexidade das interacções necessárias entre os nós até que um comportamento de adaptação global seja determinado. *Feedback* positivo é usado para reforçar a selecção de um novo nível de qualidade desejado, enquanto que o *feedback* negativo desencoraja os nós a actuar de modo egoísta uma vez que essa atitude tem um impacto negativo no nível de qualidade fornecido pelos nós vizinhos. Através da troca de *feedback* às acções de adaptação individuais os nós convergem para uma solução global, mesmo que isso signifique não fornecer as suas soluções individuais óptimas. Como resultado, cada nó, apesar de autónomo, é influenciado, e pode influenciar, o comportamento de outros nós numa coligação.

As mudanças dinâmicas dos requisitos dos serviços é gerida de um modo previsível, impondo restrições temporais com um certo grau de flexibilidade, obtendo o desejado equilíbrio entre *performance* previsível e um uso eficiente dos recursos do sistema. O escalonador CSS (*Capacity Sharing and Stealing*) proposto, um algoritmo dinâmico baseado em servidores com prioridades dinâmicas, suporta a coexistência de servidores com reservas garantidas e não garantidas de tempos de computação de modo a gerir eficientemente as sobrecargas computacionais usando capacidade computacional extra a partir de duas fontes: (i) capacidade residual alocada mas não usada quando as tarefas terminam antes do seu tempo de execução pré-reservado; (ii) roubando capacidade aos servidores não isolados inactivos, usados para escalonar tarefas não periódicas de gestão da *framework* baseadas numa política de melhor esforço.

A eficiência e reduzida complexidade do CSS na gestão de capacidades reservadas em ambientes onde não existe um conhecimento prévio das reais necessidades dos serviços em execução torna-o apropriado a ser usado como base de um escalonador mais poderoso, capaz de gerir interdependências entre tarefas, nomeadamente partilha de recursos e restrições de precedência. Esta tese propõe o escalonador CXP (*Capacity Exchange Protocol*), um protocolo que integra o conceito de herança de capacidades com a gestão egoísta de partilha e roubo de capacidades computacionais do CSS. Em vez de tentar contabilizar as capacidades obtidas por empréstimo e restituí-las mais tarde exactamente na mesma medida, o CXP centra-se na troca egoísta das capacidades extra disponíveis tão cedo, e não necessariamente tão equitativamente, quanto possível de modo a minimizar o grau de desvio em relação ao comportamento ideal do sistema causado pelo bloqueio entre tarefas.

Resumé

Comme un nombre croissant d'utilisateurs tourne à la fois en temps réel et non tempsréel des applications dans le même système, la question de savoir comment fournir une utilisation efficace des resources dynamiques ouvertes et dans un evironment partagé est très importante. La nécessité découle du fait que des services indépendants développés peuvent entrer et sortir du système à tout moment sans aucune connaissance de leurs véritables besoins et l'exécution des táches inter-les heures d'arrivée mais, néanmoins, la résponse à des événements doit encore être fournie dans un timing contraint afin de garantir un certain niveau de performance.

Cette thèse propose un cadre QoS-aware coopératif, appelé COOPERATES (Cooperative Embedded Systems), qui permet à des dispositifs limités d'exécuter collectivement des services avec de plus puissants voisins et qui rencontre les exigences nonfonctionnelles qui, autrement, ne seraient pas atteints par une exécution. Des nœuds d'un environnement distribué de coopération dynamique s'associent eux-mêmes dans une nouvelle coalition, en allocant des ressources pour chaque nouveau service et en établissant une première configuration qui maximise la satisfaction des contraints QoS que s'associe à ce nouveau service et minimise l'impact sur le QoS global causé par l'arrivée du nouveau service.

Cette thèse reformule l'approche d'une optimisation traditionnelle QoS, concentrée surtout sur la recherche optimale seule ou sur des solutions sous-optimalisé liées, comme une base heuristique sur l'optimisation que à tout moment peut être interrompue et réagir rapidement aux changements de l'environnement, en adaptant le service distribué à la délibération du temps disponible qui est imposée dynamiquement en raison des nouvelles conditions environnementales. Les algorithmes sont proposés à tout moment en mesure de trouver rapidement une bonne solution initiale au service de manière à optimiser la vitesse dans laquelle la qualité de la solution actuelle s'améliore à chaque itération de l'algorithme, avec un tableau qui puisse être considéré comme négligeable quand comparé contre les avantages introduites.

Cette thèse propose une démarche décentralisée, un modèle de coordination sur la base d'un mécanisme de rétroaction efficace pour réduire la complexité des interactions nécessaires entre les noeuds jusqu'à ce que l'adaptation du comportement collectif est déterminée. La rétroaction positive est utilisé pour renforcer la sélection de la nouvelle solution globale des services souhaités, tandis que la rétroaction négative décourage les nœuds d'agir d'une manière vorace effets négatifs sur les niveaux de service fournis à des noeuds voisins. En échangeant des informations sur les actions auto-adaptative, les noeuds convergent vers une solution globale, même si cela implique de ne pas fournir de meilleures solutions individuelles. En conséquence, chaque nœud, malgré que autonome, est influencé et peux influer sur le comportement des autres noeuds du système.

Au même temps, une nouvelle approche de planification est nécessaire pour gérer l'évolution dynamique des services et les besoins d'une façon prévisible, en exécutant de contraintes temporelles avec un certain degré de flexibilité, visant à réaliser les compromis entre les performances prévisibles et l'utilisation efficace des ressources. Par conséquent, cette thèse intègre et prolonge des avances récents en deadline dynamique, en planifiant avec des ressources de réservation, en proposant le gestionnaire CSS (Capacity Sharing and Stealing), un gestionnair basé sur serveur dynamique basé qui supporte la coexistence des serveurs de bande passante, garanties et non-garanties, de manière à traiter de une forme efficace les tâches-soft surchargées en permettant une capacité supplémentaire de deux sources: (i) capacité résiduelle alloués mais non utilisés lors de l'emploi complet en moins de leur temps d'exécution, (ii) la capacité des serveurs inactifs, non-isolés de voler utilisé pour planifier les meilleurs efforts de gestion des tâches apériodiques.

L'efficacité et la diminution de la complexité des CSS en manipulant les capacités réservées et inutilisées sans aucune connaissance préalable sur les tâches d'exécution de comportement, sont appropriés pour être utilisé comme la base d'un gestionnaire puissant capable de gérer des ensembles de tâches dépendantes qui partagent l'accès entre certaines ressources du système et d'exposer les contraintes précédentes. Le proposé CXP (Capacity Exchange Protocol) intègre le concept d'héritage des bandes passantes avec la capacité vorace de partager et voler la politique des CSS. Plutôt que d'essayer d'emprunter les capacités et de les échanger plus tard dans le même montant, CXP se concentre sur les capacités extra voraces d'échange au plus tôt possible. Cet approche réduit effectivement le degré de déviation du comportement idéal du système causé par le blocage inter-application.

Contents

Abstract **5**

Resumo **7**

Resumé **9**

List of Tables **15**

List of Figures **18**

1 Introduction **19**

1.1 Main concepts 19

1.2 Motivation 23

1.3 Contributions 25

1.4 Outline 26

2 Adaptive real-time systems **29**

2.1 Introduction 29

2.2 Quality of Service (QoS) management 31

2.3 Adaptive middleware 38

2.4 Real-time scheduling 43

2.5 Summary 50

3 Cooperative embedded systems **51**

3.1 Introduction 51
3.2 Problem description and system model 53
3.3 Expressing Quality of Service 55
3.4 The CooperatES framework 59
3.5 Coalition formation 63
3.6 Service proposal formulation 66
3.7 Supporting runtime QoS adaptation and stability 69
3.7.1 Promised stability periods 71
3.7.2 Determine possible upgrades of previously downgraded Service Level Agreements (SLAs) 72
3.8 Summary 74

4 Time-bounded service configuration 75

4.1 Introduction 75
4.2 Anytime algorithms 77
4.2.1 An anytime QoS optimisation approach 78
4.3 Anytime coalition formation 81
4.3.1 Formal description of the algorithm's anytime behaviour 82
4.3.2 Conformity with the desirable properties of anytime algorithms 84
4.4 Anytime service proposal formulation 87
4.4.1 Formal description of the algorithm's anytime behaviour 90
4.4.2 Conformity of with the desirable properties of anytime algorithms 91
4.5 Anytime upgrade of previously downgraded SLAs 94
4.5.1 Formal description of the algorithm's anytime behaviour 95
4.5.2 Conformity with the desirable properties of anytime algorithms 97
4.6 Summary 99

5 Scheduling tasks in open systems 101

5.1 Introduction 101

5.2 System model . . . 103
5.3 The Capacity Sharing and Stealing (CSS) approach . . . 104
5.3.1 The CSS scheduler . . . 108
5.3.2 Handling overloads with CSS . . . 109
5.4 Theoretical validation for independent tasks . . . 111
5.5 Summary . . . 114

6 Handling QoS inter-dependencies 115
6.1 Introduction . . . 115
6.2 Local QoS inter-dependencies . . . 117
6.2.1 Anytime service proposal formulation for inter-dependent task sets . . . 119
6.2.2 Anytime re-upgrade of previously downgraded levels of service for inter-dependent task sets . . . 121
6.3 Distributed QoS inter-dependencies . . . 123
6.3.1 A one-step decentralised coordination model . . . 125
6.3.2 Properties of the proposed decentralised coordination model . . . 129
6.3.2.1 Coordinating upgrades . . . 130
6.3.2.2 Coordinating downgrades . . . 132
6.3.3 Number of exchanged messages . . . 134
6.4 Summary . . . 135

7 Scheduling inter-dependent task sets 137
7.1 Introduction . . . 137
7.2 System model . . . 138
7.3 Sharing resources in open systems . . . 140
7.4 The Capacity Exchange Protocol (CXP) . . . 142
7.4.1 Minimising the cost of blocking with CXP . . . 143
7.5 Handling precedence constraints in open systems . . . 144
7.5.1 Handling tasks' precedences with CXP . . . 146

7.6 Theoretical validation for dependent tasks . . . 147
7.6.1 Blocking time computation . . . 152
7.7 Summary . . . 153

8 Evaluation **155**
8.1 Introduction . . . 155
8.2 Evaluated scenario . . . 156
8.3 Anytime approach's behaviour and overhead . . . 158
8.3.1 Coalition formation . . . 158
8.3.2 Service proposal formulation . . . 160
8.3.3 Services' runtime adaptation . . . 163
8.4 Coordinating distributed inter-dependent adaptations . . . 166
8.5 Efficiency of the proposed scheduling algorithms . . . 169
8.5.1 Residual capacity reclaiming . . . 169
8.5.2 Allowing capacity stealing . . . 171
8.5.3 Sharing resources among tasks . . . 173
8.5.4 Imposing precedence constraints among tasks . . . 177
8.6 Summary . . . 178

9 Conclusion **179**
9.1 Introduction . . . 179
9.2 General conclusions . . . 181
9.3 Summary of the main contributions . . . 181
9.4 Future research directions . . . 185

Bibliography **187**

List of Tables

4.1 Iterative QoS optimisation 90

8.1 Possible characteristics of nodes 157

List of Figures

3.1 Framework structure 60

3.2 Resource managers' layering 62

3.3 QoS Provider 62

3.4 Acceptable service quality 63

5.1 State transitions of CSS servers 105

5.2 Handling overloads with CSS 110

7.1 BWI's drawbacks 141

7.2 Sharing resources with CXP 144

7.3 Handling tasks' precedences with CXP 147

8.1 Coalition formation: Anytime behaviour 159

8.2 Coalition formation: Anytime vs Traditional 159

8.3 Proposal formulation: Anytime behaviour with spare resources 161

8.4 Proposal formulation: Anytime behaviour with limited resources 161

8.5 Proposal formulation: Anytime vs Traditional with spare resources . . 162

8.6 Proposal formulation: Anytime vs Traditional with limited resources . . 163

8.7 QoS re-upgrade: Anytime behaviour 164

8.8 QoS re-upgrade: users' influence 164

8.9 QoS re-upgrade: Anytime vs Traditional 165

8.10 Average number of exchanged messages 166

8.11 Needed time until the global adaptation result is determined 167

8.12 Relative solution's utility as a function of available resources 168
8.13 Performance in dynamic scenarios 170
8.14 Small variation in execution times 172
8.15 Large variation in execution times 172
8.16 Capacity consumed by the SP task 174
8.17 Capacity consumed by the LP task 174
8.18 Performance in dynamic scenarios 175
8.19 Overhead using BWI as reference 176
8.20 Overhead of handling precedence constraints 177

Chapter 1

Introduction

Traditionally, most real-time systems were built to achieve a specific set of goals and the set of tasks to be executed was well known at design time. In contrast to this, real-time systems are undoubtedly becoming more open and increasingly support a broader spectrum of soft real-time applications characterised by strongly varying resource requirements only known at runtime. Nevertheless, a timely answer to events must still be provided in order to guarantee a desired level of performance.

The challenge is how to efficiently execute applications in these new open real-time systems while meeting non-functional requirements arising from the operating environment, the users, and applications. This thesis advocates that such demands are adequately addressed through a cooperative decentralised QoS-aware execution model, supported by anytime QoS optimisation algorithms and effective flexible scheduling mechanisms.

This chapter discusses some important concepts referred to throughout this document, introduces the problem it aims to solve, presents its main contributions, and outlines the rest of the document.

1.1 Main concepts

Throughout this thesis, the concepts of *open real-time system*, *embedded system*, and *Quality of Service* appear very frequently. Considering the broad use of these terms in different research and industrial communities, we decided to provide a number of definitions that, hopefully, will help the reader to better understand the rest of this document.

Real-time system. A real-time system is a system whose performance depends not only on the values of its outputs, but also on the time at which these values are produced [Sta88].

Real-time systems span a large spectrum of activities. Examples include nuclear plants supervision, military command and control, medicine and emergency response, avionics and air traffic control, multimedia, mobile phone networks, real-time databases, robotics, and agile manufacturing systems. In all these scenarios, *time* is the basic constraint to deal with.

A common misconception is to consider a real-time system as a fast system. The time scale may vary largely. Its magnitude can be a microsecond in a radar, a second in a human-computer interface, a minute in an assembly line, or an hour in a chemical reaction.

No matter how fast a computer is, its performance must always be guaranteed against the characteristics of the environment. The most important feature for a real-time system is not speed but predictability. Typically, in a system with several concurrent activities, high-speed tends to maximise the average performance of the task set, whereas a predictable behaviour aims at guaranteeing the individual timing constraints of each task.

Depending on the consequences of missing timing constraints, real-time tasks are usually distinguished into hard and soft [1]. A real-time task is said to be hard if missing a single deadline may cause catastrophic consequences on the controlled system. A real-time task is said to be soft if missing one or more deadlines does not jeopardise the correct system behaviour, but only causes a performance degradation. For soft real-time systems, the goal is typically to meet some Quality of Service requirements.

Open real-time system. A real-time system is defined "open" if independently developed applications can be activated and terminated at any moment, generating a time-varying workload [DLS97].

One challenge facing the real-time systems community is how to build and deliver open real-time systems where a dynamic mix of independently developed applications with different timeliness constraints can coexist. Some of the difficulties arise from the fact that both resource availability and the mix of applications and their aggregate

[1] More complete classifications can be found in the literature. However, this simple classification suffices to explain the point in question

resource and timing requirements are unknown until runtime, but, still, a timely answer to events must be provided in order to guarantee a desired level of performance.

This calls for a somewhat different and more flexible approach than those typically used today for building fixed-purpose real-time systems. Classical methodologies for real-time systems design are hardly applicable, because they assume a prior knowledge both on the applications' requirements and on the availability of resources.

A promising approach is to design open systems using Quality of Service negotiation and runtime adaptation techniques in order to ensure that, despite the uncertain factors that trigger the occurrence of changes in the environment, service is still provided within contracted levels.

Quality of Service. The collective effect of service and performances that determine the degree of the user's satisfaction [MR02].

QoS is apparent at all layers in any architecture, but is "viewed" differently by each layer. The end user is concerned with perceptual QoS that essentially defines how "good" a service appears to the user, together with non-performance related QoS such as cost. This satisfaction is usually associated with a number of non-functional requirements or QoS characteristics, such as dependability, reliability, timeliness, robustness, throughput, etc.

User QoS is also the least service specific representation (in terms of how dependent upon the type of service the QoS characteristic is) and is produced as a direct effect of application performance and the supporting environment.

There is a fine line between *Application QoS* and *User QoS*, but it is probably best differentiated by the fact that the former is usually in terms of computational concepts where *User QoS* is not.

System QoS defines the QoS expected by the application of the underlying system and incorporates everything between the application and the hardware devices. It is more service specific since its QoS characteristics are directly relevant to the type of service.

Finally, at the lowest level, QoS is specified with respect to device capabilities, either peripheral or network. These QoS specifications, termed *Device QoS*, are usually very detailed and often concerned with the performance a particular device can offer.

Quality of Service management. The process of controlling the performance of a system as a function of workload variations [ART04].

QoS management is the complete set of activities related to the control and administration of QoS within a system. Often the term is used in close relation to resource management, since QoS is often some complex function of end-to end resource utilisation.

Resource allocation mechanisms are typically activated during the establishment phase. However, resources may be subject to re-allocation as a result of QoS violations and degradation, or re-negotiations in response to changing service requirements. Admission control mechanisms limit the acceptance of requests in order to ensure that resources are not overloaded or that existing timing constraints are not disrupted.

Embedded system. A device (and its software) is considered embedded if it is an integral component of a larger system [Emb].

Traditionally, an embedded system was a special-purpose computer system designed to perform one or a few dedicated functions, often with real-time computing constraints. It was usually embedded as part of a complete device including hardware and mechanical parts.

Nowadays, embedded systems span all aspects of modern life and there are many examples of their use. Telecommunications systems employ numerous embedded systems from telephone switches to mobile phones at the end-user. Computer networking uses dedicated routers and network bridges to route data.

Consumer electronics include personal digital assistants (PDAs), mp3 players, mobile phones, video-game consoles, digital cameras, DVD players, GPS receivers, and printers. Many household appliances, such as microwave ovens, washing machines and dishwashers, are including embedded systems to provide flexibility, efficiency and features. Advanced air conditioning systems use networked thermostats to more accurately and efficiently control temperature that can change by time of day and season. Home automation uses wired and wireless networking that can be used to control lights, climate, security, audio/visual, surveillance, etc., all of which use embedded devices for sensing and controlling.

Transportation systems from flight to automobiles increasingly use embedded systems to reduce costs, maximise efficiency, and reduce pollution. New air-planes contain advanced avionics such as inertial guidance systems and GPS receivers that also have considerable safety requirements. Automotive safety systems, such as anti-lock braking system (ABS), Electronic Stability Control (ESC/ESP), traction control (TCS), and automatic four-wheel drive are increasingly using microprocessors as a core system component instead of using dedicated hardware.

Medical equipment is continuing to advance with more embedded systems for vital signs monitoring, electronic stethoscopes for amplifying sounds, and various medical imaging systems for non-invasive internal inspections.

Most of these embedded systems share a number of important properties, such as: (i) *limited resources*, due to cost constraints related with mass production and strong industrial competition. In order to make devices cost-effective, it is mandatory to make a very efficient use of the computational resources; (ii) *demanding quality requirements*. Users of consumer electronics products, home appliances, and mobile devices are accustomed to robust and well behaving devices. It is obvious that this requirement will not be relaxed because of the usage of processors in their construction; and a (iii) *tight interaction with the environment*. An embedded system acts within, and in many cases on, the physical environment, which requires the system to react to events within timing constraints.

1.2 Motivation

Traditionally, embedded systems have been able to rely on their closed environments to limit the possible inputs and on static resource management techniques to supply pre-defined Quality of Service (QoS) levels [KRP+93].

However, nowadays' embedded systems combine the stringent QoS requirements of traditional closed embedded systems with the challenges of an open, heterogeneous, and dynamic environment [HLR01, GRHL04]. Open systems are inherently uncertain and dynamic and accurate optimisation models are difficult to obtain and quickly become outdated. Nevertheless, despite their uncertainty, responses to events still have to be provided within timing constraints in order to guarantee a desired level of performance. Achieving the necessary predictable real-time behaviour relies on the ability to dynamically manage resources by adapting and reconfiguring the set of provided levels of service to the dynamically changing environmental conditions.

However, even if the system has this ability, an individual embedded device may not have sufficient resources to deliver the minimum desired quality to every application along each of its QoS characteristics [SCZ05, EEGL03, SLSL05]. A redistribution of the computational load across a set of computational devices (hereafter called neighbour nodes) would then enable the execution of far more complex and resource-demanding services that otherwise would be able to be executed on a stand-alone basis.

With the rapid development of embedded technology, cooperative computing, which enables large-scale resource sharing and collaboration, emerges as a promising distributed computing paradigm to face the stringent demands on resources and performance of new embedded real-time systems. Service partitioning and offloading to a remote machine has been successfully proposed for power and performance gains [GMG+04, KHR01, LWX01, OH98, RRPK98, WL04]. These works conclude that the efficiency of an application execution can be improved by careful partitioning the workload between a resource constrained device and a fixed, more powerful, neighbour.

Nevertheless, it is known that users might tolerate different levels of service, or could be satisfied with different quality combination choices [RLLS97] and, to the best of our knowledge, none of the works on computation offloading takes that into consideration. We believe that supporting each user's specific QoS preferences while offloading computation is a key issue but there is still no method for distributing a resource intensive service by the subset of neighbour nodes which offers the best QoS according to a particular user's service request.

In addition, QoS optimisation techniques have traditionally been mainly concentrated on finding single optimal or with a fixed sub-optimality bound solutions. However, the increased complexity of open real-time environments may prevent the possibility of computing both optimal local and global resource allocations within an useful and bounded time. Anytime algorithms have shown themselves to be particularly appropriate in such settings, incorporating the notion that the needed computation time to obtain optimal service solutions will typically reduce the overall solution's utility [Zil96].

At the same time, embedded real-time systems are becoming increasingly unpredictable due to the increasing use of independently developed data-driven applications whose actual execution parameters vary significantly with input data and cannot be predicted in advance [AB98]. While several scheduling solutions have already been proposed to achieve guaranteed service and inter-task isolation, unused reserved capacities can be more efficiently used to meet deadlines of tasks whose resource usage exceeds their reservations. Isolation can be reduced in a controlled fashion in order to donate reserved, but still unused, capacities to overloaded servers, handling overloads with additional computational capacity wasted by the currently available bandwidth reservation schedulers.

Furthermore, most of the existing work on overrun control is proposed under the assumption that soft and hard real-time tasks are independent. However, tasks are rarely independent in real world embedded systems. As such, it is important to

propose an efficient scheduling technique to support shared resources and precedence constraints among tasks of open real-time systems, without compromising the real-time guarantees of hard tasks.

As such, the implementation of adaptive embedded real-time systems, operating in open and dynamic environments, requires several issues to be considered at the same time: (i) a cooperative execution of resource intensive services; (ii) time-bounded QoS management mechanisms; and (iii) more efficient scheduling strategies.

1.3 Contributions

This thesis proposes the fundamental basis of a real-time cooperative framework that can provide a more efficient and predictable support to the development of quality-aware embedded systems, characterised by high complexity, dynamic behaviour, and distributed organisation.

The central proposition of this thesis is that heterogeneous, dynamic, and open real-time embedded systems are adequately built using a cooperative decentralised model, supported by anytime QoS optimisation algorithms and effective flexible scheduling mechanisms.

The CooperatES (Cooperative Embedded Systems) framework [PNB05, NP06a] facilitates the cooperation among neighbour devices when a particular set of user-imposed QoS constraints cannot be satisfyingly answered by a single node. Nodes dynamically group themselves into a new coalition, allocating resources to each new service and establishing an initial Service Level Agreement (SLA) that maximises the satisfaction of the user's QoS constraints under negotiation [NP05].

The increased complexity of open real-time environments may prevent the possibility of computing optimal local and global resource allocations within a useful and bounded time. As such, the QoS optimisation problem is here reformulated as a heuristic-based anytime optimisation problem that can be interrupted at any time and quickly respond to environmental changes [NP06c, NP09b]. The proposed anytime algorithms are able to quickly find a good initial service solution and effectively optimise the rate at which the quality of the current solution improves at each iteration of the algorithms, with an overhead that can be considered negligible when compared against the introduced benefits.

Runtime adaptation is a fundamental issue in resource-constrained QoS-aware systems since it determines how well users' service requests are satisfied in the presence of

dynamically changing operating conditions. As such, during a coalition's lifetime, the initially promised SLA may be downgraded in order to accommodate new service requests with a higher utility for the system or re-upgraded when the needed resources become once again available. However, while some users or applications may prefer to always get the best possible instantaneous QoS independently of the reconfiguration rate of their requested services, others may find that frequent QoS reconfigurations are undesirable. This thesis explores these ideas and proposes a QoS adaptation mechanism that allows users to control the runtime adaptation behaviour of their applications [NP06b, NP08a].

While runtime adaptation is widely recognised as valuable, adaptations in most existing systems are limited to changing independent execution parameters. This thesis provides support for a more realistic solution that considers (i) runtime adaptations that span multiple dependent components at one node [NP08b, NP08a]; and (ii) a one-step decentralised coordination model that reduces the complexity of the needed interactions among nodes until a collective adaptation behaviour is determined, whenever the autonomous self-adaptations have an effect on other nodes in a coalition [NP09a].

Based upon a careful study of the ways in which unused reserved capacities can be more efficiently used to meet deadlines whenever a task needs to exceed its reserved amount of computation time, this thesis also proposes the Capacity Sharing and Stealing (CSS) scheduler [NP07a]. CSS integrates and extends some of the best principles of previous scheduling approaches to improve the responsiveness of soft real-time tasks in the presence of overruns while ensuring that the schedulability of hard tasks is not compromised.

Since this thesis considers possibly inter-dependent task sets, a novel scheduling strategy for supporting shared resources and precedence constraints among tasks of open real-time systems is also proposed. The Capacity Exchange Protocol (CXP) [NP07b, NP08c] merges the benefits of CSS with the concept of bandwidth inheritance to allow a task to be executed on more than its dedicated server, efficiently exchanging capacities among servers and reducing the undesirable effects caused by inter-application blocking.

1.4 Outline

The rest of this document is organised as follows:

Chapter 2 is devoted to discuss specific topics relevant for the development of adaptive real-time embedded systems directly related to the main contributions of this thesis.

Chapter 3 proposes cooperative computing as a valuable solution for addressing the increasing needs for resources and performance in modern embedded real-time systems. It describes the main architecture of the CooperatES framework and explores utility-based resource allocation and adaptation policies. These take into consideration the increasing demand for customisable service provisioning tailored to each user's specific QoS preferences and needs.

Chapter 4 discusses a novel anytime approach to deal with a large number of dynamic tasks, multiple resources, and real-time operation constraints in open dynamic real-time systems. The algorithms proposed in Chapter 3 are reformulated as a heuristic-based anytime optimisation approach in which there is 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.

Chapter 5 presents CSS, a novel approach to dynamic server-based scheduling in open real-systems. CSS supports the coexistence of guaranteed and non-guaranteed bandwidth servers to efficiently handle soft-tasks' overloads by making additional capacity available from two sources: (i) residual capacity allocated but unused when jobs complete in less than their budgeted execution time; (ii) stealing capacity from inactive non-isolated servers used to schedule best-effort jobs.

Chapter 6 addresses the problem of QoS dependencies among both local and distributed tasks of open real-time systems. It starts by extending the local anytime QoS optimisation algorithms proposed in Chapter 4 to handle adaptations of services that share resources and whose execution behaviour and input/output qualities are interdependent. It continues by proposing a decentralised coordination model for groups of autonomous distributed nodes whose system-wide behaviour is established and maintained through local interactions. Each node is able to get information about its nearest dependent neighbours and then complement its partial knowledge of the global adaptation problem with the state of its own resources. As a result, each node, although autonomous, is influenced by, and can influence, the behaviour of other nodes in the system.

Chapter 7 presents CXP, addressing the challenging problem of how to schedule tasks that share resources or exhibit precedence constraints without any previous information on critical sections and computation times. While preserving the isolation principles of independent tasks and inheritance properties of critical sections, CXP

introduces significant improvements in the system's performance through a greedy capacity exchange policy that takes advantage of all the available unused computation capacity.

Chapter 8 evaluates the behaviour of the several algorithms proposed in this thesis through both quantitative and qualitative analysis.

Finally, Chapter 9 offers some concluding remarks and suggests some areas of future work.

Chapter 2

Adaptive real-time systems

While early research on real-time computing was concerned with guaranteeing avoidance of undesirable effects such as overloads and deadline misses during the system's development, adaptive real-time systems are designed to dynamically handle such situations at runtime. This chapter discusses representative research efforts on that direction directly related to the main contributions of this thesis.

2.1 Introduction

Real-time computing systems were originally developed to support safety critical, mission critical, or business critical control applications characterised by stringent timing constraints and, indeed, much of embedded computing is still for these types of applications. In these systems, missing a single deadline can jeopardise the entire system behaviour or even cause catastrophic consequences. Hence, they need to be designed under worst-case assumptions, identified through a static design before the system is deployed, and executed with predictable kernel mechanisms to meet the required performance in all anticipated scenarios, thus ensuring a correct behaviour and eliminating changes during the system's operation.

While the high cost of such approach is acceptable for applications with dramatic failure consequences, it is no longer justified in a growing number of new embedded systems, in areas such as multimedia, automotive information and entertainment systems, mobile phone networks, robotics, and radar tracking, which, instead of a strict hard real-time behaviour for the entire system, only demand some temporal control in order to be accepted by their users. A deadline miss does not cause a

system or application failure but it is only less satisfactory for the user and, as such, reserving resources based on average needs is generally regarded as cheaper and more flexible.

The challenge is how to efficiently execute applications in these new embedded systems while meeting non-functional requirements, such as timeliness, robustness, dependability, performance etc. In fact, in order to satisfy a set of constraints related to weight, space, and energy consumption, most of these systems are typically built using small microprocessors with low processing power and limited resources.

This is where QoS management applies. It seems evident that an application cannot provide stable QoS characteristics if it has not some guarantees on the available amount of resources. As such, reserving resources is basic for supporting QoS mechanisms. The operating system or middleware reserves a portion of the system's resources to an application, which then has to provide a predefined stable output quality. This has been precisely the goal of years of research on real-time scheduling and schedulability analysis: to ensure that there are enough resources (CPU, network bandwidth, etc.) for meeting time requirements.

However, the move from self-enclosed to open real-time embedded systems is also one of moving from static to dynamic environments. Open real-time systems allow a mix of independently developed real-time and non real-time applications to coexist in the same system. As such, the set of applications to be executed and their aggregate resource and timing requirements are unknown until runtime, which implies that perfect *a priori* schedulability analysis is impossible. At the same time, users increasingly consider QoS as important as functionality, that is, how well an application performs its function is as important as what it does.

In such scenario, static resource allocations might be appropriate for a situation at a single point in time, e.g. initial deployment, but quickly become insufficient as conditions change. When reasoning in terms of QoS support in dynamic environments, which implies the establishment of contracts between clients and service providers, the idea is to design systems using QoS adaptation and renegotiation techniques and ensuring that, despite the uncertain factors that trigger the occurrence of changes in the environment, the QoS contracts remain valid.

Recent research activities in this field have covered the following topics: (i) scheduling mechanisms that define the execution of competing tasks at runtime; (ii) feedback-based management strategies to cope with scarcely known or time-varying execution requirements of tasks; and (iii) architectural solutions for operating systems and middleware to support the technologies described above.

This chapter is devoted to discuss these topics, relevant for the development of adaptive real-time embedded systems and directly related to the main contributions of this thesis. Section 2.2 discusses the different approaches that can be used for controlling QoS in a real-time embedded system. Section 2.3 is focused in real-time adaptive middleware, the software layer provided above the operating system to facilitate the development of distributed applications. Whether a set of real-time tasks can meet their deadlines depends on the characteristics of the tasks and the scheduling algorithm used. Section 2.4 is devoted to discuss several approaches in designing real-time scheduling algorithms.

2.2 Quality of Service (QoS) management

The main reason for a QoS management layer in real-time embedded systems is to provide flexibility for systems and environments where requirements on resources are inherently unstable and difficult to predict in advance. Such a difficulty is due to different causes. First of all, modern computer architectures include several low-level mechanisms that are designed to enhance the average performance of applications but, unfortunately, introduce high variations on tasks' execution times [CP03]. In other situations, as in multimedia systems, processes can have highly variable execution times that also depend on input data.

Performing an efficient QoS management requires specific support at different levels of the system's architecture. Today's embedded real-time systems are dynamic inter-operating systems with the need for a QoS management that is [SLS+06]: (i) *multi-layered*, basing low level allocations of resources and control behaviour on high-level system goals; (ii) *aggregate*, mediating and managing the conflicting QoS needs across competing applications and users; and (iii) *adaptive*, adjusting QoS provision on the fly as situations, conditions, and needs change. Hence, new software methodologies are emerging in embedded systems, which strictly relate to real-time operating systems, middleware, and networks.

Several researchers have focused on developing mechanisms for operating systems that provide soft real-time application support by allowing applications to miss some or all of their deadlines. These systems generally rely on scheduling primitives that use application-supplied information concerning each application's CPU and timing requirements. When all of the application resource requirements cannot be met, these systems tend to reject additional applications or stop low-priority tasks that are already executing [MST94]. Other approaches are explicitly guided by the user in

selecting which application to eliminate when resource availability falls below a threshold point [CT94]. However, these methods do not support the principle of graceful degradation wherein all users' requests are honoured and resources are allocated to the applications in an equitable manner.

Other systems reduce the resources provided to each application based on formulas of resource needs or application importance [Fan95, JB95, NL97, JRR97]. These systems rely on the applications themselves to adjust their processing to fit within the resources they have been allocated. However, none of these latter systems provide a model for how the applications can reduce their resource requirements to function within a less-than-optimal allocation.

To speed time-to-market and reduce costs, embedded system developers increasingly rely on real-time operating systems that reduce the cost of computer-based automation and control systems by adopting cost-effective hardware and software. At the time of this writing, there were 101 commercial real-time operating systems listed and described in [Emb]. A commercial real-time operating system is generally chosen not only for its real-time characteristics, but also for its file system and communication stack, for its portability, and for the associated cross-development environment. It is usually marketed as the runtime component of an embedded development platform, which also includes a comprehensive suite of development tools and utilities and a range of communications options for the target connection to the host, in an Integrated Development Environment (IDE). Currently, VxWorks [Rivb], from Wind River, is the major commercial real-time operating system.

There has also been a considerable amount of work in making Linux a real-time operating system. A list of Linux real-time variants can be found at [Fou]. One can distinguish two basic approaches: (i) to use a small real-time executive as a base and execute Linux as a thread in this executive; and (ii) to directly modify the Linux internals. RTLinux [Riva] and RTAI [DdIA] are examples of the first approach, whereas Linux RK [RtMSL] is an example of the second approach.

At the network level, a lot of research has been conducted on the end-system or end-to-end architectures for QoS support, and also on link, network, and transport layers. Scheduling algorithms for package deliberation provide specific quality levels [CSZ92], while resource reservation protocols such as the Resource ReSerVation Protocol (RSVP) [ZDE+93] provide support for end-to-end resource reservation for specific sessions. DiffServ and IntServ [BCS94] are examples of IETF standards that integrate RSVP for the support of real-time as well as the current non real-time service in IP networks. Each protocol defines different approaches for the classification of network

traffic, services, and interfaces for their support. IntServ was especially ambitious for the support of Internet real-time systems (remote video, multimedia and virtual reality). DiffServ is a layer 3 traffic-handling mechanism that tries to reduce the complexity of IntServ and include new services such as SLA (Service Level Agreement), which specify the amount of customer traffic that can be accommodated at each service level.

The Real-Time Transport Protocol (RTP) [SCFJ96] is a transport protocol for carrying real-time traffic flows in an IP network. It provides a standard packet header format which gives sequence numbering, media-specific time stamp data, source identification, and payload identification, among other things. RTP is usually carried using UDP. RTP is supplemented by the Real-Time Transfer Control Protocol (RTCP), which carries control information about the current RTP session. RTP does not address the issue of resource reservation but relies on reservation protocols such as RSVP.

The Resource Negotiation and Pricing (RNAP) protocol and architecture [WS00], based on pricing mechanisms, has been proposed as a framework to enable a user to select from a set of available network services with different QoS characteristics, and enable the user and network to dynamically re-negotiate the contracted service parameters and price. RNAP has some features and goals in common with the work on differentiated services [NBBB98] and RSVP [ZDE+93]. RNAP is intended for use by both adaptive and non-adaptive applications. Non-adaptive applications may choose services that offer a static price, or absorb any changes in price while maintaining their sending rate. Adaptive applications adapt their sending rate and/or choice of network services in response to changes in network service prices.

However, most of this research has been focused on low-level system mechanisms. While individual resource management is an important factor for an efficient QoS management, we believe that, by itself, it is not sufficient for the ultimate end-users who experience the resulting QoS. It is known that different users might tolerate different levels of service, or could be satisfied with different quality combination choices [RLLS97]. As such, an effective QoS management which takes each user's specific QoS preferences into account must span individual resources in an integrated and accessible way.

Note that the service's consumer determines the QoS requirements, which might change over time, while the data source (frequently remote from the consumer and therefore using different resources) and transport medium determine the quality and form of information. Furthermore, there might be multiple, simultaneous bottlenecks (i.e., the most constrained resources) and the bottlenecks might change over time.

A QoS manager must therefore capture the QoS requirements from each individual user and application, manage all the resources that could be bottlenecks, mediate conflicting demands for resources, effectively utilise allocated resources, and dynamically reallocate them as conditions change.

Jensen et al. proposed a soft real-time scheduling technique based on applications' benefit [JLT85]. Each application specified a benefit curve that indicated the relative benefit to be obtained by scheduling the application at various times with respect to its deadlines. The goal was to schedule applications so as to maximise an overall system's benefit. While this approach is intuitively very appealing, it is computationally intractable. Nevertheless, this work led to the adoption of utility functions to represent varying satisfaction with service changes in several other works.

Research on adaptive QoS control [TK93, MJ95, LKRM96] brings us a step closer to the QoS support from a user's perspective by providing a mechanism to accommodate potential dynamic changes in the operating environment. Other works [VN97, SWM95] propose analytical models to support QoS metrics of video and multimedia applications. Both works identify the QoS parameters related to an user's satisfaction and resource consumption in these types of applications and the corresponding functions for the relationship of resources and users' satisfaction. However, these mechanisms are still mainly system-oriented and the user has limited influence over the QoS to be delivered or adapted.

In coping with the shortage of QoS support from an end-user point of view, Rajkumar et al. [RLLS97] proposed Q-RAM, a QoS-based resource allocation model in which multiple resources are allocated to maximise the overall system's utility. The static resource allocation algorithms of Q-RAM were extended to support a dynamic task traffic model [HLR01] and to handle non-monotonic dimensions [GRH+03]. Q-RAM can be regarded as a generic approach to an imprecise computation approach, with the resource allocation problem treated as a general nonlinear or integer programming problem to be solved offline. Thus, the Q-RAM solution is generally not adaptable for dynamic environments. However, Rajkumar et al. [GHRL04] reduced the computation complexity of the initial proposal and adapted Q-RAM to dwell control for phased array radar by repeatedly testing schedulability online using high-performance computers.

In [JLDB95], a user-centric approach is also taken. The user's QoS preferences are considered for the application's runtime behaviour control and resource allocation planning. Example preferences include statements such as "a video-phone call should pause a movie unless it's being recorded" and "video should be degraded before audio

when all desired resources are not available". These are useful hints for high-level QoS control and resource planning, but are inadequate for quantitatively measuring QoS or analytically planning and allocating resources.

Numerous other studies on the arbitration of applications based on precise specifications of QoS requirements have also been published. In the next paragraphs, some representative works are discussed.

Abdelzaher et al. [AAS00] proposed a QoS negotiation mechanism to ensure graceful service degradation in cases of overload, failures, or violation of pre-runtime assumptions, and applied the approach in operating systems and middleware implementations [AS98, AS99]. Tasks' acceptable QoS levels are described a priori, as well as a quantitative perceived utility of receiving service at each of those levels. A similar approach is taken in [Kha98]. But none of these works addresses the balancing of competing resource demands, considers the dynamic negotiation of QoS levels, or has developed an effective specification method of QoS preferences or a mechanism to facilitate utility data acquisition.

Fan [Fan95] describes an architecture in which applications request a continuous range of QoS commitment from a centralised QoS Manager. Based on the current state of the system, the QoS Manager may increase or decrease an application's current resource allocation within this pre-negotiated range. However, the proposed system suffers from instability due to the fact that ranges are continuously being adjusted. Furthermore, it lacks a strong mechanism for deciding which applications' allocations to modify and when. It also assumes that any application can be written in such a way as to work reasonably with any resource allocation within a particular range.

A similar approach based on a centralised QoS manager is proposed by Brand et al. in [BNBM98]. Their work proposes a mediation method for resource allocation based on the maximums processor usage and users' benefit as measures for QoS levels and presents the Dynamic QoS Manager (DQM) architecture. DQM is based on the notion of applications' specified execution levels that reflect algorithmic modes in which applications can execute. It uses the execution level information and the current state of the system to dynamically determine appropriate QoS allocations for the running applications.

Hola-QoS [GVARG02] is a QoS-aware architecture based on four layers. Each one handles a different conceptual entity: (i) QoS management, to decide which applications should be executed according to the user's preferences; (ii) Quality control, to negotiate with the selected applications a service configuration and to find the feasible configuration that maximises the user's satisfaction; (iii) Budget control, to perform

the feasibility test of the set of budgets required to support a candidate configuration. It is in charge of creating and initialising budgets and monitoring how budgets are used; and (iv) Run-time control, that can be viewed as an extension to the operating system to provide the basic functionality of a resource kernel.

Foster et al. [FFR+04] propose the General-purpose Architecture for Reservation and Allocation (GARA), a generic architecture for resource reservation and allocation that supports flow-specific QoS specifications as well as online monitoring and control of both individual resources and heterogeneous resource sets. The architecture builds on differentiated service mechanisms to enable the coordinated management of distinct flow types, networks, CPUs, and storage systems.

Palopoli et al. [PVC+05] introduce an architecture for supporting the feedback-based adaptive management of multiple resources on a general purpose operating system, extending a prior architecture for adaptation of the CPU bandwidth for QoS control [CPM+04]. The architecture allows applications to share access to resources by specifying their fraction of usage, which are then dynamically adapted by the system at runtime, based on made observations on the hosted activities.

While we certainly share some concerns with these works, and also apply utility-based adaptation strategies [NP05], we go a step further and propose a QoS-aware cooperative service execution to deal with a large number of possibly dependent tasks, multiple resources, and highly dynamic real-time operation constraints in open real-time systems.

Several studies in computation offloading propose task partition/allocation schemes that allow the computation to be offloaded, either entirely or partially, from resource constrained (wireless) devices to a more powerful neighbour [WL04, GMG+04, LWX02]. These works conclude that the efficiency of an application execution can be improved by careful partitioning the workload between a device and a fixed neighbour. Often, the goal is to reduce the needed computation time and energy consumption [LWX01, KHR01, OH98, RRPK98, CKK+04] by monitoring different resources, predicting the cost of local execution and that of a remote one and deciding between a local or remote execution. However, most of the work in this direction is limited to the case where there is only one resource-limited device and one relatively more capable neighbour to offload computation to. Also, none of these works supports the maximisation of each user's specific QoS preferences while offloading computation.

The CooperatES framework proposed in this thesis not only facilitates the cooperation among heterogeneous nodes whenever a particular set of QoS constraints cannot be satisfyingly answered by a single node, but also ensures that the resulting coalition is

the one which maximises the satisfaction of the QoS constraints associated with the new service and minimises the impact on the global QoS caused by the new service's arrival. Chapter 3 discusses this approach in detail.

Furthermore, the CooperatES framework differs from other QoS-aware frameworks by considering, due to the increasing complexity of open real-time systems, the needed tradeoff between the level of optimisation and the usefulness of an optimal runtime system's adaptation behaviour. The fundamental problem that has to be faced is the uncertainty of the environment. In particular, when considering real-time requirements, uncertainty means that desired bounds may not be met when adapting the system to the dynamically changing environmental conditions. As such, QoS optimisation should be based on incremental processing resource-aware algorithms with variable completion times, able to adapt their performance based on the computing time made available to them.

Imprecise computation and anytime algorithms provide such flexibility. Imprecise computation [LLS+91] logically divides each task into a mandatory and an optional part. This division is done in such a way that the system still performs acceptably as long as all the mandatory parts are executed. The optional part is (usually) an iterative refinement algorithm that progressively improves the quality of the result generated by the mandatory part.

These concepts were integrated with replication and checkpoint techniques to reduce the cost of providing fault tolerance and enhanced availability of real-time systems [LLB+94, HFL95]. The Imprecise Computation Environment (ICE) [HFL95] was proposed as an environment for implementing imprecise real-time systems on top of Real-Time Mach. It uses a modified version of the standard client/server architecture that adds support for the automatic generation of imprecise servers. Whenever a client calls an imprecise server, it specifies the maximum imprecision it can tolerate. Then, the server creates an imprecise task for each request, which is then scheduled by the operating system along with other tasks in the system.

Recently, imprecise computation models have been used to maximise QoS provisioning under energy constraints in embedded systems [YVH08, WF08, WWGF08]. These works propose preemptive schedulers for imprecise tasks which prevent the execution of optional subtasks whenever there is the possibility of deadline loss or depletion of the energy source.

Anytime algorithms [DB88, Hor88, Zil96] are structured to take advantage of as much time as is available, undertaking more detailed calculations as more computing is given. They are able to return a partial answer, if interrupted at any time before reaching

completion, whose quality depends on the amount of computation they were able to perform. Nevertheless, their use in QoS-aware real-time systems has not, to the best of our knowledge, been actively explored before.

One noticeable exception is [ACS03], where an anytime approach is proposed to develop new model-based algorithms for generating and evaluating aircraft context-sensitive trajectories under resource and timing constraints. The work was part of DARPA's Software Enabled Control program [Off], whose goal was to develop and demonstrate the software infrastructure necessary to enable high-performance control algorithms, which inevitably have complex computational properties, that could be reliably used in avionics systems.

This thesis advocates that the success of open real-time embedded systems has not only to do with the accuracy of the performed adaptations but also with the timeliness criteria that affects the usefulness of the system's adaptation behaviour. Chapter 4 reformulates the QoS optimisation problem as a heuristic-based anytime optimisation problem that can be interrupted at any time and quickly respond to environmental changes. A strong argument in favour of our approach is that it benefits from this anytime QoS optimisation which is critical for use in highly dynamic and heterogeneous systems like those we discussed before.

2.3 Adaptive middleware

Middleware encapsulates a set of services residing above the operating system layer and below the user application layer, facilitating the communication and coordination of applications' components that are potentially distributed across several networked hosts. Moreover, middleware provides application developers with high-level programming abstractions which hide interprocess communication, mask the heterogeneity of the underlying systems (hardware devices, operating systems, and network protocols), and facilitate the use of multiple programming languages at the application level.

Emmerich [Emm00] provides a frequently referenced taxonomy of middleware. His taxonomy is based on the type of programming-language abstraction that the middleware provides for interaction among distributed software components: transactional, message-oriented, procedural, or object-oriented. The corresponding primitive communication techniques are distributed transactions, message passing, remote procedure calls, and remote object invocations, respectively. Bakken [Bak01] introduced a similar taxonomy that also includes four classifications: distributed tuples, message-oriented,

remote procedure call, and distributed object. In this chapter,the Emmerich's taxonomy is used to help lay a foundation for our later discussion of adaptive middleware.

Transactional middleware supports distributed transactions among processes running on distributed hosts. Originally, this type of middleware was targeted at interconnecting heterogeneous database systems. Goals include providing data integrity, high-performance, and availability using the two-phase commit protocol [BHG87]. IBM CICS [Hud94] and BEA Tuxedo [Hal96] are two examples of this category.

Message-oriented middleware facilitates asynchronous message exchange between clients and servers using the message-queue programming abstraction, a generalisation of an operating system's mailbox. Messages do not block a client and are deposited into a queue with no specific receiver information. In addition, the message-queue abstraction decouples clients and servers, which enables interaction among otherwise incompatible systems. IBM MQSeries [GS96] and Sun Java Message Queue [HBS99] are two examples of this category.

Procedural middleware extends procedural programming languages to include remote procedure calls (RPC), where the body of the procedure resides on a remote host and can be called the same way as a local procedure. Birrell and Nelson [BN84] implemented the first full-fledged version of RPC. Sun Microsystems adopted RPC as part of its open network computing [Sri95]. Later, Open Group developed a standard for RPC called distributed computing environment (DCE) [Gro97]. Most Unix and Windows operating systems now support RPC facilities.

Finally, object-oriented middleware is based on both the object-oriented programming paradigm and the RPC architecture. It provides the abstraction of a remote object, whose methods can be invoked as if the object were in the same address space as its client. Encapsulation, inheritance, and polymorphism are often supported by this type of middleware. CORBA [Groa], Java RMI [Mica], and DCOM [Cora] are three major object-oriented middleware approaches.

Among these four types, our primary focus is on object-oriented middleware, which is the basis for most research in adaptive middleware. Therefore, the remainder of this section reviews some of the most important object-oriented middleware approaches.

In the last years, the OMG (Object Management Group) has improved the CORBA standard specifications with respect to real-time issues. For instance, it has adopted the CORBA/e [Grob] and Real-Time CORBA specifications [Groc].

Similarly, with the appearance of pervasive computing paradigms and ad hoc networks, the possibility of having a large number of devices collaborating within a flexible

structure has to be exploited. Even if they do not provide any real-time features, both Jini [Micb] (in the area of service discovery for embedded networked devices) and .NET [Corb] (in the field of platforms for the development of distributed software systems) are worth mentioning due to their increasing popularity. Both allow to build distributed embedded systems based on services that appear and disappear dynamically.

However, these middleware platforms are very limited in their ability to support adaptation. As such, adaptive middleware emerges as a solution to address in an unified manner the complexity of programming interprocess communication, the need to support services across heterogeneous platforms, and the need to adapt to dynamically changing environmental conditions.

One of the earliest adaptive middleware projects proposed is the Adaptive Communication Environment (ACE) [Sch93, SH02], a real-time object-oriented framework written in C++, that provides high-performance and real-time communication services. ACE employs software design patterns to support distributed applications with efficiency and predictability, including low latency for delay-sensitive applications, high performance for bandwidth-intensive applications, and predictability for real-time applications.

Schmidt et al. [SLM98] extended their ACE work to create the ACE ORB (TAO), a CORBA compliant real-time ORB built atop of the ACE components. TAO enhances the standard CORBA event service to provide real-time event dispatching and scheduling required by real-time applications such as avionics, telecommunications and network management systems. Earlier versions of TAO employ the strategy design pattern [GHJV95] to encapsulate different aspects of the ORB internals, such as IIOP pluggable protocols, concurrency, request demultiplexing, scheduling, and connection management. A configuration file is used to specify the strategies used to implement these aspects during startup time. TAO parses the configuration file and loads the required strategies. Recent versions of TAO decomposes the C++ implementation of TAO into several core ORB components that can be dynamically loaded on demand using the virtual component pattern [CSKO02].

The Component-Integrated ACE ORB (CIAO) [WSK03] is the TAO implementation of CORBA's Component Model (CCM), which also resides in the distribution layer. CIAO intended to provide component-based design to distributed real-time and embedded (DRE) system developers by abstracting systemic aspects, such as QoS requirements and composable meta-data units supported by the component framework.

Kon et al. [KRL+00] proposed a dynamically adaptive version of TAO called Dynamic-

TAO using computational reflection. To provide real-time services, DynamicTAO uses the Dynamic Soft Real-Time Scheduler (DSRT) [Gar99] that provides QoS guarantees to applications with soft real-time requirements.

DSRT also integrates the QualMan (QoS-aware resource management) middleware [NhCN98], which consists of a set of resource servers (schedulers and brokers) to provide QoS negotiation, admission, and reservation capabilities for sharing resources such as CPU, network, and memory. It was designed to support QoS requirements of distributed multimedia applications.

QuO [ZBS97] is a well-known adaptive middleware framework for the integration of QoS management in CORBA and Java RMI. QuO's emphasis is on QoS specification, measurement, control, and adaptation to changes. It provides a number of capabilities, including application-level specification and monitoring of QoS, flexible adaptation and control when the quality requirements are not being met, and integration of different subsystems providing QoS mechanisms and services.

Blair et al. [BCD98] have investigated the middleware implementation for mobile multimedia applications which can be dynamically adapted in response to the environmental changes in the context of the Adapt project. In the OpenORB project [BCRP98], the successor of Adapt, Blair et al. continued their investigation studying the role of computational reflection in middleware. More recently, Blair et al. [BCA+01] designed OpenORB v2 that adds a component-based design framework to the OpenORB reflective framework. OpenCOM [CBCP01] is the implementation of OpenORB v2, designed for Microsoft COM systems. All these projects are greatly influenced by the ITU-T/ISO RM-ODP [IT95], a meta standard for multimedia applications. FlexiNet [Hay97] is another CORBA compliant ORB implemented in Java that uses reflection to provide dynamic adaptation. FlexiNet is designed as a set of components, which can be dynamically assembled. Similar to DynamicTAO [KRL+00], FlexiNet provides coarse-grained ORB-wide adaptation. FlexiNet can dynamically modify the underlying communication's protocol stack through the replacement and insertion of layers. Similar to OpenORB [BCRP98], FlexiNet also provides fine-grained per-interface adaptation.

Squirrel [KBH+01b, Kos02] is a QoS-oriented middleware specialised for distributed multimedia applications. Squirrel uses the Infopipes abstraction [KBH+01a] to support streaming data. The designers argue that CORBA stubs and skeletons generated from IDL interfaces follow a standard protocol (marshalling and unmarshalling) that is not suitable for multimedia applications with different QoS requirements. To solve this problem, Squirrel introduces smart proxies [Kos02], which are service-specific stubs

that include adaptive code. A smart proxy for a specific application can be developed and shipped to the client program statically at compile time or dynamically at runtime.

MetaSockets [SMK03], developed at Michigan State University, also address the issue of adaptable multimedia streams. MetaSockets are created from existing Java socket classes using Adaptive Java, a reflective extension to Java, whose structure and behaviour can be adapted dynamically in response to external stimuli. MetaSockets provides a pipeline abstraction similar to that of Squirrel. However, the adaptation supported in MetaSockets are finer-grained due to dynamic insertion and removal of filters instead of the whole pipeline as in Squirrel. A filter is a Java class that can be developed by third parties and can be inserted into the MetaSocket pipeline during runtime to adapt the application behaviour.

Adaptive middleware is still an ongoing research area. Dealing with highly dynamic interactions among nodes and continuously changing environments that demand predictable operation is still an open challenge. Since our focus is on complex real-time systems made of embedded components, then even more stringent requirements have to be taken into account, namely to achieve distributed, safe and timely process control. In this context, the provision of adequate interaction paradigms is a fundamental aspect [BMB+00].

From another perspective, it is important to observe that the emergence of applications operating independently of direct human control is inevitable [ACH+01, HB01]. Research on high-level models for this class of applications has revealed the shortcomings of current architectures and middleware interaction paradigms. Typical characteristics of this class of applications, such as autonomy or mobility must be accommodated, while allowing the possibility to handle nonfunctional requirements like reliability, timeliness or security.

In contrast with the client/server or RPC based paradigms supported by current state-of-the-art object-oriented middleware, event models have shown to be quite promising in this area [HLS97, MC02, Crn02]. Event notifications contain data that represent a change to the state of the sending applications' component, avoiding a centralised control and requiring a less tightly coupled communication relationship between applications' components compared to the traditional client/server communication model. However, the existing middleware approaches offering event services often lack one key point, the provision of support for nonfunctional attributes.

Therefore, this thesis proposes new architectural constructs that are adequate to such event-based interaction models and, at the same time, provide adequate support to address the specific requirements of embedded real-time systems. Chapter 6 presents an

one-step decentralised coordination model based on an effective feedback mechanism to reduce the complexity of the needed interactions among nodes until a collective adaptation behaviour is determined whenever the autonomous self-adaptations to the changing environmental conditions have an impact on other coalition members. Positive feedback is used to reinforce the selection of the new desired global service solution, while negative feedback discourages nodes to act in a greedy fashion as this adversely impacts on the provided service levels at neighbouring nodes.

2.4 Real-time scheduling

In a multitasking system, scheduling has two main functions: (i) maximise the processor's usage, i.e., the ratio between active and idle time; and (ii) minimise tasks' response time, i.e., the time between tasks' release time and the end of its execution. At best, response time may be equal to execution time, when a task is elected immediately and executed without preemption.

A scheduling algorithm is then a set of rules defining the execution of tasks at runtime. It is provided with a schedulability or feasibility test, which determines, whether a set of tasks with parameters describing their temporal behaviour will meet their temporal constraints if executed according to the rules of the algorithm.

There is a fundamental difference between hard and soft real-time scheduling. Hard real-time preserves temporal and functional feasibility, even in the worst case. Hard real-time scheduling has been concerned with providing guarantees for temporal feasibility of task execution in all anticipated situations, focusing on the worst case.

The temporal attributes and demands used by real-time scheduling for feasibility analysis and runtime execution form the task model an algorithm can handle. Early applications, such as simple control loops, had temporal characteristics that can be represented by simple temporal constraints. Hence, most algorithms and task models are dominated by attributes such as period, computation time, and a deadline. While periods and deadlines are typically derived from application characteristics, computation time is a function of the task code.

Most scheduling algorithms for periodic task sets have been developed around one of three basic schemes: table driven, fixed priority, or dynamic priority. Depending on whether the majority of scheduling issues are resolved before or at runtime, the algorithm is referred as an offline or online scheduler.

Offline scheduling builds a complete planning sequence before tasks' execution. In

Table-Driven Scheduling (TDS) approaches [RS94] a table determines which tasks to execute at which points in time. As only a table lookup is necessary to execute the schedule, task dispatching is very simple and does not introduce a large runtime overhead. TDS methods are capable of managing distributed applications with complex constraints, such as precedence, jitter, and end-to-end deadlines and are the ones usually associated with Time-Triggered architectures, such as TTP, which is commercially available. While its rigidity enables deterministic behaviour, it limits flexibility drastically. Furthermore, a previous knowledge about all the activities and events that may occur at runtime may be hard or impossible to obtain in some environments.

Online scheduling methods overcome these shortcomings by choosing, at any time, the next task to be executed. When a new event occurs, the running task may be changed without the need to know in advance the time of this event's occurrence. This dynamic approach provides less precise statements than the static one and has a higher implementation overhead. However, it manages the unpredictable arrival of tasks and allows a progressive creation of the scheduling sequence.

Fixed priority scheduling (FPS) [LL73] is a common policy in many standard operating systems, assigning priorities to tasks before the system's runtime and executing, at runtime, the task with the highest priority from the set of ready tasks. The basic fixed priority scheduling algorithms are Rate Monotonic [LL73] and Deadline monotonic [LW82].

Dynamic priority scheduling, as applied by the Earliest Deadline First (EDF) [LL73] policy, selects, at runtime, the task which has the closest deadline from the set of ready tasks. As such, priorities do not follow a fixed pattern, but change dynamically at runtime.

However, the strict compliance with every deadline is not mandatory in many embedded time-sensitive applications, characterised by implicit timing constraints for which occasional failures can be tolerable as long as they do not become too frequent. For instance, when streaming a MPEG movie, the delayed decoding of a few frames is not even perceived by the user as long as the system behaves "well" in average.

In the scientific community there has been, and still is, much interest in reservation-based scheduling. One major reason is to provide acceptable response for soft real-time tasks, while bounding their interference of hard-real time tasks. With reservation-based scheduling, a task or subsystem receives a real-time share of the system resources according to a (pre-negotiated) contract. Thus, the contract contains timing requirements. In general, such a contract boils down to some approximation of having

a private processor that runs at reduced speed.

As such, instead of scheduling tasks based on worst-case execution measures, guarantees based on average estimations of the needed computation time are typically acceptable for soft real-time tasks. A deadline miss does not constitute a system or application failure but it is only less satisfactory for the user and, as such, the approach is generally regarded as cheaper and more flexible.

Since the actual execution time of tasks can be affected by several factors, every task may either need more or less than its reserved computation time at runtime, when scheduled based on average estimations. If a task needs more than its reserved budget, the overload should remain isolated to that particular task, not jeopardising the schedulability of other tasks. Not only it is desirable to achieve temporal isolation among soft real-time tasks but also to not compromise the schedulability of hard real-time tasks. On the other hand, whenever a task completes in less than its budgeted execution time, it releases computation time that can be used to advance the execution of overloaded tasks.

A large number of schemes, both in fixed and dynamic scheduling approaches, have been described in the literature to reclaim any spare time coming from early completions and to handle overload situations preserving the schedulability of hard tasks.

Optimal fixed priority capacity reclaiming algorithms that minimise soft tasks' response times whilst guaranteeing that the deadlines of hard tasks are met were proposed in [LRT92, DTB93]. However, they present some drawbacks. The work in [LRT92] relies on a pre-computed table that defines the residual capacity present on each invocation of a hard task. In contrast, [DTB93] determines the amount of available residual capacity at runtime, but the execution time overhead introduced by the optimal dynamic approach is infeasible in practice [Dav93].

In [BB02], Bernat and Burns propose a capacity sharing protocol for enhancing soft aperiodic responsiveness in a fixed priority environment, where each task is handled by a dedicated server. The protocol allows an overloaded server to steal capacity from other servers to advance the execution of the served tasks, thus loosing isolation among the served tasks.

The capacity sharing protocol of [BB02] has been extended by the HisReWri algorithm [BBB04]. The algorithm identifies those tasks that did execute when a hard task has released some of its maximum allocated capacity and retrospectively assigns their execution times to the hard task. If there is residual capacity available, tasks' budgets are replenished by the amount of residual capacities they consumed. As execution time

is retrospectively reallocated, the authors describe the protocol as history rewriting.

In dynamic scheduling, a well known technique for limiting the effects of overruns was proposed by Abeni and Buttazo [AB98]. The Constant Bandwidth Server (CBS) scheduler handles soft real-time requests with a variable or unknown execution behaviour under EDF [LL73] scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task gets a fraction of the CPU and it is scheduled in such a way that it will never demand more than its reserved bandwidth, independently of its actual requests. This is achieved by assigning each soft task a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded. As a consequence, overruns occurring on a served task will only delay that task, without compromising the bandwidth assigned to other tasks.

However, with CBS, if a server completes a task in less than its budgeted execution time no other server is able to efficiently reuse the amount of computational resources left unused. To overcome this drawback, CBS has been extended by several resource reclaiming schemes [LB00, CBS00, MLBC04, CBT05, LB05], proposed to support an efficient sharing of computational resources left unused by early completing tasks. Such techniques have been proved to be successful in improving the response times of soft real-time tasks while preserving all hard real-time constraints.

GRUB [LB00] reduces the number of task preemptions by assigning all the excess capacity to the currently executing CBS server. Although a greedy reclamation policy is used, excess capacity always tends to be distributed in a fair manner among needed servers across the time line. However, GRUB always postpones a server's deadline before starting a new job, regardless of the current value of the server's budget. A critical parameter of this approach is the time granularity used in the algorithm, since a small period reduces the scheduling error, but increases the overhead due to context switches [CBS00].

CASH [CBS00] uses a global queue of residual capacities originated by early completions, ordered by deadline. Whenever a CBS server is scheduled for execution it will first use any queued capacity whose deadline is less than or equal to its own, reducing the number of deadline shifts and executing periodic tasks with more stable frequencies. However, since CASH immediately recharges the servers' capacities without suspending the tasks on every capacity exhaustion, tasks may be scheduled in an unexpected way [MLBC04]. An improvement to CASH's residual bandwidth

reclaiming and the ability to work in the presence of shared resources have been later proposed in [CBT05].

IRIS [MLBC04] identifies the deadline aging problem in previous extensions to CBS when scheduling acyclic tasks (tasks that are continuously active for large intervals of time) and proposes to suspend each task's replenishment until a specific time, following a hard reservation approach [RJM+98]. It is also a fair algorithm in the sense that residual capacity is equally distributed among the servers that need to execute more than the reserved time. However, residual capacity reclaiming is only performed after all the servers had exhausted their reserved capacities, potentially wasting valuable bandwidth that could otherwise have been used.

BACKSLASH [LB05] proposes to retroactively allocate residual capacities to tasks that have previously borrowed their current resource reservations to complete previous overloaded jobs, using an EDF version of the mechanism implemented in HisReWri [BBB04]. At every capacity exhaustion, servers' capacities are immediately recharged and their deadlines extended as in CBS. However, a task that borrows from a future job remains eligible to residual capacity reclaiming with the priority of its previous deadline. The main problem of this approach is that allowing a task to use resources allocated to the next job of the same task may cause future jobs of that task to miss their deadlines by larger amounts. Considering the mean tardiness of a set of periodic tasks on higher system loads, BACKSLASH can be outperformed by an algorithm that do not borrows from future resources [LB05].

While these scheduling schemes generally improve the system's performance, new scheduling requirements are emerging as the number of applications with soft real-time constraints continues to grow. In fact, current real-time scheduling methods focus on periodic tasks with bounded execution times. However, many tasks used in real-time control and optimisation applications do not fit this pattern [Haw03, ACS03, SCC04, BB04, vdBFK06, NP06c, NP09b].

Consider for example a route optimiser that is part of the navigation system of an automated vehicle [SCC04]. Given the state of the external and internal world, the system is continuously searching for the best path. The task execution is unbounded, data-driven, not predictably regular, and as a result its operation is not easily parcelled for a periodic execution. With an anytime approach [ZR96], the optimisation task can be interrupted at any time and still be able to provide a solution and a measure of its quality. As such, systems can take advantage of the flexibility offered by anytime algorithms as long as a scheduling mechanism that can regulate their behaviour is developed.

Furthermore, existing scheduling schemes are only able to reclaim the unused allocated capacity made available when jobs complete in less than their budgeted execution time and a proper reclaiming of unused capacities of idle servers is not supported. Isolation can be reduced in a controlled fashion in order to donate reserved, but still unused, capacities to currently overloaded servers. Chapter 5 proposes the Capacity Sharing and Stealing scheduler, able to distinguish between reclaiming "residual capacity" due to earlier completions of periodic tasks (predictable arrivals) and reclaiming "non-isolated capacity" from inactive servers which handle aperiodic best-effort tasks (non-predictable arrivals) in order to advance the execution of overloaded servers and reduce their mean tardiness.

Particular attention has also been given by the research community to tasks with uncertainty about the actual arrival time, i.e., they do not occur in a periodic manner. These are called aperiodic if no assumptions at all can be made about their arrival time and sporadic if at least a minimum time between to consecutive arrivals can be given. In the latter case, a worst-case assumption, the minimum inter-arrival time, can be used to include sporadic tasks as periodic in the feasibility test [Mok83]. Aperiodic tasks are usually generated by external events and activated by interrupts, for example coming from a sensory acquisition board.

While no hard guarantees can be made about aperiodic tasks, algorithms have been presented to allocate a certain amount of processing time reserved for aperiodic tasks, which allows analysis of response times of the entire set of aperiodic tasks. Typically, this reservation is done via server tasks [SSL89], which lend their resources to aperiodic tasks and which are included in the offline feasibility analysis as place holders, or as bandwidth [SB94], i.e., a portion of the processing time. In the case of table driven scheduling, the amount and location of unused resources is known to serve aperiodic tasks [Foh95].

However, all of the schedulers discussed above assume tasks are independent. When tasks share access to some of the system's resources they introduce a contention issue that affects schedulability. Let us consider a critical resource R, shared by two tasks τ_1 and τ_2, that must be accessed under mutual exclusion. Specific mechanisms, such as semaphores or protected objects, provided by a real-time kernel or programming language can be used to access those critical sections.

The question is how to ensure a predictable response time of real-time tasks in a preemptive scheduling mechanism. In fact, if classical mutual exclusion semaphores are used, a particular problem arises, usually referred as priority inversion [SRL90]. If a higher priority task is blocked on a semaphore by a lower priority task and another

medium priority task arrives, the latter can preempt the lower priority task causing an unbounded blocking delay to the higher priority task. It is important to note that in a non-preemptive context this problem does not arise since, by definition, a task cannot be preempted during a critical section.

Several protocols have been developed for preventing the priority inversion under the Rate Monotonic and Earliest Deadline First scheduling context. These protocols determine an upper bound of the blocking time due to a critical resource access for each task in the system. This maximum blocking duration is then integrated into the schedulability tests of classical scheduling algorithms.

The basic idea of the Priority Inheritance Protocol [SRL90] is to dynamically change the priority of those tasks accessing a critical section. A task τ_i, which is inside a critical section, gets the priority of any higher priority task τ_j waiting for the resource. Consequently, task τ_i is scheduled, during the duration of the critical section, with a higher priority than its initially assigned priority. This new context leads to freeing the resource earlier, minimising the waiting time of higher priority tasks.

The Priority Ceiling Protocol [CL90] extends the previous protocol by preventing a task to enter in a critical section that leads to blocking it, in order to avoid deadlocks and chained blocking. To do so, each resource is assigned a priority, called priority celling, equal to the priority of the highest priority task that can use it. However, it is important to note that this protocol needs to know a priori all the tasks' priorities and the resources used by each task.

The Stack Resource Policy [Bak90] allows the use of multi-unit resources and can be applied with a variable-priority scheduling like EDF. In addition to the classical priority, each task is assigned a new parameter π, called level of preemption, which is related to the time devoted to its execution (π is inversely proportional to its relative deadline). This level of preemption is such that a task τ_i cannot preempt a task τ_j unless $\pi(\tau_i) > \pi(\tau_j)$. The main difference between the Priority Ceiling Protocol and the Stack Resource Policy is the time at which a task is blocked. With the Priority Ceiling Protocol, a task is blocked when it wants to use a resource, while with the Stack Resource Policy a task is blocked as soon as it wants to get the processor.

Some scheduling solutions based on these protocols were already proposed [Jef92, CS01, CBT05, Bar06] but they all require a prior knowledge of the maximum resource usage and, as such, cannot be directly applied to open real-time systems.

Coherently with the resource reservation approach of CBS, resource sharing among tasks of open real-time systems started to be addressed in [LLA01] without requiring

any *a priori* knowledge about the tasks' structure and temporal behaviour. The proposed Bandwidth Inheritance (BWI) protocol extends the CBS scheduler to work in the presence of shared resources, adopting the Priority Inheritance Protocol to handle tasks' blocking. However, its main drawback is its unfairness in bandwidth distribution. A blocking task can use most (or all) of the reserved capacity of one or more blocked tasks, without compensating the tasks it blocked. Blocked tasks may then loose deadlines that could otherwise be met.

To address the lack of a compensation mechanism of BWI, BWE [WLP02] and CFA [SLS04] try to fairly compensate blocked servers in exactly the same amount of capacity that was consumed by a blocking task while executing in a blocked server. However, these attempts to fairly compensate borrowed capacities introduce a high overhead.

Chapter 7 proposes the Capacity Exchange Protocol (CXP), extending CSS to efficiently schedule inter-dependent task sets without introducing any significant overhead. CXP merges the benefits of a smart greedy capacity reclaiming policy with the concepts of bandwidth inheritance and hard reservations, focusing on greedily exchanging extra capacities as early, and not necessarily as fairly, as possible.

2.5 Summary

Traditionally, real-time systems were focused on providing a single, specific solution to single, specific applications, treating all activities with the same methods, geared towards the most demanding scenarios. However, in the last years, the use of processor-based devices has increased dramatically and there is extensive research work on topics such as ambient intelligence, pervasive systems, disappearing computer, home automation, and ubiquitous computing, which aim at a better integration of computers in our in our daily lives.

In these open and uncertain environments, classical methodologies for real-time systems design are hardly applicable since the set of applications to be executed and their aggregate resource and timing requirements are unknown until runtime. At the same time, users increasingly demand for QoS guarantees. As such, runtime adaptation is a must in order to achieve a desired level of performance.

This chapter discussed specific topics relevant for the development of adaptive real-time embedded systems, their current limitations, and introduces the novel research directions that are proposed in this thesis to increase the flexibility and enhance the functionality of new embedded real-time systems.

Chapter 3

Cooperative embedded systems

The scarcity and diversity of resources among the devices of heterogeneous computing environments may affect their ability to execute services within users' acceptable QoS levels. The problem is even more complex in open real-time environments where the characteristics of the computational load cannot always be predicted in advance but, nevertheless, response to events still has to be provided within precise timing constraints in order to guarantee a desired level of performance.

This chapter addresses these complex demands by proposing a cooperative QoS-aware framework, allowing resource constrained devices to collectively execute services with their more powerful or less congestioned neighbours, meeting non-functional requirements that otherwise would not be met by an individual execution.

3.1 Introduction

Embedded real-time systems are increasingly at the core of a wide range of domains, including consumer electronics, telecommunications, medicine, avionics and automotive where independently developed real-time and non-real-time services may coexist in a system with a set of finite resources under management. As services dynamically enter and leave the system at any time, resource requirements are inherently unstable and difficult to predict in advance [GRHL04]. As a consequence, the overall system's workload is subject to significant variations, which can overload resources and degrade the entire system's performance in an unpredictable fashion [HLR01].

In order to achieve the users' acceptance requirements, the underlying resource management layer must supply services and protocols which know how to negotiate, admit, and enforce resource allocations according to the dynamically changing resource requirements. Online methods that react to load variations and adapt the system's performance in a controlled fashion overcome the shortcomings of a rigid off-line design and worst-case assumptions, providing the needed flexibility.

Decision making in the presence of several QoS dimensions and service requests can be performed through multi-criteria decision analysis. The aim is to rank competing services according to their QoS characteristics and users' service preferences. Section 3.3 proposes a sufficiently expressive model for defining the QoS dimensions subject to negotiation, their attributes and the quality constraints in terms of possible values for each attribute, as well as inter-dependency relations between some of those QoS parameters. Given such model, a service provider will be able to appropriately compare services and take better decisions according to each user's service preferences.

At the same time, an increasing number of applications needs a considerable amount of computation power and is pushing the limits of traditional data processing infrastructures [SCZ05]. During loaded periods, a particular node may not have sufficient resources to deliver the minimum desired quality to every application along each of its QoS dimensions. Consider, for example, the real-time stream processing systems described in [MOFR01, EEGL03, SLSL05]. The quantity of data produced by a variety of data sources and sent to end devices for further processing is growing significantly, increasingly demanding more processing power. The challenges become even more critical when a coordinated content analysis of data sent from multiple sources is necessary [EEGL03]. Thus, with a potentially unbounded amount of stream data and limited resources, some of the processing tasks may not be satisfyingly answered by a single node, even at the users' minimum acceptable QoS levels [SLSL05].

Hence, decisions must be made by the underlying resource management framework to share available resources among applications such that a global objective is maximised. By redistributing the computational load across a set of nodes, a cooperative environment enables the execution of far more complex and resource-demanding services than those that otherwise would be able to be executed on a stand-alone basis.

Section 3.4 proposes the CooperatES (Cooperative Embedded Systems) framework [NP05, PNB05] to facilitate the cooperation among neighbours whenever a particular set of constraints cannot be satisfyingly answered by a single node. Nodes dynamically group themselves into a new coalition, allocating resources to each new service and establishing an initial Service Level Agreement (SLA). The coalition is dynamically

formed as the set of nodes which maximise the satisfaction of the QoS constraints associated with the new service and minimise the impact on the global QoS caused by the new service's arrival [NP05].

Nevertheless, short term dynamic environmental changes impose that the promised SLA for a service S_i can never be more than an expectation of a best-effort service quality during long term periods [Bur03]. The system must be adaptive, that is, it must be able to adapt its timing expectations to the current conditions of the environment, possibly sacrificing the quality of other, non-time related, parameters. As such, once a SLA is admitted, it may be downgraded to a lower QoS level in order to accommodate new service requests with a higher utility or (re)upgraded when the needed resources become available.

However, while some users or applications may prefer to always get the best possible instantaneous QoS, independently of their services' reconfiguration rate, others may find that frequent QoS reconfigurations are undesirable [NP06b]. Section 3.7 details our approach for adapting the services' execution to the dynamically changing system's conditions under the control of each individual user's stability preferences.

The work discussed in this chapter is partially presented in [NP05, PNB05, NP06b].

3.2 Problem description and system model

Consider an open distributed system with several heterogeneous nodes, each with its specific set of resources R_i where independently developed services, some of them with real-time execution constraints, can appear while other are being executed, at any time, at any node. Due to these characteristics, resource availability is highly dynamic and unpredictable in advance.

Each service S_i has a set of parameters that can be changed in order to achieve an efficient resource usage that constantly adapts to the devices' specific constraints, nature of executing tasks and dynamically changing system conditions. Each subset of parameters that relates to a single aspect of service quality is called a *QoS dimension*.

For example, consider the transmission of multiple audio/video streams over a network. This scenario involves a network with a given bandwidth and nodes serving and receiving the streams. Typical audio related parameters are the sampling rate (8, 16, 24, 44, 48 kHz), the sampling bits (8, 16), and the end-to-end latency (100, 75, 50, 25 ms), while in video it is usually considered the picture dimension (SQCIF, QCIF, CIF, CIF4), colour depth (1, 3, 8, 16, ...), and frame rate (1, ..., 30).

Each of these QoS dimensions has different resource requirements for each possible level of service. We make the reasonable assumption that services' execution modes associated with higher QoS levels require higher resource amounts.

Furthermore, note that different configurations of a stream can have different utility values for different users and applications. For example, for a particular user a transmission of a music concert may place higher quality requirements on audio, although colour video may also be desirable, while another user of a remote surveillance system may require higher video quality with a minimum of gray scale images.

Users provide a single specification of their own range of QoS preferences Q_i for a complete service S_i, ranging from a desired QoS level $L_{desired}$ to the maximum tolerable service degradation, specified by a minimum acceptable QoS level $L_{minimum}$, without having to understand the individual components that make up the service. As a result, the user is able to express acceptable compromises in the desired QoS and assign utility values to QoS levels. Note that this assignment is decoupled from the process of establishing the supplied service QoS levels themselves and determining the resource requirements for each level.

Let Q_i be the set of the user's QoS constraints associated with service S_i. Each Q_{kj} is a finite set of quality choices for the j^{th} attribute of dimension k. This can be either a discrete or continuous set.

For some of the system's nodes there may be a constraint on the type and size of services they can execute within the users' acceptable QoS levels Q_i. Given a node n and a set of SLAs σ to be provided, we assume the existence of the following function:

Definition 3.2.0.1

$$\begin{aligned} feasibility(\sigma_n) &= true, \text{ if } \sigma \text{ is feasible in node } n \\ feasibility(\sigma_n) &= false, \text{ otherwise} \end{aligned}$$

Proposition 3.2.0.1 *Given a node n and a set of SLAs σ to be provided, the function $feasibility(\sigma_n)$ always terminates and returns true if σ is feasible in n or false otherwise.*

Therefore, this thesis addresses a distributed cooperative execution of resource intensive services in order to maximise the users' satisfaction with the obtained QoS. Nodes may cooperate either because they can not deal alone with the resource allocation demands imposed by users and services or because they can reduce the associated execution cost by working together.

There will be a set of independent tasks $\tau_1, \ldots, \tau_n$ to be executed, resulting from partitioning the resource intensive service S_i. Correct decisions on service partitioning must

be made at run time when sufficient information about workload and communication requirements become available [WL04], since they may change with different execution instances and users' QoS preferences.

Given the spectrum of the user's acceptable QoS levels Q_i for service $S_i = \{\tau_1, \ldots, \tau_n\}$, the coalition formation problem can be described as:

> Given a set of neighbour nodes N and a resource allocation demand enforced by Q_i, 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 in N to cooperatively execute S_i should be influenced by both the maximisation of the QoS constraints Q_i associated with S_i and by the minimisation of the impact on the current QoS of the previously accepted services caused by the arrival of S_i.

The reader should note that this thesis is focused on the initial configuration and runtime adaptation of a distributed cooperative QoS-aware service execution. The proposed approach is completely independent from how the code to be executed on the original node's behalf arrives to the coalition members. Services' remote blocks can be migrated to the selected partners after the coalition formation decision or, alternatively, all the nodes in the system are a priori equipped with all the code blocks.

3.3 Expressing Quality of Service

Given the heterogeneity of services to be executed, users' quality preferences, underlying operating systems, networks, devices, and the dynamics of their resource usages, QoS specification becomes an important issue in the context of a distributed QoS-aware cooperative service execution framework. However, as open distributed systems become more complex, so is the specification of requested and supplied QoS among users and service providers. Nodes must either have a common understanding of how QoS should be specified, or be able to map their individual specifications into a common one.

The definition of such a generic QoS scheme must include quality dimensions, attributes and values, as well as relations that map dimensions to attributes and attributes to values. Adopting a common QoS description scheme in an open distributed environment guarantees information consistency and compatibility in a community of

heterogeneous nodes. Information consistency is satisfied when each specific expression has the same meaning for every node. Information compatibility is achieved when any concept is described by the same expression, for all the nodes. Furthermore, a generic QoS scheme should also be extensible to support the later addition of news terms and relations as the system evolves.

We model each of these diverse requirements by the following structure [NP05], that can be expressed in several QoS description languages [JN04]:

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

where $Dim = \{Dim_0, \ldots, Dim_n\}$ is the set of QoS dimensions, $Attr = \{Attr_0, \ldots, Attr_m\}$ is the set of attributes identifiers, $Val = \{Val_0, \ldots, Val_k\}$ is the set of attribute's values identifiers, DAr and AVr are the set of relationships that assign attributes to dimensions and values to attributes, respectively, and $Deps$ is the set of dependencies among the values of different QoS attributes.

Each value is represented by a triple

$$Val_i = \{Value, Type, Domain\}$$

where $Type = \{integer, float, string\}$, and $Domain = \{continuous, discrete\}$.

The set of relationships DA_r assigns to each dimension in Dim a set of attributes in $Attr$ and is defined as

$$DA_r : Dim_i \rightarrow Atr, \forall_{Dim_i} \in Dim$$

The set of relationships AV_r assigns to each attribute in $Attr$ a specific value in Val and is represented as

$$AV_r : Atr_i \rightarrow Val_k, \forall_{Atr_i} \in Atr, \exists^1{}_{Val_k} \in Val$$

$Deps$ defines the set of existing dependencies among the values of the existing attributes. There are n QoS attributes $x_1, x_2, \ldots, x_n$, whose values are taken from the domains $D_1, D_2, \ldots, D_n$, respectively, and a set of dependency constraints on their values. The constraint

$$Dep_{ij} = p_k(x_{k1}, \ldots, x_{kj}), \forall Attr_i, Attr_j \in Attr$$

is a predicate that is defined on the Cartesian product $D_{k1}x \ldots xD_{kj}$. This predicate p_k is true if and only if the value assignment of these variables satisfies this constraint. Note that there is no restriction on the form of the predicate. It can be a mathematical or logical formula or any arbitrary relation defined by a tuple of acceptable values.

Using a video streaming application as an example, the following is a list of quality dimensions that might be associated with any particular application. The list is given to illustrate the proposed model and is not intended to be exhaustive.

```
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 = {compression index, color depth, frame size,
                    frame rate}

DA Audio Quality = {sampling rate, sample bits}

AV compression index = {[0,100]},
AV frame size (pixels) = {80x40, 240x180, 320x240, 640x480, 720x480,
                          ...}
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}
```

Having such a QoS characterisation of a particular application domain, users and service providers are now able to define their service requirements and proposals in order to reach an agreement on service provisioning. Since QoS is often multi-dimensional, a user (or application) might want to make some quality tradeoff, especially when the available resources are scarce. Therefore, it is to the user's advantage to be able to specify a set of personal QoS requirements using an interface that explicitly allows the definition of quality tradeoffs.

Consider the following example. Typically, the video frame rate fluctuates as the system's load fluctuates. However, frame rate is an important QoS parameter for talk shows because it affects lip synchronisation [Nak98]. As such, other QoS parameters

like the frame size or the compression index may be better candidates for degradation when the needed resources become scarce. On the other hand, in a remote video surveillance system, a grey scale, low frame rate may be sufficient, but a high image quality is important. As such, an efficient system's QoS optimisation policy must consider the specific quality requirements of each user or application.

A flexible approach to deal with the heterogeneity and load variations of dynamic open environments is to define such personal quality requirements through a utility model. Several works associate with each pre-defined QoS level a utility function that specifies the user's benefit in obtaining service within those values [AAS00, RLLS97, LL07]. However, it may be clearly infeasible to make the user specify an absolute utility value for every pre-defined quality choice. While we want a semantically rich request in order to achieve a service provisioning closely related to the user's quality preferences, we also want the user to actually be able to express personal QoS preferences in a service request.

Furthermore, rather than demanding the user to predefine QoS levels with associated utility values, we propose to dynamically determine QoS levels according to each user's acceptable QoS values and local resource availability and compute the reward of executing a task at one of those dynamically determined QoS levels based on the number, and relative importance, of the QoS dimensions being served closer to the user's desired QoS level.

Following those goals, a more natural and realistic way to describe acceptable QoS levels and their related utility is to simply formulate a service request based on a qualitative, not quantitative, measure. With a relative decreasing order on quality dimensions, their attributes, and accepted values, a user is able to encode the relative importance of the new service's performance at the different QoS levels without the need to quantify every quality tradeoff with absolute values.

For example, a user of a remote video surveillance system can easily state that video is more important than audio, and the image's quality is more important than the obtained frame rate and colour depth with the following service request:

```
1. Video Quality
 (a) compression index : {[0,20], [21,30]}
 (a) frame rate    : {[10,6], [5,1]}
 (b) color depth   : {3, 1}
2. Audio Quality
 (a) sampling rate : {11, 8}
```

```
(b) sample bits   : {8, 4}
```

Note that for each of the QoS attributes a preference order may be as well expressed. The evaluation of the user's acceptability of each service proposal with respect to the expressed quality preferences is detailed in Section 3.5.

3.4 The CooperatES framework

Currently, middleware technologies such as CORBA or .NET are being widely used in many application areas to mask out the heterogeneity of systems and networks and alleviate the inherent complexity of distributed systems. However, the recent emergence of new application areas for embedded real-time systems imposes new challenges in terms of resource sharing, dynamism, and timeliness which most existing middleware platforms are unable to tackle.

In this section, we propose a generic solution to the problem of task allocation among autonomous heterogeneous nodes and suggest that nodes form coalitions in order to execute services that otherwise could not be delivered within the users' acceptable QoS levels.

The CooperatES (Cooperative Embedded Systems) framework enables resource-constrained devices to solve computationally expensive services by redistributing parts of the service onto other devices, forming temporary coalitions for a cooperative service execution. Such distribution is influenced by the maximisation of the QoS preferences associated with the new service request, addressing the increasingly complex demands on performance and customisable service provisioning.

Each node has a significant degree of autonomy and it is capable of performing tasks and sharing resources with other nodes. A service can be executed by a single node or by a group of nodes, depending on the user's device capabilities and imposed quality constraints. In either case, the service is processed in a transparent way for the user, as users are not aware of the exact distribution used to solve the computationally expensive services.

In the proposed model, QoS-aware applications must explicitly request the service execution to the underlying CooperatES framework, thus providing explicit admission control, abstracting from the existing underlying distributed middleware and operating system. The model itself abstracts from the communication and execution environments.

Figure 3.1 presents the structure of the proposed framework, running on every node of the network.

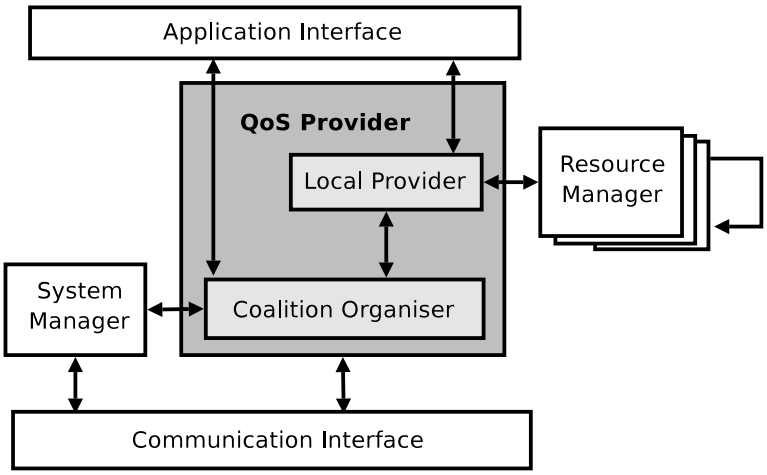

Figure 3.1: Framework structure

Central to the behaviour of the framework is the *QoS Provider* of each node which is responsible for processing both local and remote resource requests. Rather than reserving local resources directly, it contacts the *Resource Managers* to grant specific resource amounts to the requesting tasks. This negotiation is based on a contract model: there is trading of quality by resources. For this purpose, resource usage accounting, budget enforcement, and monitoring are required mechanisms. Note that in this thesis it is assumed that failures of resources will not occur during services' execution but only that they may get overloaded.

Each *Resource Manager* is a module that manages a particular resource. The module interfaces with the actual implementation in a particular system of the resource controller, such as the device driver for the network, the scheduler for the CPU, or with the software that manages other resources (such as memory). Although we consider a collaborative environment, proper resource usage must be monitored at run time [BP04], in order to make decisions based on the actual system's resource usage and not only on the resource usage assumptions of requesting services.

An effective QoS-aware resource management requires resolving multiple views of QoS ranging from high-level user perceptive quality down to lowest level views closer to individual resources and their controls. Although there can be many layers, we identify three in particular:

System layer. At the system layer, there is the knowledge of the system's goals, the

applications in the system, and the available resources. This is also the layer at which there is an understanding of the relative importance of applications to mission goals, resource allocation strategies for each goal, and the policies for mediating conflicting application resource needs.

Application layer. An application view of resource management involves acquiring whatever resources are needed to meet applications' requirements and to effectively utilise the available resources. If there are not enough resources, then an application needs to be flexible enough to adjust its resource needs by gracefully degrading its quality level.

Resource layer. Resource specific mechanisms control access to each individual resource, deciding whether and how a request for a resource allocation should be granted. Typical resource allocation mechanisms have little or no knowledge of the applications using them or their requirements, although some limited information can be propagated to the resource level in the form of relative priorities and reservation requests.

As such, resource managers have the ability to use each other in order to allow systems to be built supporting QoS requirements either from the point of view of the user (e.g. user-perceived high quality), of applications (e.g. video frame rate) or of the system (e.g. CPU cost). As an example, a particular system may provide the resource manager layering of Figure 3.2. An interactive application can be more user friendly and easier to use by providing only high-level user perceptive quality, whilst other applications can be programmed to use application-related QoS constraints.

The *System Manager* maintains the overall system configuration, controlling and monitoring the partner's execution and resolving conflicts arising from the autonomous adaptation of nodes.

Figure 3.3 details the structure of the *QoS Provider*. To guarantee the execution of a local or remote service request, the node's *Local Provider* tries to find a feasible set of SLAs that maximises the utility associated with the new service's QoS configuration and minimises the impact on the current QoS of previously accepted services, using the service proposal formulation algorithms discussed throughout this thesis.

If the resource demand imposed by an user's QoS constraints cannot be locally satisfied, the *Coalition Organiser* is responsible for the coalition formation process. It broadcasts the service's description as well as the user's quality constraints, evaluates the received service proposals and decides which nodes will form the new coalition, using the coalition formation algorithms discussed throughout this thesis.

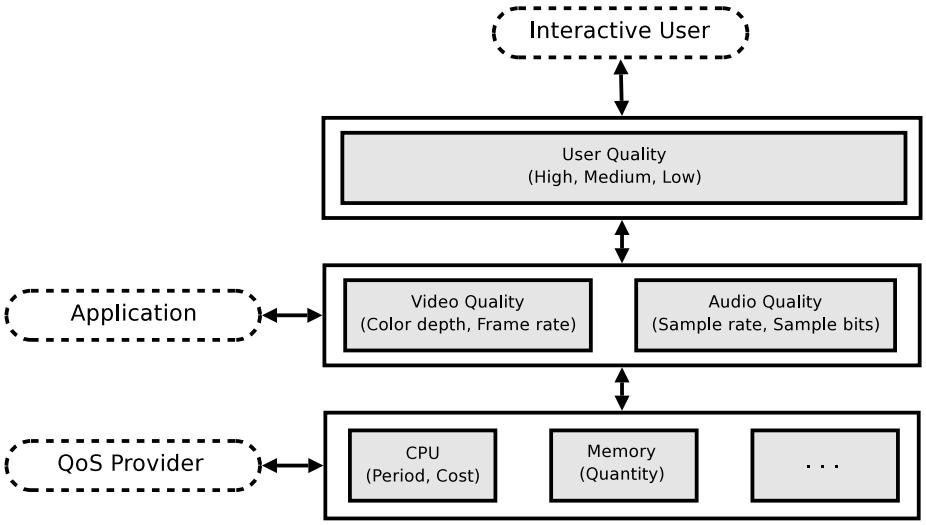

Figure 3.2: Resource managers' layering

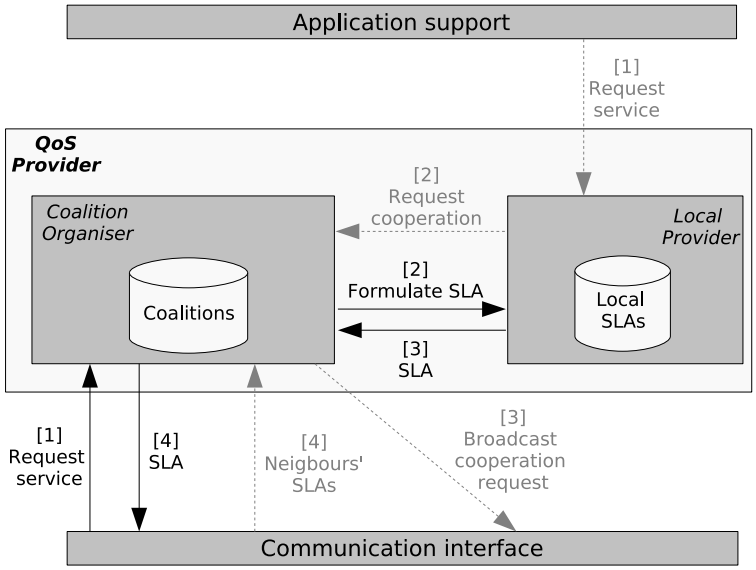

Figure 3.3: QoS Provider

The details of an initial implementation of a prototype of the CooperatES framework in Ada are described in [PNB05]. The paper assesses the suitability of the Ada language to be used in dynamic QoS-aware systems.

3.5 Coalition formation

The formation of a new coalition for a cooperative service execution should enable the selection of individual nodes that, based on their own resources and availability, will constitute the best group to satisfy the user's QoS requirements associated with the resource intensive service.

As previously discussed, a service request is considered to be formulated through the relative decreasing importance ($k = 1 \ldots n$) of a set of n QoS dimensions, ranging from a desired QoS level $L_{desired}$ to the maximum tolerable service degradation, specified by a minimum acceptable QoS level $L_{minimum}$. For each dimension, a relative decreasing importance order of attributes is also specified ($i = 1 \ldots attr_j$), where j is the number of attributes of dimension k. Please note that k and i are not the identifiers of dimensions and attributes in a domain's QoS description, but their relative position in a user's service request.

Consider that the user's service request can be translated into the acceptable QoS region represented in Figure 3.4 for the video dimension.

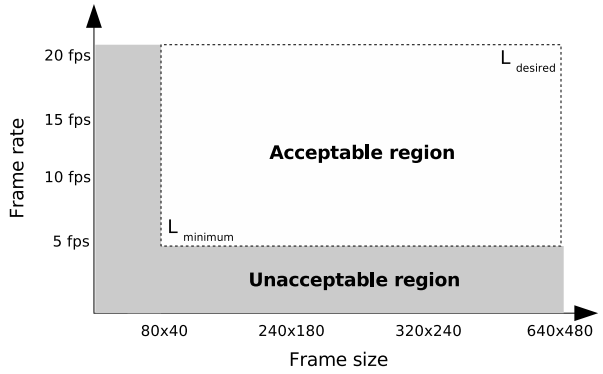

Figure 3.4: Acceptable service quality

Whenever the user's node n_u is unable to execute an entire service S_i within the user's acceptable QoS levels Q_i, its *QoS Provider* broadcasts a cooperation request. The set of tasks that can be remotely executed is determined by a task partition/allocation scheme that dynamically considers the tradeoff between local execution requirements and communication costs [WL04]. The cooperation request includes a description of each remote task τ_i and the user's QoS constraints Q_i for the entire service S_i.

Every neighbour node n_j which is able to execute one of the tasks τ_i within Q_i formulates a service proposal according to a local QoS optimisation algorithm (please

refer to Section 3.6 for details) and replies to the user's node n_u with both its service proposal P_{ji} and its local reward R_j, resulting from its cooperation acceptance. For now, it suffices to say that the local reward is an indicator of the node's local QoS optimisation level, according to the set of services being locally executed and their associated QoS constraints. How each node measures its local reward will be detailed in Section 3.6.

It is clear that different groups of nodes will have different degrees of efficiency in the service's cooperative execution performance due to different capabilities of their members and their current state. As such, the coalition's members selection should be determined by the proximity of the nodes' service proposals with respect to the expressed user's multi-dimensional QoS constraints.

Each admissible proposal[1] P_{ji} is then evaluated by determining, for each QoS dimension Q_k, a weighted sum of the differences between the user's preferred values and the values proposed in P_{ji}, using Equation 3.1.

$$distance(P_{ji}) = \sum_{k=1}^{n} w_k * dif(Q_k) \tag{3.1}$$

where n is the number of QoS dimensions under negotiation and $0 \leq w_k \leq 1$ is the relative importance of the k^{th} QoS dimension Q_k to the user and can be defined as

$$w_k = \frac{n - k + 1}{n} \tag{3.2}$$

The utility of each proposed value $prop_{ki}$ for the QoS attribute $attr_{ki}$ when compared to the user's preferred one $pref_{ki}$ is given by Equation 3.3, considering separately continuous and discrete domains.

$$dif(Q_k) = \sum_{i=1}^{m} w_i * |da(prop_{ki}, pref_{ki})| \tag{3.3}$$

where m is the number of attributes in the k^{th} QoS dimension and $0 \leq w_i \leq 1$ is the relative importance of the i^{th} attribute to the user and can be defined as

$$w_i = \frac{m - i + 1}{m} \tag{3.4}$$

[1] A service proposal for a task τ_i is admissible if it can satisfy all QoS dimensions within the user's acceptable QoS levels Q_i

In Equation 3.3, the function $da(prop_{ki}, pref_{ki})$ quantifies, for a QoS attribute $attr_{ki}$, the degree of the user's acceptability of the proposed value $prop_{ki}$, when compared to the user's preferred value $pref_{ki}$ and is defined as

$$da = \begin{cases} \dfrac{prop_{ki} - pref_{ki}}{max(Q_k) - min(Q_k)} & \text{, if continuous } Q_{ki} \\ \\ \dfrac{pos(prop_{ki}) - pos(pref_{ki})}{length(Q_k) - 1} & \text{, if discrete } Q_{ki} \end{cases} \tag{3.5}$$

If attribute $attr_{ki}$ has a continuous domain, this quantification is a normalised difference between the proposed value $prop_{ki}$ and the preferred one $pref_{ki}$. For discrete domains, Equation 3.5 considers the preferences attached to $prop_{ki}$ and $pref_{ki}$ by using their relative position in the service's QoS requirements specification.

In [LLS+99] the authors use the notion of a *quality index*, defining a bijective function that maps the elements of a discrete domain into integer values. Here, we use a similar approach by mapping the position (index) of that attribute in the domain's specification into $prop_{ki}$'s and $pref_{ki}$'s scoring values.

Whenever the domain's QoS description defines the possible values for some attribute of a QoS dimension Q_k by a set of intervals, Q_k in Equation 3.5 must relate to the particular interval where $prop_{ki}$ is found. In a similar fashion, if the user expresses a set of acceptable intervals for a QoS attribute $attr_{ki}$ of dimension Q_k, the considered preferred value $pref_{ki}$ should be the first value of the particular interval where $prop_{ki}$ is found and the relative decreasing order of importance w_k of that interval to the user must also be considered.

The best proposal for each of the service's tasks is thus the one that presents the lowest distance to the user's quality preferences in all QoS dimensions. Furthermore, each node's local reward can be used to improve a global load balancing. Consider two proposals whose evaluation differ by an amount less than α (this value can be defined by the user or by the framework). For a particular user, the perceived utility will be equally acceptable if any of those nodes is selected for being part of the new coalition but the global system's resource optimisation is improved if the node with a higher local reward is selected instead. Selecting the node with a higher local reward from two similar service proposals, not only maximises the service's quality for a particular user, but also maximises the global system's utility.

As such, the coalition formation process, detailed in Algorithm 1, enables the selection of those nodes which offer service closer to the user's desired QoS level and, at the

same time, efficiently distributes the computational load across available nodes.

Algorithm 1 Coalition formation

Let $E_{P_{ki}}$ be the evaluation value of proposal P_{ki}, sent by node n_k for task τ_i
Let $Best_{\tau_i}$ be the best proposal for task τ_i
Let R_k be the local reward of node n_k
Let R_{best} be the local reward of the node which has proposed $Best_{\tau_i}$ for task τ_i
Let C be the formed coalition

1: Start with an empty coalition C ▷ $C = \emptyset$
2: **for** each task $\tau_i \in S_i$ **do**
3: **for** each received k^{th} proposal P_{ki} for task τ_i **do**
4: $E_{P_{ki}} = distance(P_{ki})$
5: **if** $(Best_{\tau_i} - E_{P_{ki}} > \alpha)$ or $(0 < Best_{\tau_i} - E_{P_{ki}} < \alpha$ and $R_k > R_{best})$ **then**
6: $Best_{\tau_i} = E_{P_{ki}}$
7: Update coalition with node n_k for task τ_i ▷ $C = C \setminus (n_j, \tau_i) \cup (n_k, \tau_i)$
8: **end if**
9: **end for**
10: **end for**
11: **return** coalition C

The algorithm terminates when all the received proposals are evaluated or if it finds that the quality of a coalition cannot be further improved because all the tasks $\tau_i \in S_i$ will be served at the user's preferred QoS level $L_{desired}$.

3.6 Service proposal formulation

All nodes that participate in a cooperative QoS-aware service execution negotiation must provide sufficient resources to formulate a SLA within the user's acceptable QoS levels $[L_{minimum}, L_{desired}]$. It is therefore the responsibility of each individual *QoS Provider* to map the user's QoS constraints to local resource requirements, and then reserve resources accordingly (recall that resource reservations are made through *Resource Managers*).

The interpretation of QoS constraints and consequent mapping on the needed resource quantities has been explored, for example, in [RLLS97, FWMM97, GP99, BSLH05]. This thesis is focused in the dynamic formation and runtime adaptation of cooperative coalitions and does not deal with this mapping. The reader can assume that services

make a reasonable accurate analysis of their resource requirements computed a priori by resource monitoring tools and improved by run-time adaptation.

Requests for a cooperative service execution arrive dynamically at any node and are formulated as a set of acceptable multi-dimensional QoS levels. Such QoS management is known to be NP-hard [LLS+99]. As a consequence, there are no optimal solution techniques other than a (possibly complete) enumeration of the solution space. On the other hand, QoS management calls for on-line solutions because the optimisation module will ideally be used in the admission control of an adaptive QoS management system. Therefore the goal is to strike the right balance between solution quality and computational complexity.

Conventional admission control schemes either guarantee or reject each service request based on current local resource availability. On the other hand, we here propose a QoS negotiation mechanism that, in cases of overload, or violation of pre-runtime assumptions guarantees a graceful quality degradation in a controlled fashion. Offering QoS degradation as an alternative to a simple service rejection has been proved to achieve a higher perceived utility [AAS00]. An important attribute of the proposed QoS optimisation algorithm is its incremental and state-reuse property in order to avoid having to completely redo expensive computations to accommodate the dynamic arrival and departure of tasks.

To locally guarantee the cooperation request, each node's *QoS Provider* executes a local gradient descent QoS optimisation algorithm, quadratic in the number of tasks and resources and linear in the number of QoS levels. The goal is to maximise the satisfaction of the new service's QoS constraints while minimising the impact on the current QoS of previously accepted services. However, the CooperatES framework ensures a dynamically forecasted stability period Δ_t for each accepted service S_i, indicating that during that specific time interval the promised QoS level for a service S_i will be assured. As such, only services whose stability period has already expired can be downgraded to a lower quality level to accommodate new services with a higher utility. This subject will be discussed in detail in Section 3.7.

The proposed QoS optimisation, detailed in Algorithm 2, operates in rounds. In each round, it considers all the possible decrements of the services' QoS levels and identify the one that produces the minimum utility decrease, without violating the minimum requested QoS level. This process is repeated until a feasible set of QoS levels σ is found, or until the algorithm finds the set of SLAs unschedulable even at the lowest acceptable QoS level for each service. In this later case the new service request is rejected.

Algorithm 2 Service proposal formulation

Let τ^e be the set of previously accepted tasks whose stability period Δ_t has expired.
Let τ^p be the set of all previously accepted tasks whose current QoS cannot be changed.
Let τ^a be the newly arrived task.
Each task τ_i has associated a set of user's defined QoS constraints Q_i.
Each Q_{kj} is a finite set of n quality choices for the j^{th} attribute, expressed in decreasing preference order, for all k QoS dimensions.
Let σ be the determined set of SLAs, updated at each step of the algorithm

1: Select the maximum requested QoS level $Q_{kj}[0]$, for all the j attributes of the k QoS dimensions, for the newly arrived task τ^a, defining SLA_{τ_a}
2: Keep the current QoS level for each task $\tau_k \in \tau^e$
3: Update the current set of SLAs σ ▷ $\sigma = \sigma \cup SLA_{\tau_a}$
4: **while** $feasibility(\sigma) \neq$ **TRUE do**
5: **if** there are no task τ_i being served at $Q_{kj}[m] > Q_{kj}[n]$, for any j^{th} attribute of any k QoS dimension **then**
6: Reject τ_a
7: **end if**
8: **for** each task $\tau_i \in \tau^e \cup \tau_a$ **do**
9: **for** each j^{th} attribute of any k QoS dimension in τ_a receiving service at $Q_{kj}[m] > Q_{kj}[n]$ **do**
10: Determine the reward decrease by downgrading attribute j to $Q_{kj}[m+1]$
11: **end for**
12: **end for**
13: Find task τ_{min} whose reward decrease is minimum
14: Define the new $SLA'_{\tau_{min}}$ for task τ_{min} with the new value $Q_{yx}[m+1]$ for attribute x of the QoS dimension y
15: Update the current set of promised SLAs σ ▷ $\sigma = \sigma \setminus SLA_{\tau_{min}} \cup SLA'_{\tau_{min}}$
16: **end while**
17: **return** the new local set of promised SLAs σ

The reward r_{τ_i} of executing a task τ_i at the determined SLA depends on the number, and relative importance, of the QoS dimensions being served closer to the user's desired QoS level $L_{desired}$. The distance between the user's desired and the node's proposed values is computed through Equation 3.6,

$$r_{\tau_i} = \begin{cases} 1 & \text{, if task } \tau_i \text{ is being best} \\ & \text{served in all QoS dimensions} \\ 1 - \sum_{j=0}^{n} w_j * penalty_j & \text{, if } Q_{jk} < Q_{best_j} \end{cases} \tag{3.6}$$

where *penalty* is a parameter that decreases the reward value. This parameter can be fine tuned by the user or the framework's manager according to several criteria and its value should increase with the distance to the user's preferred values.

Using the reward achieved by each proposed SLA it is possible to determine a measure of the node's local QoS optimisation resulting from the acceptance of the new service request. For a node N_j, the local reward R_j achieved by the set of proposed SLAs is given by

$$R_j = \frac{\sum_{i=1}^{n} r_{\tau_i}}{n} \tag{3.7}$$

Note that unless all tasks are executed at their highest requested QoS level there is a difference between the determined set of SLAs and the maximum theoretical local reward that would be achieved if all local tasks were executed at their highest QoS level. This difference can be caused by either resource limitations, which is unavoidable, or poor load balancing, which, as discussed in the previous section, can be improved by sending nodes' local rewards along with their service proposals, and selecting, for proposals with similar evaluation values, those nodes that achieve a higher local optimisation.

3.7 Supporting runtime QoS adaptation and stability

Any service provider's resource allocation policy is subject to environmental uncertainties, and for that reason, the promised SLA can never be more than an expectation of an average service quality [Bur03]. To cope with dynamic environments, a system must be adaptive, that is, it must be able to adjust its current level of service in response to changes in the environment.

Traditionally, the adaptation behaviour was integrated within applications. Although minimum system changes were required to implement QoS delivery, there are some

disadvantages in that approach. Different applications running on the same system may have a different adaptive behaviour when a fluctuation on the task traffic flow occurs. Some of them may require a considerable amount of resources to perform their desired adaptation, while others may not be able to perform any adaptations at all. Furthermore, the adaptation component integrated into an application is not generic and reusable.

It is therefore desirable to propose a generic mechanism that adapts the services' execution to the dynamically changing system's conditions. Nevertheless, while some users or applications may prefer to always get the best possible instantaneous QoS, independently of the reconfiguration rate of their requested services, others may find that frequent QoS reconfigurations are undesirable. For example, in some video applications a constant frame rate may be better than a frequent variation whose average is higher than the initial contracted level of service.

As such, we consider that the dynamic QoS arbitration among competing services should be done under the control of the user [NP06b]. This suggests that while a resource constrained device may not be able to avoid a downgrade of the currently provided QoS level of some services in order to accommodate a new service with a higher utility, upgrades to a higher QoS level can and should be controlled by each user's stability requirements. Possible attributes for such QoS stability dimension can be a minimum granted stability period Δ_{min} and a minimum increment in the service's reward U_{min} in order to upgrade the current service's QoS level. These can be interpreted as "do not change to a better service's quality state unless this gives me at least a reward's increment of U_{min} over a Δ_{min} period". The flexibility and expressiveness of the QoS scheme proposed in Section 3.3 allows the user's stability preferences and their relative order of importance to be expressed as any other QoS dimension.

The system computes the possible upgrades of the currently provided SLAs (see Section 3.7.2 for details) and periodically forecasts the granted stability period (Section 3.7.1) for an entire service S_i. If the user's stability requirements are met, the current service's SLA is upgraded. Otherwise, the service is kept in its current QoS level and the pre-reserved resource amounts for the computed upgrades become immediately available for subsequent QoS negotiations.

3.7.1 Promised stability periods

From the service provider's side, each proposed SLA should now be complemented with a stability period Δ_t, indicating that during that specific time interval the promised QoS level for a service S_i will be assured either on the arrival and departure of other services.

Note that service stability could be achieved by using a fixed, large enough value for Δ_t, but this would then result in lack of responsiveness in adaptability to environmental changes. Furthermore, fixed values only make sense when there is some knowledge about the tasks' traffic model, which is not the case in open real-time systems. Proposed stability periods should then be periodically updated in response to variations in the tasks' traffic flow and corresponding resource usage, efficiently adapting the system's behaviour to the observed environmental changes.

Time series analysis comprises methods that attempt to understand a sequence of data points, typically measured at successive times, spaced at (often uniform) time intervals to make forecasts. Several algorithms are available for making predictions within a time series [BJ90]. Generally speaking, prediction algorithms are based on the analysis of the past observed n samples and range from simple solutions, e.g. moving averages, to complex ones based on optimal filtering theory.

The simple exponential smoothing (SES) model [Bro63] has become very popular as a forecasting method for a wide variety of time series data as it is both robust and easy to apply. In fact, empirical research by Makridakis et al. [MAC+82] has shown SES to be the best choice for one-period-ahead forecasting, from among 24 other time series methods and using a variety of accuracy measures. Thus, regardless of the theoretical model for the process underlying the observed time series, simple exponential smoothing will often produce quite accurate forecasts.

Intuitively, past data should be discounted in a more gradual fashion, putting relatively more weight on the most recent observations. SES accomplishes exactly such weighting, with exponentially smaller weights being assigned to older observations. We use the SES model to forecast the length of the next stability period for a service S_i by combining the forecasts for each of the system's resources r_i it uses.

Equation 3.8 is used recursively to update (forecast) the smoothed series as new observations are recorded for each resource r_i. The observed minimum stability period for resource r_i during the period of observation t is denoted by x_t and $\Delta_t^{r_i}$ may be regarded as the best estimate of what the next value of x will be.

$$\Delta_t^{r_i} = \alpha x_t + (1 - \alpha)\Delta_{t-1}^{r_i} \tag{3.8}$$

Each new forecast is then based on the previous forecast plus a percentage of the difference between that forecast and the actual value of x_t at that point. The percentage $0 \leq \alpha \leq 1$ is known as the smoothing factor. Values of α close to 1 have less of a smoothing effect and give a greater weight to recent changes in the data, while values of α closer to 0 have a greater smoothing effect and are less responsive to recent changes. α can then be adjusted by the system's designer to create a more reactive or conservative response to recent changes in the tasks' traffic flow. Alternatively, a statistical technique may be used to optimise the value of α, minimising the difference between the predicted and observed values. For example, the method of minimum least-squares may be used to determine α's value for which the sum of the quantities $(\Delta_{t-1}^{r_i} - x_t)^2$ is minimised [Bro63].

Having the stability forecasts for each of the system's resources, the promised stability period for a work unit of a particular service S_i must be based on a coherent summary of the forecasts for each of the resources it uses. Any use of an arithmetic summarisation function that combines the values (such as a mean), will provide an incorrect stability period due to relative scaling. On the other hand, combinations of several dynamical variables using logical operators has already been proposed to provide more expressive policies for SLAs [Rod02].

Equation 3.9 determines the promised stability period for service S_i by aggregating the forecasted values for each of the resources r_i it uses through the fuzzy AND operator (the *min* function). It allows a quick and simple evaluation of stability periods for each locally accepted service and leads to a correct system behaviour.

$$\Delta_t = min(\Delta_{r_1}\ AND\ \Delta_{r_2}\ AND\ \ldots\ AND\ \Delta_{r_n}) \tag{3.9}$$

3.7.2 Determine possible upgrades of previously downgraded Service Level Agreements (SLAs)

Rather than trying to upgrade previously downgraded services on every service departure, their QoS re-upgrade should be triggered based on a specific threshold of desired system utilisation, avoiding high reconfiguration rates in dynamic systems.

Let L_t be the desired system's resource usage threshold to activate the dynamic QoS re-upgrade of previously downgraded tasks whose stability period Δ_t has already expired.

Let L be the current level of the system's load demanded by the n offered SLAs. Intuitively, $L < L_t$ indicates an underutilisation and the dynamic QoS re-upgrade should take place.

Algorithm 3 implements a gradient descendent heuristic that starts with the initial contracted level for the previously downgraded tasks whose granted stability period has already expired and terminates when it finds a set of feasible QoS levels, if any. The goal is to reallocate the needed resources to supply the initially promised SLA for every task that had to suffer a QoS downgrade.

Algorithm 3 QoS re-upgrade of previously downgraded tasks

Let τ^e be the set of previously downgraded tasks whose stability period Δ_t has expired.
Let τ^p be the set of all previously accepted tasks whose current QoS cannot be changed.
Each task τ_i has associated a set of user's defined QoS constraints Q_i.
Let $Q_{kj}[init]$ be the initially provided and $Q_{kj}[i]$ the currently provided level of service for attribute j of the k_{th} QoS dimension for task $\tau_i \in \tau^e$
Let σ be the determined set of SLAs, updated at each step of the algorithm

1: Select the initially provided value $Q_{kj}[init]$ for all j attributes of the k QoS dimensions, for all tasks $\tau_i \in \tau^e$
2: Keep the current QoS level for all tasks in τ^p
3: Update the current set of SLAs σ
4: **while** $feasibility(\sigma) \neq$ **TRUE do**
5: **for** each task $\tau_i \in \tau^e$ **do**
6: **for** each j^{th} attribute of any k QoS dimension with value $Q_{kj}[m] > Q_{kj}[i]$ **do**
7: Determine the reward decrease by downgrading attribute j to $Q_{kj}[m+1]$
8: **end for**
9: **end for**
10: Find task τ_{min} whose reward decrease is minimum
11: Define the new SLA $SLA'_{\tau_{min}}$ for task τ_{min} with the new value $Q_{yx}[m+1]$ for attribute x of the QoS dimension y
12: Update the current set of promised SLAs σ $\triangleright$ $\sigma = \sigma \setminus SLA_{\tau_{min}} \cup SLA'_{\tau_{min}}$
13: **end while**
14: **return** the new local set of promised SLAs σ

If Algorithm 3 produces a new set of upgraded SLAs, an actual upgrade of each of

the currently provided SLAs only occurs if the user's stability requirements, namely the minimum granted stability period Δ_{min} and the minimum increment in the SLA's reward U_{min}, are met. Clearly, as these constraints are stringent, it is harder to upgrade to better quality levels.

3.8 Summary

As the complexity of various new embedded real-time systems increases, multiple tasks, whose actual resource demands are only know at runtime, have to compete for the limited resources of a single embedded device. In this context, resource constrained devices may need to collectively execute services with their neighbours in order to fulfil the complex QoS constraints imposed by users and applications. As such, an efficient arbitration of QoS levels in this highly dynamic, open, shared, and heterogeneous environment becomes very important.

This chapter presented the CooperatES framework, a QoS-aware framework that addresses the increasing demands on resources and performance in embedded real-time systems by allowing services to be executed by temporary coalitions of nodes. Users encode their own relative importance of the different QoS parameters for each service they want to execute and the framework uses this information to determine the distributed resource allocation that maximises the satisfaction of those constraints and minimises the impact on the current QoS levels of previously accepted tasks.

Particular attention was devoted in also maximising the users' influence on their services' adaptation behaviour during runtime. While a downgrade of the currently provided QoS level of some services may not be avoidable due to resource limitations, upgrades to a higher QoS level are controlled by each user's stability requirements, namely a minimum utility increment and a minimum stability period. The framework computes the possible upgrades of the currently provided QoS level and periodically forecasts the granted stability period for each accepted service. The current service's QoS level is upgraded only if the user's stability requirements are met.

Chapter 4

Time-bounded service configuration

The notion that the needed computation time to obtain optimal service solutions will typically reduce the overall utility of a cooperative service execution is formalised in this chapter using the concept of anytime algorithms. The use of an anytime approach is mainly inspired by its powerful ability to ensure a timely answer to events, despite the imprecision and uncertainty of open real-time environments.

Nodes start by negotiating partial, acceptable service proposals that are latter iteratively refined if time permits, in opposition to the traditional QoS optimisation approach proposed in the previous chapter that either runs to completion or is not able to provide a useful solution.

4.1 Introduction

Optimising QoS for systems operating in open environments involves dealing with a number of challenges not faced in many simpler domains. Open systems are inherently uncertain and dynamic and accurate optimisation models are difficult to obtain and quickly become outdated. Nevertheless, despite their uncertainty, responses to events still have to be provided within precise timing constraints in order to guarantee a desired level of performance.

As such, the design of effective QoS optimisation algorithms has been and remains very much an art. Before the search for a locally optimal solution can begin it has to be decided how to obtain an initial feasible solution. It is sometimes practical to initiate the search from several different starting points and to choose the best result.

Furthermore, a "good" neighbourhood has to be chosen for the problem at hand and a method for searching it. The choice is normally guided by intuition since very little theory is available as a guide.

Then, the analysis of the performance of a standard optimisation algorithm is concerned with the following: (i) time complexity, i.e, the time required by the algorithm to arrive at the final answer; (ii) size of the neighbourhood to be searched; (iii) choice of the pivot element, i.e., to which better neighbouring solution to move to; and (iv) the number of iterations required to reach a locally optimal solution.

For more than three decades, many researchers from the fields of mathematics, computer science and operations research have been working on combinatorial optimisation techniques that aim to reduce the needed computation time to find a solution. There are three algorithmic approaches [AL97, MT90] that have been well studied and widely used: (i) enumerative methods that are guaranteed to produce an optimal solution [Iba88]; (ii) approximation algorithms that run in polynomial time [Sah75, IK75]; and (iii) heuristic techniques that do not have a guarantee in terms of solution quality or running time, but provide a robust approach to obtaining a high quality solution to problems of a realistic size in reasonable time [AL97].

However, the increased complexity of dynamic open real-time environments may prevent the possibility of computing both locally and globally optimal resource allocations within a useful and bounded time. This is true for many soft real-time applications, where it may be preferable to have approximate results of a poorer but acceptable quality delivered on time than late results with the desirable optimal quality. For example, it is better for a collision avoidance system to issue a timely warning together with an estimated location of the obstacle than a late description of the exact evasive action. Another example concerns video and sound processing. While poorer quality images and voices on a timely basis may be acceptable, late frames and long periods of silence often are not. Other examples can be found in route optimisation of automated vehicles [vdBFK06, SCC04], computer games [Haw03], and real-time control [BB04]. What characterises these domains is that it is not computationally feasible or desirable to compute optimal answers. In other words, complex soft real-time problems need approximate solutions delivered on time.

Anytime algorithms have shown themselves to be particularly appropriate in such settings, as they usually provide an initial, possibly highly sub-optimal, solution very quickly and then concentrate on improving this solution until the time available for planning runs out. Nevertheless, there has been relatively little interaction between QoS management and anytime algorithms. QoS management research has been con-

centrated on finding single optimal, or with a fixed sub-optimality bound, solutions.

This chapter reformulates the distributed resource allocation problem for sets of independent task sets proposed in the previous chapter as a heuristic-based anytime optimisation problem 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.

The anytime approach for configuring the set of feasible QoS levels for independent task sets proposed in this chapter is partially presented in [NP06c, NP09b].

4.2 Anytime algorithms

When the problem is complex and the available time to find a solution is limited, generating optimal solutions can be infeasible. A useful approach in these situations is to employ the so-called anytime algorithms which deliver the best solution that can be generated within the available computation time.

In the broadest terms, an anytime algorithm is an iterative refinement algorithm that can be interrupted at any time during its execution and will always return a valid solution to the problem it is solving, with the possible exception of an initial time period before the first solution is found. It is expected that the quality of the answer will increase (up to some maximum quality) as the anytime algorithm is given increasing time to run, offering a tradeoff between the quality of the result and its computational cost. The term is due to Dean and Boddy who originally suggested the concept of anytime algorithms in their work on time-dependent planning [DB88].

The concept can be illustrated with some examples. An example of a problem for which there is no anytime algorithm is searching for an item. Solving this problem implies finding the item, so halting the search before the item is found will never return a solution to it. As a generalisation, all the problems with only one solution cannot be solved by an anytime approach. However, the planning of a route between two known points makes a good example of a problem that may be solved by an anytime algorithm. An initial solution can be constructed by plotting a straight line between two points. Then, this initial result can then be iteratively refined to better suit the search criteria (e.g. to avoid crossing rivers or to minimise the amount of energy used climbing hills).

A similar technique, termed flexible computation, was introduced by Horvitz [Hor88] to solve time-critical decision problems. This line of work is also closely related to

the notion of imprecise computation [LLS+91]. Imprecise computation uses monotone functions to produce intermediate results as a task executes. The value of these results is expected to improve as the execution of the task continues. The computation required to produce a result with minimum quality forms the *mandatory* part of the task. Clearly, this mandatory part must have a worst case execution time that is guaranteed by the schedulability analysis. The rest of the task's execution is called *optional*. The optional part is (usually) an iterative refinement algorithm that progressively improves the quality of the result generated by the mandatory part.

What is common to these research efforts is the recognition that the computation time needed to compute optimal solutions will typically reduce the overall utility of the system.

4.2.1 An anytime QoS optimisation approach

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 real-time systems increases, it is also increasingly difficult to achieve an optimal resource allocation that deals with both users' and nodes' constraints within an useful and bounded time.

Anytime computation extends the traditional notion of a computational procedure by allowing it to return many possible approximate answers to any given input. This flexibility makes it an obvious choice for integrating complex and unbounded QoS optimisations into highly dynamic open real-time systems. This leads to one of the primary general principles of this thesis:

> In order to enable a cooperative service execution that will function adequately in a real-time, dynamic, distributed, heterogeneous, and open environment, the algorithms used to configure such execution should be designed as anytime algorithms.

A system using an anytime service configuration gains the advantage of being able to explicitly manage resource usage during the QoS optimisation process. In this context, a resource is anything that is consumed by the optimisation process. Typically this will be CPU time, memory space, and energy. Similarly, if the utility of achieving a goal decreases as time increases, the utility of the goal can also be considered a resource when optimising for such time-dependent goals.

By using an anytime approach, the system can tradeoff the resource consumption against the quality of the produced solution. This can be done by monitoring the progress of the anytime optimisation, either through direct feedback or via a performance profile, and then interrupting the optimisation process when a set of SLAs that surpasses a certain quality threshold has been found. A performance profile is a representation of the relationship between processing time and result quality for a particular anytime algorithm and problem. Performance profiles can be used to predict how quickly a solution of a certain quality will be produced. The idea of performance profiles for anytime algorithms was first proposed in [DB88] and has been extended in works such as [Zil96] and [vHtT00].

However, although an algorithm that can be stopped at any time is potentially useful, some guarantees on its performance are necessary to ensure its applicability [Zil96]:

Measurable quality. The quality of an approximate result can be determined precisely. For example, when the quality reflects the distance between the approximate result and the correct result, it is measurable as long as the correct result can be determined.

Recognisable quality. The quality of an approximate result can easily be determined at run time (that is, within a constant time). For example, when solving a combinatorial optimisation problem (such as path planning), the quality of a result depends on how close it is to the optimal answer. In such a case, quality can be measurable but is not recognisable.

Monotonicity. The quality of the result is a nondecreasing function of time and input quality. Note that when quality is recognisable, the anytime algorithm can guarantee monotonicity by simply returning the best result generated so far rather than the last generated result.

Consistency. The quality of the result is correlated with computation time and input quality. In general, algorithms do not guarantee a deterministic output quality for a given amount of time, but it is important to have a narrow variance so that quality prediction can be performed.

Diminishing returns. The improvement in solution quality is larger at the early stages of the computation, and it diminishes over time.

Interruptibility. The algorithm can be stopped at any time and provide some answer. Originally, this was the primary characteristic of anytime algorithms.

Preemptability. The algorithm can be suspended and resumed with minimal overhead.

If these properties hold, an anytime algorithm is then able to return a valid result at any time and the longer the algorithm runs the better the results will be, until the optimal result has been reached. Note that it is not necessary that the optimal result is reached in the same time as the non-anytime, traditional version of the same algorithm, as long as the anytime version returns a good solution within a reasonable amount of time.

To what respect the preceding statement is true, and what is intended by a "good result" and "reasonably amount of time", largely depends on the problem that one is trying to solve. In the case of a QoS-aware cooperative service execution we strongly believe that a sub-optimal solution delivered on time is much better than not having any suitable service solution within the required time or having an optimal QoS optimisation delivered too late.

Our proposal, discussed in detail in the remaining sections of this chapter, is to quickly establish an initial, sub-optimal, service solution according to the set of QoS constraints that have to be satisfied. Then, if time permits, the initial solution is gradually refined until it finally reaches its optimal value or the available deliberation time expires. At each iteration, a new set of SLAs 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. The binary notion of correctness associated with traditional QoS optimisation algorithms is then replaced by a set of quality measured outputs.

As discussed above, a quality measure is an integral part of an anytime algorithm. The quality of partial solutions produced by an anytime algorithm should be both measurable and recognisable. The principle of measurable quality requires that the quality of an approximate result (i.e. not a full solution) can be determined accurately. The principle of recognisable quality requires that quality can be calculated at runtime without too much processing being required (e.g. in linear time). A quality measure is also necessary to determine whether the results returned from the algorithm are improving monotonically with respect to time.

Determining the quality of incomplete solutions produced by anytime algorithms has traditionally been a difficult proposition [Zil96]. In a first approach to the problem in our cooperative QoS-aware scenario, a good measure of success for the QoS optimisation process may seem to be how similar the service solution resulting from

the interrupted optimisation is to the solution that would have been generated if the optimisation process had been allowed to run to completion. However, this results in a measurable but not recognisable quality measure. As such, for the remaining of this thesis, each intermediate solution's quality measure indicates how close is the offered QoS level to the user's desired QoS level. Recall that the purpose of a cooperative service execution is to maximise the user's satisfaction with the provided service.

4.3 Anytime coalition formation

Iterative improvement algorithms can find a good approximation to an optimal solution and naturally yield an interruptible anytime algorithm [Zil96]. Based on this idea, this section reformulates the traditional coalition formation algorithm described in the previous chapter as an iterative refinement algorithm that can be interrupted at any time and still returns a feasible service solution.

The proposed anytime coalition formation algorithm, described in detail in Algorithm 4, uses each node's local reward as a heuristic to guide the coalition formation process. The goal is to quickly find a sufficiently good initial solution and gradually maximise its improvement at each iteration, it if time permits.

Clearly, nodes with a higher local reward have a higher probability to be offering service closer to this particular user's request under negotiation since the utility achieved by all services being locally executed is higher. Then, for each remote task $\tau_i \in S_i$, rather than depending on the order of proposals' reception, the next candidate proposal P_{ki} to be selected from the set of received proposals P_i is the one *sent by the node N_k with the greatest local reward R_k*, using Equation 4.1. [NP06c].

$$P_{ki} | P_{ki} \in P_i, max(R_k) \tag{4.1}$$

After an initial coalition has been determined, the algorithm iteratively continues, if time permits, to evaluate the remaining received proposals. Note that it is possible that some other node has sent a better proposal for the service request under negotiation even if it has a lower local reward. Recall that the service proposal formulation algorithm always suggests the best solution for the new service, even if it has to downgrade the currently provided level of service of the previously accepted services. It is the responsibility of the coalition formation algorithm to select between similar proposals (whose evaluation values differ in less than some configurable threshold α) those nodes that achieve higher local rewards, promoting load balancing.

Algorithm 4 Anytime coalition formation

Let $E_{P_{ki}}$ be the evaluation value of proposal P_{ki}, sent by node N_k for task τ_i
Let $Best_{\tau_i}$ be the best proposal for task τ_i
Let R_k be the local reward of node N_k
Let R_{best} be the local reward of the node which has proposed $Best_{\tau_i}$ for task τ_i
Let C be the formed coalition

1: Start with an empty coalition C ▷ $C = \emptyset$
2: **for** each $\tau_i \in S_i$ **do**
3: **for** each received k^{th} proposal $P_{ki} | P_{ki} \in P_i, max(R_k)$ **do**
4: $E_{P_{ki}} = distance(P_{ki})$
5: **if** $(Best_{\tau_i} - E_{P_{ki}} > \alpha)$ or $(0 < Best_{\tau_i} - E_{P_{ki}} < \alpha$ and $R_k > R_{best})$ **then**
6: $Best_{\tau_i} = E_{P_{ki}}$
7: Update coalition C with node n_k for task τ_i ▷ $C = C \setminus (n_j, \tau_i) \cup (n_k, \tau_i)$
8: **end if**
9: **end for**
10: **end for**
11: **return** coalition C

At the end of each iteration, Equation 4.2 determines the quality of the achieved solution, where $Best_{\tau_i}$ is the evaluation values of the selected service proposals, $|S_i|$ is the number of independent tasks in S_i that can be remotely executed, and $|coalition|$ is the number of tasks which already have a selected service proposal.

$$Q_{coalition} = \left\lfloor \frac{|coalition|}{|S_i|} \right\rfloor * \sum_{i=1}^{|coalition|} \frac{1 - Best_{\tau_i}}{|coalition|} \tag{4.2}$$

Note that according to Equation 4.2, the quality of an incomplete coalition for the set of tasks $\tau_i \in S_i$ that can be remotely executed is zero.

The algorithm terminates when all the received proposals are evaluated or if it finds that the quality of a coalition cannot be further improved because all the tasks $\tau_i \in S_i$ will be served at the user's preferred QoS level $L_{desired}$.

4.3.1 Formal description of the algorithm's anytime behaviour

The coalition formation's anytime behaviour can be formally described using the set of axioms presented in [vHtT00]. The authors describe the anytime functionality of an

algorithm using four axioms, each of which describes a different aspect of the anytime behaviour as follows:

Axiom 4.3.1.1 (Initial behaviour) *There is an initial period during which the algorithm does not produce a coalition for a cooperative service execution*

The algorithm does not immediately produces an intermediate solution, since it must first analyse a service proposal for each task $\tau_i \in S_i$ that can be remotely executed. If t' indicates the duration of this initial step then, if interrupted at any time $t < t'$, the algorithm will not be able to return a valid solution and the achieved quality will be zero.

$$\forall_{t<t'}\ Q_{coalition}(t) = 0$$

Axiom 4.3.1.2 (Growth direction) *The quality of a coalition only improves with increasing run time*

Algorithm 4 ensures that the coalition's members are only updated if and only if a better proposal for a task $\tau_i \in S_i$ that can be remotely executed is found. As such, the quality of the currently determined coalition can only improve with time.

$$\forall_{t'>t}\ Q_{coalition}(t) \leq Q_{coalition}(t')$$

Axiom 4.3.1.3 (Growth rate) *The amount of increase in the coalition's quality varies during computation*

Algorithm 4 selects for evaluation, at each iteration, the service proposal sent by the node with the highest local reward. Such heuristic selection has the highest probability of choosing proposals closer to the user's preferred QoS values. Then, it is expected that a coalition's quality will rapidly increase in the first steps of the algorithm and its growth rate should diminish over time.

$$\forall_{t'>t}\ Q_{coalition}(t+1) - Q_{coalition}(t) > Q_{coalition}(t'+1) - Q_{colaition}(t')$$

Axiom 4.3.1.4 (End condition) *After evaluating all candidate proposals the algorithm achieves its full functionality*

If the time required to evaluate a candidate proposal is t_e, the total required runtime of the anytime algorithm is the sum of all n evaluations. After $n * t_e$, Algorithm 4 will produce exactly the same solution quality as its traditional version proposed in Chapter 3 which only produces a solution with quality $Q'_{coalition}$ at the end of its computation time.

$$Q_{coalition}(n * t_e) = Q'_{coalition}$$

4.3.2 Conformity with the desirable properties of anytime algorithms

The conformity of the proposed anytime coalition formation algorithm with the desirable properties of anytime algorithms discussed in Section 4.2.1 is checked in the next paragraphs.

Property 4.3.2.1 (Measurable quality) *A coalition's quality can be determined precisely*

Proof 4.3.2.1 *According to Equation 4.2, the quality of the generated coalition at each iteration of the algorithm can be directly computed from the evaluation values of the best service proposals for each of the service's tasks.*

□

Property 4.3.2.2 (Recognisable quality) *The quality of a coalition can be easily determined at run time*

Proof 4.3.2.2 *Let $S_i = \tau_1, \ldots, \tau_n$ be the set of n tasks under negotiation for a cooperative execution.*

A coalition c is only updated to c′ when a better proposal for a task $\tau_i \in S_i$ is found, by replacing the previously selected service proposal P_{ki} from node N_k with $P_{k'i}$ from node $N_{k'}$.

Let $|c|$ be the size of the generated coalition to cooperatively execute service S_i, $E_{P_{k'i}}$ be the evaluation value of the new service proposal $P_{k'i}$, and $E_{P_{ki}}$ be the evaluation value of the previously selected service proposal P_{ki}.

The quality of the updated coalition $Q_{c'}$ can be quickly determined by adding the quality Q_c achieved by coalition c to the weighted difference between $E_{P_{k'i}}$ and $E_{P_{ki}}$.

$$Q_{c'} = Q_c + \frac{E_{P_{k'i}} - E_{P_{ki}}}{|c|}$$

This makes the determination of the new coalition's quality straightforward and within a constant time.

□

Property 4.3.2.3 (Monotonicity) *The quality of the generated coalition is a non-decreasing function of time*

Proof 4.3.2.3 *Node N_k is only added to a coalition if and only if it proposes a better service for task $\tau_i \in S_i$, that is, if it is closer to the user's quality preferences than the best service proposal found so far.*

The algorithm always returns the coalition formed by the best service proposals evaluated until time t, which can be different from the last set of evaluated proposals. According to Zilberstein [Zil96], this characteristic in addition to a recognisable quality is sufficient to prove the monotonicity of an anytime algorithm.

□

Property 4.3.2.4 (Consistency) *For a given amount of computation time on a given input, the quality of the generated coalition is always the same*

Proof 4.3.2.4 *For a given amount of computation time Δ_t on a given input of a set of service proposals P and user's QoS preferences Q_i for service S_i, the quality of the selected coalition for a cooperative service execution is always the same, since the selection of candidate proposals for evaluation is deterministic.*

According to Equation 4.1, for each task $\tau_i \in S_i$, the next proposal P_{ki} to be selected for evaluation is the one sent by the node with the greatest local reward. As such, the algorithm guarantees a deterministic output quality for a given amount of time and input.

□

Property 4.3.2.5 (Diminishing returns) *The improvement in the generated coalition's quality is larger at the early stages of the computation and it diminishes over time*

Proof 4.3.2.5 *The quality of each generated coalition, given by Equation 4.2, is measured using the evaluation values of the best proposals for each task $\tau_i \in S_i$. The best proposal is the one that contains the attributes' values more closely related to the user's specific QoS preferences Q_i, in all QoS dimensions.*

Each node's local reward, determined with Equation 3.7, expresses a degree of satisfaction for all the users that have tasks being locally executed with specific QoS levels, including the service being currently negotiated.

Selecting for evaluation, for each task $\tau_i \in S_i$, the proposal sent by the node that achieved the highest local reward is expected to rapidly improve the quality of the generated coalition at an early stage of execution. Nevertheless, some other node may propose a better service for the service request under negotiation at the expense of a higher QoS downgrade of previously accepted services, thus achieving a lower local reward. As such, it is still possible that the solution's quality can be further improved in the next iterations of the algorithm, but at a lower increment rate.

□

Property 4.3.2.6 (Interruptibility) *The algorithm can be stopped at any time and still be able to provide a solution*

Proof 4.3.2.6 *Let t' be the time needed to generate an initial coalition. By Axiom 4.3.1.1 it is known that if interrupted at any time $t < t'$ the algorithm will not be able to return a valid solution, resulting in zero quality.*

However, when stopped at any time $t > t'$ the algorithm always returns the coalition with the highest quality determined until time t, which can be different from the last set of evaluated proposals.

□

Property 4.3.2.7 (Preemptibility) *The algorithm can be suspended and resumed with minimal overhead*

Proof 4.3.2.7 *Since the algorithm keeps both the set of received proposals not yet evaluated until time t and the determined coalition, it can be easily resumed after an interrupt.*

□

4.4 Anytime service proposal formulation

This section reformulates the traditional service proposal formulation algorithm proposed in Chapter 3 as an iterative refinement algorithm that can be interrupted at any time and still returns a feasible service solution. Recall that requests for a cooperative service execution arrive dynamically at any node and are formulated as a set of acceptable multi-dimensional QoS levels in decreasing preference order.

In order to be useful in practice, the proposed anytime approach must try to quickly find a sufficiently good initial proposal and gradually improve it if time permits, conducting the search for a better feasible solution in a way that maximises the expected improvement in the solution's quality [Zil96]. As such, the proposed QoS optimisation algorithm, starts by keeping the QoS levels of previously accepted services and selects the lowest requested QoS level for the new requesting tasks $\tau_i \in S_i$. Note that this is the service configuration with the highest probability of being feasible without degrading the current level of service of previously accepted tasks.

As such, the proposed anytime approach clear splits the formulation of a new set of SLAs in two different scenarios. The first one, detailed in Algorithm 5, involves serving the new task without changing the QoS level of previously guaranteed tasks. The second one, detailed in Algorithm 6, due to the lack of resources, demands degrading the currently provided level of service of the previously accepted tasks in order to accommodate the new requesting task.

After quickly determining this initial service solution, the search of a better solution is guided, at each iteration, by the maximisation of the new task's QoS level and by the minimisation of the QoS degradation of the previously accepted services. When τ_i can be accommodated without degrading the QoS of the previously accepted tasks, the configuration that maximises τ_i's reward increase is selected (Step 1). On the other hand, when QoS degradation is needed to accommodate τ_i, the algorithm incrementally finds the minimal service degradation for the previously accepted services until a feasible solution is found (Step 2).

Algorithm 5 Anytime service proposal formulation - Step 1

Let τ^e be the set of previously accepted tasks whose stability period Δ_t has expired.
Let τ^p be the set of all previously accepted tasks whose current QoS cannot be changed.
Let τ^a be the newly arrived task.
Each task $\tau_i \in \tau^e \cup \tau^a \cup \tau^p$ has associated a set of user's defined QoS constraints Q_i.
Each Q_{kj} is a finite set of n quality choices for the j^{th} attribute, expressed in decreasing preference order, for all k QoS dimensions.
Let σ be the determined set of SLAs, updated at each step of the algorithm

Step 1 - Maximise the QoS level of the newly arrived task τ^a

1: Define SLA_{τ_a} by selecting the lowest requested QoS level $Q_{kj}[n]$, for all the j attributes of the k QoS dimensions for the newly arrived task τ^a
2: Keep the current QoS level for each task $\tau_k \in \tau^e$
3: Update the current set of SLAs σ ▷ $\sigma = \sigma \cup SLA_{\tau_a}$
4: **while** $feasibility(\sigma) =$ **TRUE do**
5: **for** each j^{th} attribute of any k QoS dimension in τ_a with value $Q_{kj}[m] > Q_{kj}[0]$ **do**
6: Determine the utility increase by upgrading attribute j to the next possible value $Q_{kj}[m-1]$
7: **end for**
8: Find maximum increase and define SLA'_{tau_a} for task τ_a by upgrading attribute x to the $Q_{kj}[m-1]$'s level
9: Update the current set of promised SLAs σ ▷ $\sigma = \sigma \setminus SLA_{\tau_{min}} \cup SLA'_{\tau_{min}}$
10: **end while**

At each iteration, the quality of the proposed solution can be measured by considering the reward achieved by the new arriving task r_{τ_i}, the impact on the provided QoS of the n previously accepted tasks r_{τ^p} and the value of the previously generated feasible configuration Q'_{conf}, using Equation 4.3. Initially, Q'_{conf} is set to zero and its value is only updated if the iteration's solution is feasible.

$$Q_{conf} = \left(r_{\tau_i} * \frac{\sum_{i=0}^{n} r_{\tau^p}}{n} \right)^{(1-Q'_{conf})} \tag{4.3}$$

Algorithm 6 Anytime service proposal formulation - Step 2

Step 2 - Find local minimal service degradation to accommodate τ^a

11: **while** $feasibility(\sigma) \neq$ **TRUE do**
12: **for** each task $\tau_i \in \{\tau^e \cup \tau^a\}$ **do**
13: **for** each j^{th} attribute of any k QoS dimension in τ_a with value $Q_{kj}[m] > Q_{kj}[n]$ **do**
14: Determine the reward decrease by downgrading attribute j to $Q_{kj}[m+1]$
15: **end for**
16: **end for**
17: Find task τ_{min} whose reward decrease is minimum
18: Define $SLA'_{\tau_{min}}$ for task τ_{min} with the new value $Q_{yx}[m+1]$ for attribute x of the QoS dimension y
19: Update the current set of promised SLAs σ $\triangleright$ $\sigma = \sigma \setminus SLA_{\tau_{min}} \cup SLA'_{\tau_{min}}$
20: **end while**
21: **return** the new local set of promised SLAs σ

The algorithm can be interrupted at any time as a consequence of the dynamic nature of the environment [NP07a, NP08c], or finishes when it finds a feasible set of QoS configurations whose quality cannot be further improved, or when it finds that even if all the tasks would be served at the lowest admissible QoS level it is not possible to accommodate the new requesting tasks in w_{ij}. In the latter case, the service request is rejected and the previously accepted tasks continue to be served at their current QoS levels.

The proposed anytime QoS optimisation algorithm always improves or maintains the current solution's quality as it has more time to run. This is done by keeping the best feasible solution found so far, if the result of each iteration is not always proposing a feasible service configuration for the new task set. However, each intermediate configuration, even if not feasible, is used to calculate the next solution, minimising the search effort.

The next simple example denotes this behaviour. Admit that the algorithm runs to completion or it is interrupted after its fifth iteration (Table 4.1). With this set of iterations, the algorithm would return the solution found at the fifth iteration rather than the second one, since it is the one with the greatest quality for the new service under negotiation. The second solution would only be returned as the best feasible solution if the algorithm was interrupted before it was able to complete its

fifth iteration.

Iteration	Q_{conf}	Feasible?
1^{st}	$(0.1 * 0.8)^{(1-0)} = 0.08$	yes
2^{nd}	$(0.2 * 0.8)^{(1-0.08)} = 0.185$	yes
3^{rd}	$(0.3 * 0.8)^{(1-0.185)} = 0.313$	no
4^{th}	$(0.3 * 0.75)^{(1-0.185)} = 0.297$	no
5^{th}	$(0.3 * 0.7)^{(1-0.185)} = 0.280$	yes

Table 4.1: Iterative QoS optimisation

In the next sections, although similar to what has been demonstrated for the anytime coalition formation algorithm, for the sake of completeness, we also describe the different aspects of the anytime functionality of the proposed service proposal formulation algorithm using the four axioms presented in [vHtT00] and its conformity with the desired properties of anytime algorithms presented in [Zil96].

4.4.1 Formal description of the algorithm's anytime behaviour

Axiom 4.4.1.1 (Initial behaviour) *Until a feasible set of SLAs is found the new task is rejected*

Clearly, an intermediate solution can only be considered if it produces a feasible set of SLAs. If t' indicates the time at which the first feasible solution is found then, if interrupted at anytime $t < t'$, the algorithm will reject the new task and the quality of the determined configuration will be zero.

$$\forall_{t<t'}\ Q_{conf}(t) = 0$$

Axiom 4.4.1.2 (Growth direction) *The quality of a feasible set of SLAs can only improve over time*

At each iteration, the proposed algorithm only considers a new feasible set of SLAs as the currently found solution if and only if it improves the solution's quality. When the new requesting task τ_i can be accommodated without degrading the QoS of the previously accepted tasks, the configuration that maximises τ_i's reward increase is selected. On the other hand, when QoS degradation is needed to accommodate τ_i,

the algorithm incrementally finds the minimal service degradation for the previously accepted services until a feasible solution is found.

$$\forall_{t'>t}\ Q_{conf}(t) \leq Q_{conf}(t')$$

Axiom 4.4.1.3 (Growth rate) *The amount of increase in the solution's quality varies during computation*

The solution's quality is expected to rapidly increase in the first steps of the algorithm and its growth rate should diminish over time as the algorithm starts by improving the new user's preferred quality attributes until an unfeasible set of SLAs is found or the new task can be served at the user's preferred QoS level.

On the other hand, when QoS degradation is needed in the search for a new feasible solution, the algorithm degrades the less important attributes for all services being locally executed.

$$\forall_{t'>t}\ Q_{conf}(t+1) - Q_{conf}(t) > Q_{conf}(t'+1) - Q_{conf}(t')$$

Axiom 4.4.1.4 (End condition) *When it is not possible to improve the solution's quality the algorithm achieves its full functionality*

When it runs to completion, the anytime version of the algorithm will produce exactly the same solution as its traditional version proposed in the previous chapter that only produces a solution with quality Q'_conf at the end of its computation time.

The anytime version terminates when it finds a set of QoS levels that keeps all tasks feasible and the quality of that solution can not be further improved, or when it finds that, even at the lowest QoS level for each task, the new set is unfeasible.

If the time required to improve or degrade an attribute and test for the schedulability of the solution is given by t_s, the total required runtime of the anytime algorithm is the sum of all n needed changes in attributes to find the best feasible solution.

$$Q_{conf}(n * t_s) = Q'_{conf}$$

4.4.2 Conformity of with the desirable properties of anytime algorithms

Property 4.4.2.1 (Measurable quality) *The quality of a SLA can be determined precisely*

Proof 4.4.2.1 *At each iteration of the algorithm, Equation 4.3 measures the quality of the proposed SLA by considering the proximity of the proposal with respect to the user's request under negotiation and the impact of that proximity on the global utility achieved by the previously accepted tasks.*

□

Property 4.4.2.2 (Recognisable quality) *The quality of a set of SLAs can be easily determined at run time*

Proof 4.4.2.2 *The quality of each generated feasible set of SLAs is determined by using the rewards achieved by all tasks being locally executed, which includes the newly arrived one. Using Equation 3.6, the rewards' computation is straightforward and time-bounded.*

□

Property 4.4.2.3 (Monotonicity) *The quality of the generated set of SLAs is a nondecreasing function of time*

Proof 4.4.2.3 *The algorithm produces a new set of SLAs at each iteration, as it tries to maximise the utility increase for the new requesting task while minimising the utility decrease for all previously accepted tasks. It may happen, due to resource limitations, that the generated set of SLAs at the end of an iteration is not feasible. Since a service proposal can only be considered useful within a feasible set of tasks, the algorithm always returns the best found feasible solution rather than the last generated SLA.*

According to Zilberstein [Zil96], this characteristic in addition to a recognisable quality is sufficient to prove the monotonicity of an anytime algorithm.

□

Property 4.4.2.4 (Consistency) *For a given amount of computation time on a given input, the quality of the generated SLA is always the same*

Proof 4.4.2.4 *For a given amount of computation time Δ_t on a given input of a set of QoS constraints Q associated with a set of tasks τ, the quality of the proposed set*

SLA for the new task τ_i is always the same, since the selection of attributes to improve or degrade at each iteration is deterministic.

At each iteration, the QoS attribute selected to be improved is the one that maximises an increase in the reward achieved by the new arrived task τ_i, while the QoS attribute selected to be downgraded is the one that minimises the decrease in the global reward achieved by all tasks being locally executed. As such, the algorithm guarantees a deterministic output quality for a given amount of time and input.

□

Property 4.4.2.5 (Diminishing returns) *The improvement in the quality of the generated SLA is larger at the early stages of the computation and it diminishes over time*

Proof 4.4.2.5 *An initial solution that keeps the QoS levels of the previously guaranteed tasks τ^p and selects the worst requested level in all QoS dimensions for the new arrived task τ_i is quickly generated. Its quality is given by Equation 4.3, considering the rewards achieved by all tasks.*

At each iteration, the currently found solution is improved by either upgrading the QoS attribute that maximises an increase in τ_i's utility or by downgrading the QoS attribute that minimises the decrease in the utility of the local set of tasks. As such, the increment in the solution's quality is expected to be larger at the firsts iterations and it diminishes over time.

□

Property 4.4.2.6 (Interruptibility) *The algorithm can be stopped at any time and still provide a solution*

Proof 4.4.2.6 *Let t' be the time needed to generate the first feasible solution. By Axiom 4.4.1.1 its is known that if interrupted at any time $t < t'$, the algorithm will not be able to return a new set of SLAs and the quality will be zero.*

Nevertheless, when interrupted at time $t > t'$ the algorithm returns the best feasible set of SLAs generated until t, which can be different from the last evaluated set.

□

Property 4.4.2.7 (Preemptibility) *The algorithm can be suspended and resumed with minimal overhead*

Proof 4.4.2.7 *Since the algorithm keeps the best generated feasible solution and the configuration values determined at the last iteration it can be easily resumed after an interrupt.*

□

4.5 Anytime upgrade of previously downgraded SLAs

This section reformulates the traditional QoS re-upgrade algorithm described in the previous chapter as an iterative refinement algorithm that can be interrupted at any time and still returns a feasible service solution, detailed in Algorithm 7.

Recall that in the previous chapter we have defined that the system periodically forecasts the granted stability period for each locally accepted service S_i and if the user's stability requirements are met, the currently provided service's SLA is upgraded. Otherwise, the service is kept in its current QoS level and the pre-reserved resource amounts for the computed upgrades become immediately available for subsequent QoS negotiations.

The search for the set of possible service upgrades is guided, at each iteration, by the maximisation of each task's reward increase. The algorithm can be interrupted at any time or finishes when it finds a feasible set of QoS configurations whose quality cannot be further improved due to resource limitations or all the initially granted SLAs are reached.

At each iteration, the achieved solution's quality is measured by Equation 4.4, considering the rewards achieved by the new upgraded SLAs for the previously downgraded tasks r_{τ^e} and the value of the previous generated feasible configuration Q'_{conf}. Initially, Q'_{conf} is set to zero and its value is only updated if the achieved solution is feasible.

$$Q_{conf} = \left(\frac{\sum_{i=0}^{n} r_{\tau^e}}{n} \right)^{(1-Q'_{conf})} \tag{4.4}$$

In the next sections, although similar to what has been demonstrated for the anytime

Algorithm 7 Anytime QoS re-upgrade of previously downgraded tasks

Let τ^e be the set of previously downgraded tasks whose Δ_t has expired
Let τ^p be the set of all previously accepted tasks whose current QoS cannot be changed.
Each task $\tau_i \in \tau^e \cup \tau^p$ has associated a set of user's defined QoS constraints Q_i.
Let $Q_{kj}[init]$ be the initially provided and $Q_{kj}[i]$ the currently provided level of service for attribute j of the k_{th} QoS dimension for task $\tau_i \in \tau^e$
Let σ be the determined set of SLAs, updated at each step of the algorithm

1: **while** $feasibility(\sigma) =$ **TRUE do**
2: **for** each task $\tau_i \in \tau^e$ **do**
3: **for** each j^{th} attribute of any k QoS dimension with value $Q_{kj}[m] > Q_{kj}[init]$ **do**
4: Determine the utility increase by upgrading attribute j to $Q_{kj}[m-1]$
5: **end for**
6: **end for**
7: Find task τ_{max} whose reward increase is maximum
8: Define $SLA'_{\tau_{max}}$ for task τ_{max} with the new value $Q_{yx}[m-1]$ for attribute x of the QoS dimension y
9: Update the current set of promised SLAs σ $\triangleright$ $\sigma = \sigma \setminus SLA_{\tau_{max}} \cup SLA'_{\tau_{max}}$
10: **end while**
11: **return** the new local set of promised SLAs σ

coalition formation algorithm and the anytime service proposal formulation, for the sake of completeness, we also describe the different aspects of the anytime functionality of the proposed service proposal formulation algorithm using the four axioms presented in [vHtT00] and its conformity with the desired properties of anytime algorithms presented in [Zil96].

4.5.1 Formal description of the algorithm's anytime behaviour

Axiom 4.5.1.1 (Initial behaviour) *Until a feasible set of SLAs is found no new set of possible upgrades is returned*

Clearly, a new set of upgraded SLAs can only be considered within a feasible set of service configurations. If t' indicates the time at which the first feasible solution is

found then, if interrupted at anytime $t < t'$, the algorithm will not be able to provide any set of possible service upgrades.

$$\forall_{t<t'}\ Q_{conf}(t) = 0$$

Axiom 4.5.1.2 (Growth direction) *The quality of a feasible set of upgraded SLAs can only improve over time*

At each iteration, the algorithm iteratively maximises each task's reward increase, by upgrading the most desired QoS attributes for the user. As such, if a new set of feasible SLAs is found, it can only be one with a higher global utility.

$$\forall_{t'>t}\ Q_{conf}(t) \leq Q_{conf}(t')$$

Axiom 4.5.1.3 (Growth rate) *The amount of increase in the solution's quality varies during computation*

Since the algorithm starts by upgrading the users' preferred QoS attributes it is expected that quality of the produced solution rapidly increases in the first steps of the algorithm and its growth rate diminishes over time.

$$\forall_{t'>t}\ Q_{conf}(t+1) - Q_{conf}(t) > Q_{conf}(t'+1) - Q_{conf}(t')$$

Axiom 4.5.1.4 (End condition) *When it is not possible to improve the solution's quality the algorithm achieves its full functionality*

When it runs to completion, the anytime version of the algorithm will produce exactly the same solution as the traditional QoS re-upgrade discussed in the previous chapter that only produces a solution with quality Q'_conf at the end of its computation time.

The anytime version terminates its computation when it finds a set of QoS levels that keeps all tasks feasible and the quality of that solution can not be further improved. If the time required to improve an attribute and test for the schedulability of the solution is given by t_s, the total required runtime of the anytime algorithm is the sum of all n needed changes in attributes to find the best feasible solution.

$$Q_{conf}(n * t_s) = Q'_{conf}$$

4.5.2 Conformity with the desirable properties of anytime algorithms

Property 4.5.2.1 (Measurable quality) *The quality of an upgraded set of SLAs can be determined precisely*

Proof 4.5.2.1 *At each iteration of the algorithm, Equation 4.3 measures the quality of the proposed set of upgraded SLAs by considering the proximity of the new set of service proposals with respect to the users' preferred QoS levels.*

□

Property 4.5.2.2 (Recognisable quality) *The quality of an upgraded set of SLAs can be easily determined at run time*

Proof 4.5.2.2 *The quality of each generated feasible set of upgraded SLAs is determined by using the rewards achieved by all tasks being locally executed. Using Equation 3.6, the rewards' computation is straightforward and time-bounded.*

□

Property 4.5.2.3 (Monotonicity) *The quality of the generated set of upgraded SLAs is a nondecreasing function of time*

Proof 4.5.2.3 *The algorithm produces a new set of SLAs at each iteration, as it tries to maximise the utility increase of all previously downgraded tasks. Since an upgraded set of SLAs can only be considered useful within a feasible set of service configurations, the algorithm always returns the best found feasible solution rather than the last generated set of upgraded SLAs.*

According to Zilberstein [Zil96], this characteristic in addition to a recognisable quality is sufficient to prove the monotonicity of an anytime algorithm.

□

Property 4.5.2.4 (Consistency) *For a given amount of computation time on a given input, the quality of the generated upgraded set of SLAs is always the same*

Proof 4.5.2.4 *For a given amount of computation time Δ_t on a given input of a set of QoS constraints Q associated with a set of tasks τ, the quality of the proposed set of upgraded SLAs is always the same, since the selection of attributes to improve each iteration of the algorithm is deterministic.*

At each iteration, the QoS attribute selected to be improved is the one that maximises an increase in the node's local reward. As such, the algorithm guarantees a deterministic output quality for a given amount of time and input.

□

Property 4.5.2.5 (Diminishing returns) *The improvement in the quality of the generated set of upgraded SLAs is larger at the early stages of the computation and it diminishes over time*

Proof 4.5.2.5 *At each iteration, the currently found service solution is improved by upgrading the QoS attribute that maximises an increase in the node's utility. As such, the increment in the solution's quality is expected to be larger at the firsts iterations and it diminishes over time.*

□

Property 4.5.2.6 (Interruptibility) *The algorithm can be stopped at any time and still provide a solution*

Proof 4.5.2.6 *Let t' be the time needed to generate the first upgraded set of SLAs. By Axiom 4.5.1.1 it is known that if interrupted at any time $t < t'$ the algorithm will not be able to return a new set or upgraded SLAs.*

However, when stopped at any time $t > t'$, the algorithm returns the best feasible set of upgraded SLAs generated until time t.

□

Property 4.5.2.7 (Preemptibility) *The algorithm can be suspended and resumed with minimal overhead*

Proof 4.5.2.7 *Since the algorithm keeps the best generated feasible solution and the configuration values determined at the last iteration it can be easily resumed after an interrupt.*

□

4.6 Summary

As an increasing number of end users runs both real-time and traditional desktop applications in the same distributed embedded system, the issue of how to provide an efficient Quality of Service (QoS) control in a highly dynamic, open, shared, and heterogeneous environment becomes very important.

However, finding an optimal distributed service provisioning that deals with both users' and service providers' quality constraints can be extremely complex and impossible to achieve in a useful and bounded time. Unlike conventional QoS optimisation algorithms that guarantee a correct output only after termination, this chapter proposed an anytime approach that does not rely on the availability of the complete deliberation time to provide a service solution and a measure of its quality. Nodes start by negotiating partial, acceptable service proposals that are latter refined if time permits, with a quality which is expected to improve as the run time of the algorithm increases.

Contrary to a traditional QoS optimisation, the proposed anytime approach considers the needed tradeoff between the level of optimisation and the usefulness of an optimal runtime system's adaptation behaviour. Such tradeoff is a powerful and useful approach in open dynamic real-time systems where, despite their uncertainty, responses to events still have to be provided within precise timing constraints.

Chapter 5

Scheduling tasks in open systems

The basic assumptions made on classical real-time scheduling theory are no longer valid in new open and dynamic embedded systems. A new approach is needed to handle the dynamic changes of services' requirements in a predictable fashion, enforcing timing constraints with a certain degree of flexibility, aiming to achieve the desired tradeoff between predictable performance and an efficient use of resources.

This chapter proposes a dynamic server-based scheduler that supports the coexistence of guaranteed and non-guaranteed bandwidth servers to efficiently handle soft-tasks' overloads by making additional capacity available from two sources: (i) residual capacity allocated but unused when jobs complete in less than their budgeted execution time; (ii) stealing capacity from inactive non-isolated servers used to schedule best-effort jobs.

5.1 Introduction

As an increasing number of end users runs both real-time and traditional desktop applications in the same system, the issue of how to provide an efficient resource utilisation in this highly dynamic, open, and shared environment becomes very important. The need arises from the fact that independently developed services can enter and leave the system at any time, without any previous knowledge about their real execution requirements and tasks' inter-arrival times.

For most of these systems, the classical real-time approach based on a rigid off-line design and worst-case execution time (WCET) assumptions would keep resources

unused for most of the time. Usually, tasks' WCET is rare and much longer than the average case. At the same time, it is increasingly difficult to compute WCET bounds in modern hardware without introducing excessive pessimism [CP03]. Such a waste of resources can only be justified for very critical systems in which a single missed deadline may have catastrophic consequences.

A well known technique for limiting the effects of overruns, when a task needs to execute more than its guaranteed reserved time, was proposed by Abeni and Buttazo [AB98]. The Constant Bandwidth Server (CBS) scheduler handles soft real-time requests with a variable or unknown execution behaviour under the Earliest Deadline First (EDF) [LL73] scheduling policy. To avoid unpredictable delays on hard real-time tasks, soft tasks are isolated through a bandwidth reservation mechanism, according to which each soft task gets a fraction of the CPU and it is scheduled in such a way that it will never demand more than its reserved bandwidth, independently of its actual requests. This is achieved by assigning each soft task a deadline, computed as a function of the reserved bandwidth and its actual requests. If a task requires to execute more than its expected computation time, its deadline is postponed so that its reserved bandwidth is not exceeded. As a consequence, overruns occurring on a served task will only delay that task, without compromising the bandwidth assigned to other tasks.

However, with CBS, if a server completes a task in less than its budgeted execution time no other server is able to efficiently reuse the amount of computational resources left unused. To overcome this drawback, CBS has been extended by several resource reclaiming schemes [LB00, CBS00, MLBC04, CBT05, LB05], proposed to support an efficient sharing of computational resources left unused by early completing tasks. Such techniques have been proved to be successful in improving the response times of soft real-time tasks while preserving all hard real-time constraints.

Nevertheless, not all computational tasks in modern open real-time systems follow a traditional periodic pattern. For example, aperiodic complex optimisation tasks may take varying amounts of time to complete depending on the desired solution's quality or current state of the environment. Some examples can be found in [Haw03, ACS03, SCC04, BB04, vdBFK06] or in the anytime algorithms discussed in the previous chapter. Furthermore, the existing reclaiming schemes are unable to reduce isolation in a controlled fashion and donate reserved, but still unused, capacities to currently overloaded servers.

Based upon a careful study of the ways in which unused reserved capacities can be more efficiently used to meet deadlines of tasks whose resource usage exceeds their

reservations, this chapter presents the Capacity Sharing and Stealing (CSS) scheduler.

CSS considers the coexistence of the traditional *isolated* servers with a novel *non-isolated* type of servers, combining an efficient reclamation of residual capacities with a controlled isolation loss. The goal is to reduce the mean tardiness of periodic guaranteed jobs by handling overloads with additional capacity available from two sources: (i) by reclaiming unused allocated capacity when jobs complete in less than their budgeted execution time; and (ii) by stealing allocated capacities from inactive non-isolated servers used to schedule aperiodic best-effort jobs.

CSS is partially presented in [NP07a]. The integration of CSS into the CooperatES framework is discussed in [NP06a].

5.2 System model

The work presented in this chapter focus on dynamic open real-time systems where all accepted tasks execute on a single shared processor, the sum of the reserved capacities is no more than the maximum capacity of the processor, and the scheduler does not have any previous and complete knowledge about the services' execution requirements.

When a new service arrives to a node, requiring a certain amount of resources based on expected average needs, an admission test is run. If, given the current system's load, the required amount can be guaranteed, the service is accepted and the requested amount is reserved.

A service can be composed by a set of independent real-time and non-real-time tasks which can generate a virtually infinite sequence of jobs. The j^{th} job of task τ_i arrives at time $a_{i,j}$, is released to the ready queue at time $r_{i,j}$, and starts to be executed at time $s_{i,j}$ with deadline $d_{i,j} = a_{i,j} + T_i$, with T_i being the period of τ_i. The arrival time of a particular job is only revealed at runtime and the exact execution requirements $e_{i,j}$, as well as which resources will be accessed and by how long they will be held, can only be determined by actually executing the job to completion until time $f_{i,j}$. These times are characterised by the relations $a_{i,j} \leq r_{i,j} \leq s_{i,j} \leq f_{i,j}$.

Each accepted real-time task τ_i is associated to a CSS server S_i characterised by a pair (Q_i, T_i), where Q_i is the server's maximum reserved capacity and T_i its period. Note that these values are based on average estimations for soft real-time tasks.

At any given time, it is selected for execution the server with the earliest deadline and pending work to do, based on the EDF [LL73] priority assignment. When no server is

selected, the processor is idle or it is executing non-real time tasks.

5.3 The Capacity Sharing and Stealing (CSS) approach

The CSS scheduler [NP07a] integrates and extends some of the best principles of previous scheduling approaches to improve the responsiveness of soft real-time tasks in the presence of overruns while ensuring that the schedulability of hard tasks is not compromised. To ease the algorithm's discussion, the main principles of the proposed approach are discussed in the next paragraphs and the CSS scheduler is formally presented in Section 5.3.1.

All tasks in the system are assumed to be independent and no task is allowed to suspend itself waiting for a shared resource or a synchronisation event. The system consists of n servers and a global scheduler based on the EDF priority assignment. A single ready queue exists and, at each instant, the active server with the earliest deadline S_i is selected and its corresponding task τ_i is dispatched to execute.

A CSS server S_i is characterised by an ordered pair (Q_i, T_i), where Q_i is the server's maximum reserved capacity and T_i is the server's period. At each instant, the following values are associated with each server: its currently assigned deadline $d_{i,k}$, its currently available capacity c_i, the amount of residual capacity c_r that can be reclaimed by other servers, and its currently assigned capacity recharging time r_i.

The algorithm considers two different types of servers: *isolated* servers used to schedule periodic and sporadic guaranteed tasks and *non-isolated* servers for aperiodic best-effort tasks. For an isolated server, the amount of reserved capacity Q_i is ensured to be available every period T_i, while an inactive non-isolated server can have some or all of its reserved capacity Q_i stolen by a needed overloaded server.

By setting a specific recharging time r_i rather than automatically replenish the server's capacity and update its deadline on every capacity exhaustion, CSS follows a hard reservation approach (please refer to [RJM+98] for a description of hard vs soft reservations).

Recall that CBS presents some drawbacks when serving tasks that are active for long intervals of time, covering therefore many periods of a server. Since CBS automatically recharges a server's capacity and postpones its deadline on every capacity exhaustion, if the server's deadline, although postponed, is still the earliest, the renewed capacity

can be used within the same period. This leads to a temporal over execution that may be followed by a starvation period, altering the rate of periodic tasks.

As such, advancing the recharging times when there is pending work is against our purpose of executing periodic activities with stable frequencies. Note that if pending jobs are a consequence of early arrivals, executing periodic services with a stable frequency suggests that those early arrived jobs should only begin their execution in the expected period of arrival. Furthermore, a hard reservation approach enables a server whose capacity as been exhausted to continue its execution either by stealing capacities from inactive non-isolated servers or by using any new residual capacities that eventually is released until its currently assigned deadline, keeping its current priority.

At time t, a server S_i is said to be *active* if (i) the served task is ready to execute; (ii) is executing; or (iii) the server is supplying its residual capacity to other servers until its currently assigned deadline $d_{i,k}$. Otherwise, S_i is *inactive* if (i) there are no pending jobs to serve; and (ii) the server has no residual capacity to supply to the other servers.

State transitions are determined by the (i) arrival of a new job, (ii) capacity exhaustion, or (iii) non-existence of pending jobs at replenishment time (Figure 5.1). An inactive server becomes active with the arrival of the new j^{th} job at time $a_{i,j}$, if $a_{i,j} \geq d_{i,j-1}$. If $a_{i,j} < d_{i,j-1}$, the job is only released at the next server's replenishment instant r_i. On the other hand, an active server becomes inactive if (i) all its reserved capacity is consumed and there are no pending jobs to serve (capacity exhaustion can occur while a server is supplying its residual capacity to other servers or is using its capacity to advance a job's execution); or (ii) there are no pending jobs at replenishment time.

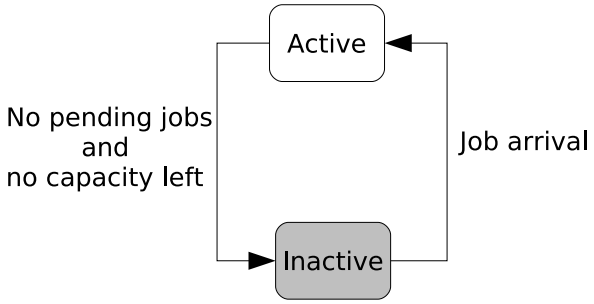

Figure 5.1: State transitions of CSS servers

For each server S_i, when $t = r_i$, the action to be taken depends on the existence, at time t, of pending jobs to be executed, that is, if there is a job $J_{i,k}$ such that

$a_{i,k} \leq t < f_{i,k}$. An active server without pending work (it must be supplying its residual capacity to other servers) becomes inactive and its residual capacity is discharged. On the other hand, for a server with pending jobs, a new deadline is generated to $d_{i,k} = max\{a_{i,k}, d_{i,k-1}\} + T_i$, the server's capacity is replenished to its maximum value $c_i = Q_i$, the recharging time is set to the server's new deadline $r_i = d_{i,k}$, and the server's residual capacity is set to zero $c_r = 0$.

Whenever a job is completed, the remaining reserved capacity of its dedicated server can (and should) be used by any active task to advance its execution. Although the idea of residual capacity reclaiming is not new, CSS proposes to efficiently reclaim unused computation times as early as possible. Note that when using a residual capacity of another server, a task must be scheduled using the current deadline of the server to which the residual capacity belongs to. Since reserved capacities expire at their deadlines, it makes sense to reclaim residual capacities before consuming the server's own reserved capacity Q_i in order to increase the probability of effectively using them.

Let A be the set of all active servers. The set of active servers A_r eligible for residual capacity reclaiming when a server S_i is scheduled for execution is given by $A_r = \{S_r | S_r \in A, d_r \leq d_{i,k}, c_r > 0\}$, where d_r is the current deadline of early completed jobs and $d_{i,k}$ is the currently assigned deadline of server S_i.

When scheduled with CSS, a server S_i starts by reclaiming the residual capacity c_r supplied by the earliest deadline active server S_r from the set of eligible servers A_r, either until the job's completion or c_r's exhaustion. S_r is then defined as $\exists^1 S_r \in A_r : min_{d_r}(A_r), A_r \neq \emptyset$.

Since the execution requirements of each job are not known beforehand, it also makes sense to devote as much excess capacity as possible to the currently executing server. As such, while there is pending work to do, remaining residual capacities are greedily consumed by the currently executing server according to a EDF policy.

We carefully considered this fairness issue. The increased computational complexity of fairly assigning residual capacities to all active servers and the fact that fairly distributing residual capacities to a large number of servers (usually in proportion to the servers' bandwidths) can originate a situation where not enough excess capacity is provided to any one to avoid a deadline miss, leading us to assign all residual capacity to the currently executing server S_i. Such a greedy capacity reclaiming not only has a reduced computational complexity, but also minimises deadline postponements and the number of preemptions and tends to be fair in the long run [LB00].

If all available residual capacities are exhausted and the current job is not complete, S_i consumes its own reserved capacity c_i, either until the job's completion or c_i's exhaustion. The hard reservation approach that was adopted enables a overloaded server S_i whose capacity has been exhausted to be kept active with its current deadline and to continue to execute its current job either by stealing capacities from inactive non-isolated servers or by using any new residual capacities that eventually will be released until $d_{i,k}$.

Let I be the set of all inactive non-isolated servers. The set of inactive non-isolated servers I_s^N eligible for capacity stealing, when the currently executing server S_i has reclaimed all the eligible residual capacity and has exhausted its own reserved capacity, is given by $I_s^N = \{S_s | S_s \in I, d_s < d_{i,k}, c_s > 0\}$, where d_s is the current deadline of each inactive non-isolated server.

With CSS, S_i is able to steal the non-isolated capacity of the earliest deadline inactive non-isolated server S_s from the set of eligible servers I_s^N, determined by $\exists^1 S_s \in I_s^N : min_{d_s}(I_s^N), I_s^N \neq \emptyset$.

Similarly to the residual capacity reclaiming phase, and due to the same reasons, non-isolated capacity stealing also follows a greedy approach. When the capacity being stolen is exhausted and the job has not yet been completed, the next non-isolated capacity c'_s is used (if any) by S_i to advance its execution. However, capacity stealing is interrupted whenever (i) the currently executing server S_i is preempted; (ii) a replenishment event occurs on the capacity c_s being stolen; or (iii) a new job arrives for the inactive non-isolated server S_s whose reserved capacity is being used by S_i. As expected, to preserve the system's schedulability, when a new job arrives for the inactive non-isolated server S_s, it reaches the active state with its remaining capacity c_s. Note that an active non-isolated server can also take advantage of available residual capacities, share its residual capacity with other servers and steal inactive non-isolated capacities.

Since the parameters of inactive servers are not automatically updated, when the currently executing server S_i tries to steal the earliest deadline inactive non-isolated capacity of server S_s it must check if an update of the current values of the deadline and reserved capacity of server S_s are needed. If the previously generated absolute deadline d_s of the selected non-isolated server S_s is lower than the current time ($d_s < t$), a new deadline ($d_s = t + T_s$) is generated and the server's capacity is recharged to its maximum value ($c_s = Q_s$). Otherwise, S_s's current values are used. In either case, S_s is kept in the inactive state.

5.3.1 The CSS scheduler

CSS differs from the original CBS scheduler in three main characteristics: (i) it follows a hard reservation approach; (ii) it greedily reclaims, as early as possible, the unused computation times originated by early completions; and (iii) it reduces isolation in a controlled fashion to donate reserved, but still unused, capacities to currently overloaded servers.

Whenever a new job $J_{i,k}$ arrives at time $a_{i,k}$ for server S_i, if S_i is active, the job is buffered and will be served later. If S_i is inactive and if $a_{i,k} < d_{i,k}$, the server becomes active and the job is served with the last generated deadline $d_{i,k}$, using the current capacity c_i. Otherwise, S_i's capacity is recharged to its maximum value $c_i = Q_i$, a new deadline is generated to $d_{i,k} = max\{a_{i,k}, d_{i,k-1}\} + T_i$, the recharging time is set to $r_i = d_{i,k}$ and its residual capacity is set to $c_r = 0$. At replenishment time r_i, unused reserved capacities are discarded.

Naturally, while a server is executing, the consumed capacity must be accounted for. By dynamically managing a pointer to the server from which the capacity is going to be decreased, the proposed dynamic accounting mechanism of CSS eliminates the need of extra queues or additional server states as used in other served-based scheduling approaches, clearly reducing its overhead. With CSS, the server from which the accounting is going to be performed is dynamically determined at the time instant when a capacity is needed.

CSS uses the following rules to manage reserved capacities:

- **Rule A:** Whenever a server S_j completes its k^{th} job and it has no pending work, its remaining reserved capacity $c_j > 0$ is immediately released as residual capacity $c_r = c_j$. c_r can now be reclaimed by eligible active servers until the currently assigned S_j's deadline $d_{j,k}$ or c_r's exhaustion. S_j is kept active with its current deadline and its reserved capacity c_j is set to zero.

- **Rule B:** The next server S_i scheduled for execution points to the earliest deadline server S_r from the set of eligible active servers with residual capacity $c_r > 0$ and deadlines $d_r \leq d_{i,k}$. S_i consumes the pointed residual capacity c_r, running with the deadline d_r of the pointed server S_r. Whenever c_r is exhausted and there is pending work, S_i disconnects from S_r and selects the next available server S'_r (if any).

- **Rule C:** If all available residual capacities are exhausted and the current k^{th} job of server S_i is not yet completed, S_i consumes its own reserved capacity c_i

either until the job's completion or c_i's exhaustion (whatever comes first). If c_i is exhausted and there is still pending work to do, S_i is kept active with its current deadline $d_{i,k}$.

- **Rule D:** With pending work and no reserved capacity left, the currently executing server S_i connects to the earliest deadline server S_s from the set of eligible inactive non-isolated server with remaining capacity $c_s > 0$ and deadlines $d_s \leq d_{i,k}$. S_i steals the pointed inactive capacity c_s, running with its current deadline $d_{i,k}$. Whenever c_s is exhausted and the job has not yet been completed, the next non-isolated capacity c'_s is used (if any).

Note that the proposed dynamic capacity accounting mechanism ensures that at time t, the currently executing server S_i is using a residual capacity c_r originated by an early completion of another active server, its own reserved capacity c_i, or is stealing capacity c_s from an inactive non-isolated server (for that order). Furthermore, in order to preserve the system's schedulability, it ensures that the longest time a server can be connected to another server is bounded by the currently pointed server's capacity and/or deadline.

With the above rules, CSS is then able to (i) achieve isolation among guaranteed tasks; (ii) efficiently reclaim unused computation times, exploiting early completions; and (iii) allow an overloaded server to steal non-isolated reserved capacities from inactive servers.

5.3.2 Handling overloads with CSS

The next example details how CSS can handle soft tasks' overloads without postponing their deadlines by greedily reclaiming residual capacities and stealing inactive non-isolated capacities used to schedule aperiodic best-effort services.

Consider the following periodic task set, described by average execution times and period: $\tau_1 = (2, 5)$, $\tau_2 = (4, 10)$, $\tau_3 = (3, 15)$. τ_1 is served by the non-isolated server S_1, while tasks τ_2 and τ_3 are served by the isolated servers S_2 and S_3, respectively, with a reserved capacity equal to their task's average execution time and period equal to their task's expected period.

A possible scheduling of this task set with CSS is detailed in Figure 5.2. When a server is connected to another server, either reclaiming a residual capacity or stealing an inactive non-isolated capacity, an arrow indicates where the capacity accounting is being performed.

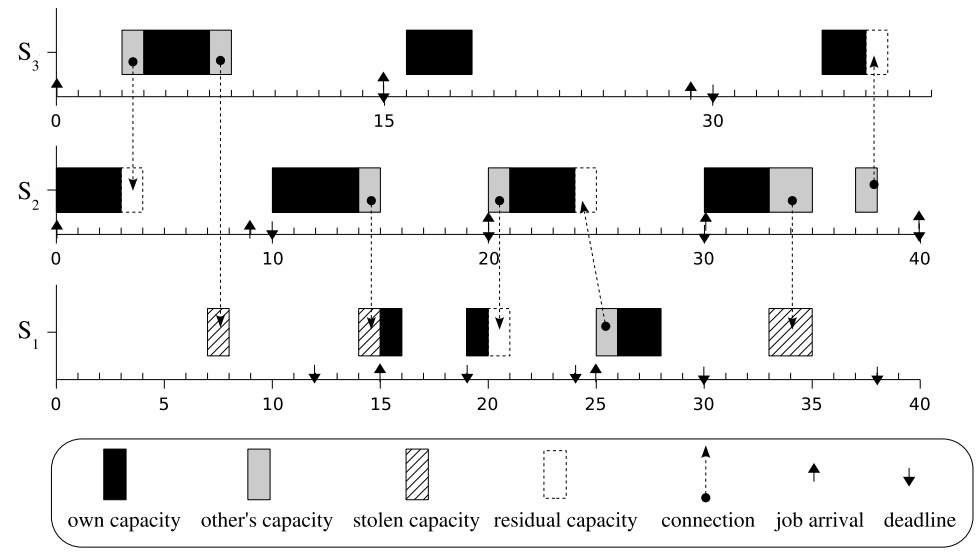

Figure 5.2: Handling overloads with CSS

At time $t = 3$, S_2 finishes the first job of task τ_2 and releases a residual capacity $c_r = 1$ with deadline $d_r = 10$ (Rule A). Server S_3 is scheduled for execution, connects to the earliest deadline residual capacity available from server S_2 and starts to execute its dedicated task τ_3, consuming the reclaimed residual capacity (Rule B). When this residual capacity is exhausted at time $t = 4$, S_2 becomes inactive and S_3 continues to execute τ_3 by using its own reserved capacity until it is exhausted at time $t = 7$ (Rule C). Since there is inactive non-isolated capacity available, S_3 handles its overload by stealing capacity previously reserved for server S_1 (Rule D). A new deadline for the stolen capacity c_s is set to time $t = 12$.

Note that at time $t = 9$ a new job of task τ_2 arrives for the inactive server S_2 but the job is only released at time $t = 10$. Recall that advancing execution times is against our purpose of executing periodic activities with stable frequencies.

At time $t = 15$, after server S_2 has completed its job by stealing some of the inactive non-isolated capacity of S_1, a new job for server S_1 arrives. At this point, S_1 becomes active, but keeping its currently available capacity and corresponding deadline.

At time $t = 16$, server S_1 exhausts its capacity and stops executing since there is no available inactive non-isolated capacity to steal. However, at time $t = 19$, a replenishment of S_1's capacity occurs and it can continue to execute the pending job. When S_1 completes its job's execution, at time $t = 20$, it releases the residual capacity $c_r = 1$ with deadline $d_r = 24$. This residual capacity is used by server S_2 before consuming its own capacity at time $t = 21$.

At time $t = 25$, a new job of task τ_1 arrives and the inactive non-isolated server S_1 becomes active. Note that it first consumes the residual capacity $c_r = 1$ with deadline $d_r = 30$, generated at time $t = 24$ by an early completion of τ_2, before consuming its own capacity.

At time $t = 33$ an overload of τ_2 is first handled by stealing capacity of the inactive non-isolated server S_1 and then, at time $t = 38$, by consuming the available residual capacity generated by an early completion of task τ_3. Recall that with CSS a server remains active, even if it has exhausted its capacity, which enables an overloaded server to take advantage of any eligible residual capacity released until its deadline.

5.4 Theoretical validation for independent tasks

This section analyses the schedulability condition for a hybrid set of independent hard and soft real-time tasks. Despite the fact that CSS is able to reduce the mean tardiness of soft real-time tasks through an efficient management of unused reserved capacities, it can only provide hard real-time guarantees if each hard real-time task is scheduled by an isolated server with a reserved capacity equal to the hard task's WCET and period equal to the task's period. When this happens, each hard task behaves like a standard hard task scheduled by EDF. The main difference is that, with CSS, hard tasks can gain access to and use extra capacities and yield their residual capacities to other tasks.

In [Abe98], it is proven that a CBS server with parameters (Q_i, T_i) cannot occupy a bandwidth greater than $\frac{Q_i}{T_i}$. That is, if $D_{S_i}(t_1, t_2)$ is the server's bandwidth demand in the interval $[t_1, t_2]$, it is shown that $\forall t_1, t_2 \in N : t_2 > t_1, D_{S_i}(t_1, t_2) \leq \frac{Q_i}{T_i}(t_2 - t_1)$.

This isolation property allow us to use a bandwidth reservation strategy to allocate a fraction of a resource to a task whose demand is not known a priori. The most important consequence of this property is that soft tasks, characterised by average values, can be scheduled together with hard tasks, even in the presence of overloads.

We state that the runtime capacity sharing and stealing performed by CSS does not affect the system's schedulability. By assigning each soft task a specific capacity, based on an average execution time estimation, the desired activation period, and isolating the effects of tasks' overloads, a hybrid task set can be guaranteed using the classical Liu and Layland condition [LL73].

Before proving such schedulability test, we start by ensuring that all generated capacities are exhausted before their respective deadlines.

Lemma 5.4.0.1 *Given a set S of isolated servers, each isolated capacity generated during scheduling is either consumed or discharged until its deadline.*

Proof

Let $a_{i,k}$ denote the instant at which a new job $J_{i,k}$ arrives and the isolated associated server S_i is inactive. At $a_{i,k}$, a new capacity $c_i = Q_i$ is generated and S_i is inserted into the ready queue.

Let $\forall_{i,k}\ d_{i,k} = max\{a_{i,k}, d_{i,k-1}\} + T_i$ be the deadline and $\forall_{i,k}\ r_i = d_{i,k}$ the replenishment time associated with capacity c_i.

Let $[t, t + \Delta t[$ denote a time interval during which server S_i is executing, consuming its own capacity c_i. Consequently, S_i has used an amount equal to $c_i' = c_i - \Delta t \geq 0$ of its own capacity during this period. As such, c_i must be decreased to c_i', until it is exhausted.

Let $f_{i,k}$ denote the time instant at which server S_i completes its job $J_{i,k}$. Assume that there are no pending jobs for server S_i at time $f_{i,k}$. According to Rule A, the available residual capacity $c_r = c_i > 0$ can immediately be reclaimed by other servers.

Let $c_r = c_i$ be the residual capacity available to other servers. At instant $f_{i,k}$, the server's capacity c_i is set to zero and another active server S_j is scheduled for execution. According to Rule B, if the inequality $d_{i,k} \leq d_{j,l}$ holds, let $[t, t + \Delta t[$ denote the time interval during which server S_j is executing, consuming the residual capacity c_r of server S_i. Consequently, S_j has used an amount equal to $c_r' = c_r - \Delta t \geq 0$ during this period. As such, c_r must be decreased to c_r', until the capacity c_r is exhausted or the currently assigned deadline $d_{i,k}$ of server S_i is reached.

At replenishment time $t = r_i$, any remaining residual capacity c_r of server S_i not used by another active server is discharged.

□

Lemma 5.4.0.2 *Given a set S of isolated and non-isolated servers, each non-isolated capacity generated during scheduling is either consumed or discharged until its deadline.*

Proof

To prove this lemma we analyse the following cases: a) a new non-isolated capacity is generated whenever an overloaded active server needs to steal the inactive non-isolated server's capacity; and b) a non-isolated capacity is generated whenever a new job arrives for the inactive non-isolated server.

Case a.

Let $a_{j,k}$ denote the time instant when an active overloaded server S_j starts to consume the non-isolated capacity c_i of the inactive non-isolated server S_i.

If the inequality $d_{i,k-1} \leq a_{j,k}$ holds, a new deadline $d_{i,k} = a_{j,k} + T_i$ is generated for the non-isolated capacity, the server's capacity c_i is recharged to its maximum value $c_i = Q_i$ and the replenishment time r_i is set to $r_i = d_{i,k}$. Otherwise, server S_i keeps its current values of c_i, $d_{i,k}$, and r_i.

Let $[t, t + \Delta t[$ denote the time interval during which server S_j is executing, stealing the non-isolated capacity c_i of server S_i. Consequently, S_j has used an amount equal to $c'_i = c_i - \Delta t \geq 0$ of the non-isolated capacity c_i. As such, c_i must be decreased to c'_i, until its value is exhausted. If a new job arrives at $a_{i,k} < a'_{i,k} < r_i$, the inactive non-isolated server S_i becomes active, using its current values for c_i, $d_{i,k}$, and r_i. If, at time $a'_{i,k}$ the capacity c_i was being stolen by an active overloaded server, capacity stealing is immediately interrupted.

While active, the behaviour of the non-isolated server S_i is equal to any other active isolated server in the system. As such, the accounting for the remaining capacity c_i is proven by Lemma 5.4.0.1.

Case b.

Let $a_{i,k}$ denote the time instant when a new job $J_{i,k}$ arrives for the inactive non-isolated server S_i.

If the inequality $d_{i,k-1} \leq a_{i,k}$ holds, a new deadline $d_{i,k} = a_{i,k} + T_i$ is generated, the server's capacity c_i is recharged to its maximum value $c_i = Q_i$ and the replenishment time r_i is set to $r_i = d_{i,k}$. Otherwise, server S_i keeps its current values for c_i, $d_{i,k}$, and r_i.

At time $a_{i,k}$ the non-isolated server S_i becomes active and it is inserted into the ready queue. As such, its capacity c_i is consumed as follows from Lemma 5.4.0.1.

□

Theorem 5.4.0.1 *Let τ_h be a set of n periodic hard real-time tasks, with each task τ_i being scheduled by a dedicated isolated server S_i with a reserved capacity Q_i equal to the task's WCET and T_i equal to the task's period. Let τ_s be a set of soft real-time tasks scheduled by a group of isolated and non-isolated severs with total utilisation U_{soft}. Then τ_h is feasible if and only if*

$$+U_{soft} + \sum_{\tau_i \in \tau_h}^{n} \frac{Q_i}{T_i} \leq 1$$

Proof

The theorem follows immediately from Lemma 5.4.0.1 and Lemma 5.4.0.2. In fact, Lemma 5.4.0.1 ensures that each generated isolated capacity is always exhausted or discharged until its deadline. The same is true for any generated non-isolated capacity, according to Lemma 5.4.0.2.

Since the worst case response time of a hard task is independent of whether the reserved capacity of some server is being used by that server to execute its dedicated task or it is being consumed by any other server in the system, the system's schedulability is independent of whether the proposed dynamic capacity accounting mechanism of CSS is in operation or not. In the worst case, the longest time a server can be connected to another server is bounded by the currently pointed server's capacity and deadline.

□

5.5 Summary

This chapter was focused on the scheduling of tasks with hard, soft, and non real-time constraints in open real-time systems. While several solutions have already been proposed to maximise resource usage and achieve a guaranteed service for hard tasks and inter-task isolation using average execution estimations for soft tasks, unused reserved capacities can be more efficiently used to meet deadlines of soft tasks whose resource usage exceeds their reservations.

The proposed Capacity Sharing and Stealing (CSS) scheduler considers the coexistence of the traditional *isolated* servers with a novel *non-isolated* type of servers, combining an efficient reclamation of residual capacities with a controlled isolation loss, making additional capacity available from two sources: (i) by reclaiming unused allocated capacity when jobs complete in less than their budgeted execution time; and (ii) by stealing allocated capacities to inactive non-isolated servers used to schedule aperiodic best-effort jobs.

In addition, a schedulability analysis for a hybrid set of independent hard and soft real-time tasks has been presented. The analysis is based on the most important formal properties of CSS, presented and proved in this chapter.

Chapter 6

Handling QoS inter-dependencies

Although a lot of research has been conducted on the support of services' dynamic QoS reconfiguration under different constraints of resource availability, most has been focused on the runtime management of independent QoS attributes.

This chapter extends the CooperatES framework to support adaptive cooperative coalitions of nodes able to autonomously organise, regulate and optimise themselves without the user's intervention or any other central entity, even when services exhibit unrestricted local and distributed QoS inter-dependencies among their tasks.

6.1 Introduction

While runtime adaptation is widely recognised as valuable, adaptations in most existing QoS-aware systems are limited to changing independent execution parameters. We believe that the true potential of existing adaptation techniques can only be achieved if support is provided for more general solutions, including inter-dependent runtime adaptations that span multiple hosts and multiple components.

Increasingly, distributed applications consist of interacting components that may exhibit unrestricted QoS inter-dependencies among them. Such relations specify that a component offers a certain level of QoS under the condition that some specified QoS will be offered by the environment or by other service's components. For example, network bandwidth can be traded for processing power by using data compression techniques. However, some compression techniques may not be lossless, thus, they

may have an impact on the quality of the data content.

As such, when the outputted QoS of some task depends not only on the amount and type of used resources but also on the quality of the received inputs sent by other tasks, the QoS adaptation process has to ensure that a source task provides a QoS which is acceptable to all consumer tasks and lies within the QoS range supported by the source task. A service's feasible QoS level must then be defined as the set of compatible QoS regions provided by all the dependent components that compose the service.

This means that QoS-aware systems must handle more and more complexity on their own. The challenge here is to find mechanisms that control the dynamics of these complex, distributed, interconnected and rapidly changing environments within a useful and bounded time. This chapter elaborates on these issues and extends the CooperatES framework to support adaptive cooperative coalitions of nodes able to autonomously organise, regulate and optimise themselves without the intervention of a user or any other central entity, even when services exhibit unrestricted QoS inter-dependencies.

Solving this dependency problem among local QoS inter-dependencies is equivalent to finding an assignment of values to all QoS attributes such that all dependency constraints are locally satisfied. Sections 6.2.1 and 6.2.2 reformulate, respectively, the anytime algorithms for service proposal formulation and dynamic QoS reconfiguration proposed in Chapter 4 to handle the local execution of services' components whose behaviour and input/output qualities are inter-dependent. To guarantee that a valid solution is available at any time, QoS dependencies are tracked and the performed changes are propagated to all the affected attributes at each iteration of the algorithms. To the best of our knowledge no other works propose an anytime approach for a distributed QoS configuration of resource intensive services in open real-time embedded services with the ability to handle tasks' inter-dependencies while maximising the satisfaction of each user's quality preferences.

However, a distributed system composed by self-managing nodes which optimise their behaviour towards some individual goals is not necessarily optimised at a global level as there is the possibility that conflicting greedy decisions may lead to interference between the different nodes' self-management behaviour, conflicts over shared resources, sub-optimal system performance and hysteresis effects [FDC02].

Section 6.3 discusses these issues and proposes an one-step decentralised coordination model based on an effective feedback mechanism to reduce the complexity of the needed interactions among nodes until a collective adaptation behaviour is determined.

Positive feedback is used to reinforce the selection of the new desired global service solution, while negative feedback discourages nodes to act in a greedy fashion as this adversely impacts on the provided service levels at neighbouring nodes.

By exchanging feedback on the desired self-adaptive actions, nodes converge towards a global solution, even if that means not supplying their individually best solutions. As a result, each node, although autonomous, is influenced by, and can influence, the behaviour of other nodes in the system.

This work has been partially published in [NP08b, NP08a, NP09a].

6.2 Local QoS inter-dependencies

As discussed in the previous chapters, runtime adaptation is a fundamental issue in resource-constrained QoS-aware systems since it determines how well users' service requests are satisfied in the presence of dynamically changing operating conditions. However, although a lot of research has been conducted on the support of services' dynamic QoS reconfiguration under different constraints of resource availability, there are still a number of challenges to be addressed.

One of those challenges is brought by the existence of QoS inter-dependencies among services' tasks. A QoS dimension Q_a is said to be dependent on another dimension Q_b if a change along the dimension Q_b will increase the needed resource demand to achieve the quality level previously achieved along Q_a [RLLS97]. As such, QoS dependencies explicitly express dependency relationships existing over a service's QoS characteristics while specifying the strength and direction of the link.

Formally, there are n QoS attributes $x_1, x_2, \ldots, x_n$ of a service S, whose values are taken from the domains $D_1, D_2, \ldots, D_n$, respectively, and a set of dependency constraints on their values. The constraint $p_k(x_{k1}, \ldots, x_{kj})$ is a predicate that is defined on the Cartesian product $D_{k1} * \ldots * D_{kj}$. This predicate is true if and only if the value assignment of these variables satisfies this constraint. Note that there is no restriction on the form of the predicate. It can be a mathematical or logical formula or any arbitrary relation defined by a tuple of acceptable values.

This means that the runtime QoS adaptation process must ensure that a source task provides a QoS which is acceptable to all consumer tasks and lies within the QoS range supported by the source task. As such, the system may have to adapt the QoS of individual tasks according to some inter-task QoS dependencies when searching for the best overall feasible solution. It seems clear that the increased complexity of such

optimisation makes it even more beneficial to use an anytime approach that can trade the achieved solution's quality by its computational cost whenever a timely answer to events is desired.

In the presence of dependency relationships among the tasks of a service S, we assume that a resource intensive service is partitioned in a set of work units $\{w_1, \ldots, w_n\}$ that can be executed at varying levels of QoS, whenever the imposed set of QoS constraints cannot be satisfyingly answered by a single node. Each work unit $w_i = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is a set of one or more tasks that must be executed in the same node due to local QoS dependencies.

We represent the set of inter-dependencies among tasks of a work unit $w_i \in S$ as a dependency graph $\mathcal{G}_{w_i} = (\mathcal{V}_{w_i}, \mathcal{E}_{w_i})$, where each vertex $v_i \in \mathcal{V}_{\sqsupseteq\rangle}$ represents a task τ_i and a directed edge $e_i \in \mathcal{E}_{\sqsupseteq\rangle}$ from τ_j to τ_k indicates that τ_k is functionally dependent on τ_j.

As such, whenever a service S can be divided in a set of independent work units $\{w_1, \ldots, w_n\}$, the runtime adaptation problem of each work unit w_i can be locally handled as a resource selection problem at each individual node by traversing the corresponding dependency graph $\mathcal{G}_{w_i}$ whenever a change in the current value of one QoS parameter of a task $\tau_j \in w_i$ has an impact on other tasks $\tau_k \in w_i$. For the sake of simplicity, we present the following functions in a declarative notation with the same operational model as a pattern matching-based functional language.

Given a graph $\mathcal{G}_{w_i} = (\mathcal{V}_{w_i}, \mathcal{E}_{w_i})$ and given two tasks $\tau_i, \tau_j \in \mathcal{V}_{w_i}$, we obtain all the tasks in the possible paths between τ_i and τ_j as the result of the function[1]:

$$
\begin{array}{lcl}
paths(\tau_i, \tau_j) & = & flatten(paths(\tau_i, \tau_j, \emptyset)) \\
 & & \\
paths(\tau_i, \tau_j, T) & = & \emptyset, \text{ if } \tau_i = \tau_j \\
paths(\tau_i, \tau_j, T) & = & \{\{\tau_i, \tau_{k_1}\} \cup paths(\tau_{k_1}, \tau_j, T \cup \{\tau_{k_1}\}), \\
 & & \vdots \\
 & & \{\tau_i, \tau_{k_n}\} \cup paths(\tau_{k_n}, \tau_j, T \cup \{\tau_{k_n}\})\}, \\
 & & \forall \tau_{k_m} \in \mathcal{V}_{w_i}, \text{ such that } (\tau_i, \tau_{k_m}) \in \mathcal{E}_{w_i} \text{ and } \tau_{k_m} \notin T \\
paths(\tau_i, \tau_j, T) & = & \bot
\end{array}
$$

Given a set A containing other sets, the function *flatten(A)* is defined as:

[1]The function is generic enough to be used later in this chapter to obtain all the nodes between dependent groups of tasks in a distributed system

$$\begin{aligned} flatten(\emptyset) &= \emptyset \\ flatten(A) &= a \cup flatten(A \setminus a), \text{ if } a \in A \end{aligned}$$

Proposition 6.2.0.1 *Given a connected graph $\mathcal{G}_{w_i} = (\mathcal{V}_{w_i}, \mathcal{E}_{w_i})$ and two tasks $\tau_i, \tau_j \in \mathcal{V}_{w_i}$, $paths(\tau_i, \tau_j)$ terminates and returns all the tasks in the possible paths between τ_i and τ_j, $\emptyset$ in case $\tau_i = \tau_j$, or $\perp$ in case there is no path between $\tau_i, \tau_j \in \mathcal{V}_{w_i}$.*

Note that the *paths* function is a breadth first approach with cycle checking to find nodes in the possible paths in graphs. It outputs all the tasks in the possible paths between two tasks τ_i and τ_j, or returns $\perp$ if there is no path between those two tasks. Nevertheless, for the sake of clarity of presentation, in the remainder of this chapter, we assume that only well-formed dependency graphs are considered in the proposed algorithms.

Having a work unit w_i, its dependency graph $\mathcal{G}_{w_i}$, and a way to traverse it, both the anytime service proposal algorithm and the anytime dynamic QoS reconfiguration proposed in Chapter 4 can be easily extended to handle the local execution of services' tasks whose behaviour and input/output qualities are interdependent. Sections 6.2.1 and 6.2.2 detail such extension.

6.2.1 Anytime service proposal formulation for inter-dependent task sets

Based on the dependency graph $\mathcal{G}_{w_i}$ associated with a work unit $w_i \in S$, the new version of the proposed anytime service proposal formulation algorithm tracks QoS dependencies and propagates the performed changes in one attribute to all locally affected attributes. If, by traversing the path of dependencies, the algorithm finds a task that is already in its list of resolved dependencies, a deadlock is detected and the service proposal formulation is aborted.

Note that the search for a feasible solution to accommodate the new work unit w_i of inter-dependent tasks uses the same deterministic heuristics for the selection of the attribute to upgrade (Step 1, detailed in Algorithm 8) or downgrade (Step 2, detailed in Algorithm 9) and the same quality measures as the previous version of the algorithm for independent task sets.

Furthermore, to guarantee that a valid solution is available if the algorithm is interrupted at any time, performed changes on dependent attributes are propagated to

Algorithm 8 Anytime service proposal formulation for inter-dependent task sets - Step 1

Let τ^e be the set of previously accepted tasks whose stability period Δ_t has expired.
Let τ^p be the set of all previously accepted tasks whose current QoS cannot be changed.
Let w_i be the newly arrived work unit of service S and $\mathcal{G}_{w_i}$ its graph of dependencies.
Let $\mathcal{L}$ be the set of local dependency graphs $\mathcal{G}_{w_x}$ of all work units $w_x \subseteq \tau^e \cup \tau^p \cup w_i$.
Each task $\tau_i \in \{\tau^e \cup \tau^p \cup w_i\}$ has associated a set of user's defined QoS constraints Q^i.
Each Q^i_{kj} is a finite set of n quality choices for the j^{th} attribute, expressed in decreasing preference order, for all k QoS dimensions.
Let σ be the determined set of SLAs, updated at each step of the algorithm.

Step 1 - Maximise the QoS level of each task $\tau_a \in w_{ij}$

1: Define $SLA_{w_{ij}}$ by selecting the lowest requested QoS level $Q^a_{kj}[n]$, for all the j attributes of the k QoS dimensions, for each newly arrived task $\tau^a \in w_i$, considering the QoS dependencies of $\mathcal{G}_{w_i}$
2: Keep the current QoS level for each task $\tau_k \in \tau^e$
3: Update the current set of SLAs σ ▷ $\sigma \leftarrow \sigma \cup SLA_{w_i}$
4: **while** $feasibility(\sigma) =$ **TRUE do**
5: **for** each task $\tau_a \in w_i$ **do**
6: **for** each j^{th} attribute of any k QoS dimension in τ_a with value $Q^a_{kj}[m] > Q^a_{kj}[0]$ **do**
7: Upgrade attribute j to the next possible value $Q^a_{kj}[m-1]$
8: Traverse $\mathcal{G}_{w_i}$ and change values accordingly
9: Determine the utility increase of this upgrade
10: **end for**
11: **end for**
12: Find task τ_{max} whose reward increase is maximum
13: Define $SLA'_{\tau_{max}}$ for task τ_{max} with the new value $Q^{max}_{yx}[m-1]$ for attribute x of the QoS dimension y
14: Update the current set of promised SLAs σ ▷ $\sigma \leftarrow \sigma \setminus SLA_{\tau_{max}} \cup SLA'_{\tau_{max}}$
15: **end while**

all affected tasks at each iteration of the algorithm by traversing the correspondent dependency graph. As such, the anytime behaviour and the conformity with the desirable properties of anytime algorithms discussed in Chapter 4 still hold on this

Algorithm 9 Anytime service proposal formulation for inter-dependent task sets - Step 2

Step 2 - Find the local minimal service degradation to accommodate each $\tau_a \in w_i$

16: **while** $feasibility(\sigma) \neq$ **TRUE do**
17: **for** each work unit $w_d \subseteq \tau^e \cup w_i$ **do**
18: **for** each task $\tau_i \in w_d$ **do**
19: **for** each j^{th} attribute of any k QoS dimension in τ_a with value $Q^i_{kj}[m] > Q^i_{kj}[n]$ **do**
20: Downgrade attribute j to the previous possible value $Q^i_{kj}[m+1]$
21: Traverse $\mathcal{G}_{w_d} \in \mathcal{L}$ and change values accordingly
22: Determine the utility decrease of this downgrade
23: **end for**
24: **end for**
25: **end for**
26: Find task τ_{min} whose reward decrease is minimum
27: Define $SLA'_{w_{min}}$ for the work unit where task τ_{min} belongs, setting the QoS values of all affected tasks according with the new value $Q^{min}_{yx}[m+1]$ for attribute x of the QoS dimension y for task τ_{min}
28: Update the current set of promised SLAs σ $\quad \triangleright\ \sigma \leftarrow \sigma \setminus SLA_{w_{min}} \cup SLA'_{w_{min}}$
29: **end while**
30: **return** new local set of promised SLAs σ

new version of the algorithm.

6.2.2 Anytime re-upgrade of previously downgraded levels of service for inter-dependent task sets

Recall that our basic viewpoint is that service stability can be more important for some users than some momentary maximum quality that does not take into consideration the services' QoS reconfiguration rate. Although the framework ensures a fixed quality level during a dynamically determined period of time Δ_t for each accepted service S, an upgrade of the current quality level should be done according to each user's stability preferences.

Possible QoS upgrades of previously downgraded SLAs for inter-dependent task sets

are determined by Algorithm 10, which are then compared against the users' stability constraints. The proposed anytime QoS reconfiguration algorithm tries to restore the initially provided SLAs by selecting, at each iteration, the new configuration that achieves the greatest reward increase, handling tasks whose execution behaviour and input/output qualities are inter-dependent.

Algorithm 10 Determine possible QoS upgrades for inter-dependent task sets

Let τ^e be the set of previously accepted tasks whose stability period Δ_t has expired.
Let τ^p be the set of all previously accepted tasks whose current QoS cannot be changed.
Each task $\tau_i \in \tau^e \cup \tau^p$ has associated a set of user's defined QoS constraints Q^i.
Let $Q^i_{kj}[init]$ be the initially provided and $Q^i_{kj}[current]$ the currently provided level of service for attribute j of the k_{th} QoS dimension for each task $\tau_i \in \tau^e$.
Let $\mathcal{L}$ be the set of local dependency graphs $\mathcal{G}_{\sqsupseteq\rangle}$ of all work units $w_i \subseteq \tau^e \cup \tau^p$.
Let σ be the determined set of SLAs, updated at each step of the algorithm.

1: **while** $feasibility(\sigma) =$ **TRUE do**
2: **for** each work unit $w_u \in \tau^e$ **do**
3: **for** each task $\tau_i \in w_u$ **do**
4: **for** each j^{th} attribute of any k QoS dimension with value $Q^i_{kj}[current] > Q^i_{kj}[init]$ **do**
5: Upgrade attribute j to the next possible value $Q^i_{kj}[m-1]$
6: Traverse $\mathcal{G}_{w_u} \in \mathcal{L}$ and change values accordingly
7: Determine the utility increase of this upgrade
8: **end for**
9: **end for**
10: **end for**
11: Find task τ_{max} whose reward increase is maximum
12: Define $SLA'_{w_{max}}$ for the work unit where task τ_{max} belongs, setting the QoS values of all affected tasks according with the new value $Q^{max}_{yx}[m-1]$ for attribute x of the QoS dimension y for task τ_{max}
13: Update the current set of promised SLAs σ $\quad \triangleright\ \sigma \leftarrow \sigma \setminus SLA_{w_{max}} \cup SLA'_{w_{max}}$
14: **end while**
15: **return** new local set of promised SLAs σ

Similarly to the anytime service proposal formulation algorithm, note that the search for a feasible set of upgraded inter-dependent SLAs uses the same deterministic heuristics for the selection of the QoS attribute to upgrade and the same quality measures as

its previous version for independent task sets. As such, the anytime behaviour and the conformity with the desirable properties of anytime algorithms discussed in Chapter 4 still hold on this new version of the algorithm.

6.3 Distributed QoS inter-dependencies

One of the key ideas of the CooperatES framework is that each coalition member should be able to take the initiative and to decide when and how to adapt to changes in the environment. However, whenever such autonomous adaptations have an impact on the outputted QoS of other coalition members the need of coordination arises: how to ensure that local, individual, adaptation actions of a node can produce a globally acceptable solution for the entire distributed service [GC92].

Yet, coordinating autonomous dependent adaptations in an open and dynamic system is challenging and may require complex communication and synchronisation strategies. According to [MC94] it borrows from as diverse areas as computer science, organisation theory, operations research, economics, linguistics, and psychology. This great diversity makes it very difficult to discuss every potential work that has some remote relation to coordination.

Nevertheless, researchers have been proposing both centralised and decentralised coordination models to describe the "glue" that connects adaptive dependent computational activities. A distributed system built using a centralised coordination model is one where the behaviour of the nodes in the system is controlled either by an active manager node or by a pre-determined plan followed by the nodes [GK04]. However, with the increasing size and complexity of open embedded systems the ability to build self-managed distributed systems using centralised coordination models is reaching its limits [MMB03], as solutions they produce require too much global knowledge.

As such, researchers are increasingly investigating decentralised coordination models to establish and maintain system properties [DC99, GAKT03, DWH03, BDK+03, DH08]. A system built using a decentralised coordination model is a self-organising system whose system-wide behaviour is established and maintained solely by the interactions of its nodes that execute using only a partial view of the system [GK04]. How these nodes were to interact in order to solve the problem (and not what they were actually doing) became the focus of decentralised coordination research. Without a central coordination entity, the collective adaptation behaviour must emerge from local interactions among nodes. This is typically accomplished through the exchange

of multiple messages to ensure that all involved nodes make the same decision about whether and how to adapt.

Bridges et al. [BCHS01] propose a framework based on Cactus [TUoA] which supports adaptations that span multiple components and multiple nodes in a distributed system. The architecture supports coordinated adaptations across layers using a fuzzy logic based controller module and coordination across hosts using a prototype protocol designed for communication oriented services. However, the authors do not specifically propose a method to obtain a globally coordinated solution.

Ren et al. [RST06] present a real-time reconfigurable coordination model (RT-RCC) that decomposes dynamic real-time information systems based on the principle of separation of concerns, namely, functional actors which are responsible for accomplishing tasks, and non-functional coordinators which are responsible for coordination among the functional actors. High level language abstractions and a framework for actors and coordinators are provided to facilitate programming with the RT-RCC model.

A similar approach is followed by Kwiat et al. [KR06] who propose a coordination model for improving software system attack-tolerance and survivability in open hostile environments. The coordination model is based on three active entities: actors, roles, and coordinators. Actors abstract the system's functionalities, while roles and coordinators statically encapsulate coordination constraints and dynamically propagate those constraints among themselves and onto the actors. Both the coordination constraints and coordination activities are distributed among the coordinators and roles, shielding the system from single points of failure.

However, none of these works proposes support for coordinating inter-dependent autonomous QoS adaptations in cooperative systems, which is the focus of our work. Furthermore, with some decentralised coordination models it becomes difficult to predict the exact behaviour of the system taken as a whole because of the large number of possible non-deterministic ways in which the system can behave [Ser06].

Whenever real-time decision making is in order, a timely answer to events suggests that after some finite and bounded time we would expect the global adaptation process to converge to a consistent solution. Furthermore, optimal decentralised control is known to be computationally intractable [DWH03], although near-optimal systems can be developed for certain classes of applications [JMB04, DC07, DH08].

Our goal is then to achieve a fast convergence to a global solution through a regulated decentralised coordination without overflowing nodes with messages. The next section details the proposed one-step decentralised coordination model based on an effective

feedback mechanism to reduce the complexity of the needed interactions among nodes until a collective adaptation behaviour is determined.

6.3.1 A one-step decentralised coordination model

The core idea behind the proposed decentralised coordination model is to support distributed systems composed of autonomous individual nodes working without any central control but still producing the desired function as a whole.

Let $W = \{w_1, \ldots, w_n\}$ be the finite set of work units of a service S. Then, the coalition formation phase defines a connected graph $\mathcal{G}_\mathcal{W} = (\mathcal{V}_W, \mathcal{E}_W)$ for dependencies among work units $w_i \in W$, on top of the service's distribution graph, where each vertex $v_i \in \mathcal{V}_\mathcal{W}$ represents a work unit w_i and a directed edge $e_i \in \mathcal{E}_\mathcal{W}$ from w_j to w_k indicates that w_k is functionally dependent on w_j. Within $\mathcal{G}_\mathcal{W} = (\mathcal{V}_W, \mathcal{E}_W)$, we call *cut vertex* to a node $n_i \in \mathcal{V}_\mathcal{W}$, if the removal of that node divides $\mathcal{G}_W$ in two separate connected graphs.

We assume that each work unit $w_i \in W$ is being executed at its current QoS level Q^i_{val} at a node n_i, from a set of predefined QoS levels $\{Q_0, \ldots, Q_n\}$, ranging from the user's desired QoS level $L_{desired}$ to the maximum tolerable service degradation, specified by the minimum acceptable QoS level $L_{minimum}$. This relation is represented by a triple (n_i, w_i, Q^i_{val}). Furthermore, for a given work unit $w_i \in W$, each node n_i knows the set $I_{w_i} = \{(n_j, w_j, Q^j_{val}), \ldots, (n_k, w_k, Q^k_{val})\}$, describing the quality of all the inputs related to work unit w_i coming from its adjacent nodes in $\mathcal{G}_\mathcal{W}$.

Upon such system model, we propose to model a self-managed coalition as a group of nodes that respond to environmental inter-dependent changes according to a distributed coordination protocol defined by the following phases:

1. **Coordination request.** Whenever $Q^i_{val'}$, the needed downgrade or desired upgrade of the currently outputted QoS Q^i_{val} for a work unit $w_i \in S$, has an impact on the currently supplied QoS level of other work units $w_j \in S$ being executed at other coalition members, a coordination request is sent to the affected partners.

2. **Local optimisation.** Affected partners become aware of the new output values $Q^i_{val'}$ of w_i and recompute their local set of SLAs in order to formulate the corresponding feedback on the requested adaptation action. We assume that coalition partners are willing to collaborate in order to achieve a global coalition's consistency, even if this might reduce the utility of their local optimisations.

However, a node only agrees with the requested adaptive action if and only if its new local set of SLAs is feasible.

3. **Adaptive action.** If the requested adaptive action is accepted by all the affected nodes in the coalition, the new local set of SLAs is imposed at each of those coalition members. Otherwise, the currently global QoS level of service S remains unchanged.

As such, requests for coordination may dynamically arrive at any time, at any node n_j. We consider the existence of feasible QoS regions [SdML99]. A region of output quality $[q(o)_1, q(o)_2]$ is defined as the QoS level that can be provided by a work unit when provided with sufficient input quality and resources. Within a QoS region, it may be possible to keep the current output quality by compensating for a decrease in input quality by an increase in the amount of used resources or vice versa.

As such, if a node n_j, despite the change in current quality of some or all of its inputs, is able to continue to produce its current QoS level there is no need to further propagate the required coordination request along the dependency graph $\mathcal{G}_{\mathcal{W}}$. Thus, a *cut-vertex* is a key node in our approach.

Consider that a node n_k proposes an upgrade to Q'_{val} for a work unit $w_k \in S$. It may happen that some other nodes, precedent in the path until the next cut-vertex n_c, may be able to upgrade to Q'_{val} and others may not. Whenever the cut-vertex n_c receives the upgrade request and its new set of inputs, if it is unable to upgrade to Q'_{val} then all the effort of previous nodes to upgrade is unnecessary and the global update fails. Otherwise, the upgrade coordination request continues along the graph, until the end-user node n_u is reached.

In the case of a downgrade to Q'_{val} initiated by node n_k, it may happen that some other nodes in the path to the next cut-vertex n_c may be able to continue to output their current QoS level despite the downgraded input and others may not. Again, if the cut-vertex is unable to keep outputting its current QoS level then all the precedent nodes which are compensating their downgraded inputs with an increased resource usage can downgrade to Q'_{val} since their effort is useless.

In these and all other cases, the formulation of the corresponding positive or negative feedback, at the *local optimisation* phase, must depend on the feasibility of the requested coordination action as a function of the quality of the node's new inputs I_{w_i} for the locally executed work unit w_i. Such feasibility is determined by the anytime local QoS optimisation algorithm detailed in Algorithm 11 which aims to minimise the impact of the requested changes on the currently provided QoS level of other services.

Algorithm 11 Feedback formulation

Let τ^e be the set of previously accepted tasks whose stability period Δ_t has expired.
Let τ^p be the set of all previously accepted tasks whose current QoS cannot be changed.
Each task $\tau_i \in \tau^e \cup \tau^p$ has associated a set of user's defined QoS constraints Q_i.
Let $Q_{kj}[i]$ be the currently provided level of service for attribute j of the k_{th} QoS dimension for each task $\tau_i \in \tau^e$
Let $\mathcal{L}$ be the set of local dependency graphs $\mathcal{G}_{w_i}$ of all work units $w_i \in \tau^e \cup \tau^p$
Let σ be the determined set of SLAs, updated at each step of the algorithm

1: Define SLA'_{wi}, the requested SLA for the work unit w_i, as a function on the new input values I_{w_i} and the required output level $Q_{val'}$
2: Update the current set of SLAs σ ▷ $\sigma \leftarrow \sigma \setminus SLA_{w_i} \cup SLA'_{w_i}$
3: **while** $feasibility(\sigma) \neq$ **TRUE do**
4: **if** there is no task τ_i being served at $Q_{kj}[m] > Q_{kj}[n]$, for any j attribute of any k QoS dimension **then**
5: **return FALSE**
6: **end if**
7: **for** each work unit $w_d \subseteq \tau^e \setminus w_i$ **do**
8: **for** each task $\tau_i \in w_d$ **do**
9: **for** each j^{th} attribute of any k QoS dimension in τ_a with value $Q_{kj}[m] > Q_{kj}[n]$ **do**
10: Downgrade attribute j to the previous possible value $Q_{kj}[m+1]$
11: Traverse $\mathcal{G}_{w_d} \in \mathcal{L}$ and change values accordingly
12: Determine the utility decrease of this downgrade
13: **end for**
14: **end for**
15: **end for**
16: Find task τ_{min} whose reward decrease is minimum
17: Define $SLA'_{w_{min}}$ for the work unit where task τ_{min} belongs, setting the QoS values of all affected tasks according with the new value $Q_{yx}[m+1]$ for attribute x of the QoS dimension y for task τ_{min}
18: Update the current set of promised SLAs σ ▷ $\sigma \leftarrow \sigma \setminus SLA_{w_{min}} \cup SLA'_{w_{min}}$
19: **end while**
20: **return TRUE**

Note that the node's local reward associated with the new local set of SLAs can be lower than the node's local reward prior to the coordination request. Recall that we make the assumption that, in cooperative environments, coalition partners are willing to collaborate in order to achieve a global coalition consistency, even if this coordination might reduce the global utility of their local QoS optimisations.

If all the nodes affected by the requested adaptation sent by node n_i agree with its new service solution, the *adaptive action* phase takes place. A "commit" message is sent by node n_i to its direct neighbours in the dependency graph, which then propagate the message to all the involved nodes in the global adaptation process.

Decentralised control is then a self-organising emergent property of the system. The proposed coordination model is based on these two basic modes of interaction: positive and negative feedback. Negative feedback loops occur when a change in one coalition member triggers an opposing response that counteracts that change at other dependent node. On the other hand, positive feedback loops promote global adaptations. The snowballing effect of positive feedback takes an initial change in one node and reinforces that change in the same direction at all the affected partners. By exchanging feedback on the performed self-adaptations, nodes converge towards a global solution, overcoming the lack of a central coordination and global knowledge.

Note that only one negotiation round is required between any pair of dependent nodes. As such, the uncertain outcome of iterative decentralised control models whose effect may not be observable until some unknowable time in the future is not present in the proposed regulated coordination model.

Also note that the normal operation of nodes continues in parallel with the *change acknowledge* and *local optimisation* phases. Every time a node recomputes its set of local SLAs, promised resources are pre-reserved until the global negotiation's outcome is known (or a timeout expires). As such, the currently provided QoS levels only actually changes at the *adaptive action* phase, as a result of a successful global coordination.

Due to the environment's dynamism, more than one coalition member can start an adaptation process that spans multiple nodes at a given time. Such request can either be a downgrade or an upgrade of its current SLA for a work unit w_i of service S. Even with multiple simultaneous negotiations for the same service S, only one of those will result in a successful adaptation at several nodes since, due to local resource limitations, only the minimum globally requested SLA will be accepted by all the negotiation participants. In order to manage these simultaneous negotiations, every negotiation has an unique identifier, generated by the requesting node.

6.3.2 Properties of the proposed decentralised coordination model

In this section we provide a global view of what is involved for the general case and analyse some of the properties of the proposed decentralised coordination model. We start with some auxiliary definitions and proofs.

Given a node n_i, a work unit w_i , the set of local SLAs $\sigma = \{SLA_{w_0}, \ldots, SLA_{w_p}\}$ for the p locally executed work units, $Q^i_{val'}$ as the new requested QoS level for w_i, and $I_{w_i} = \{(n_j, w_j, Q^j_{val}), \ldots, (n_k, w_k, Q^k_{val})\}$ as the set of QoS levels given as input to w_i, then the value of $test\_feasibility(n_i, w_i, Q^i_{val'}, I_{w_i})$ is the return value of Algorithm 11 applied to node n_i.

Lemma 6.3.2.1 (Correctness of the feasibility test) *Function $test\_feasibility$ always terminates and returns true if the new required set of SLAs for outputting the QoS level $Q^i_{val'}$ at work unit w_i is feasible or false otherwise.*

Proof 6.3.2.1 *Termination comes from the finite number of tasks τ_i being executed in node n_i and from the finite number of the k QoS dimensions and j attributes being tested. The number of QoS attributes being manipulated decreases whenever a task τ_i is configured to be served at its lowest admissible QoS level $Q_{kj}[n]$, thus leading to termination.*

Correctness comes from the heuristic selection of the QoS attribute to downgrade at each iteration of the algorithm.

Thus, after a finite number of steps the algorithm either finds a new set of feasible SLAs that complies with the coordination request or returns false if, even when all tasks are configured to be served at their lowest requested QoS level, the requested SLA for a work unit w_i cannot be supplied.

Given a connected graph $\mathcal{G}_W = (\mathcal{V}_W, \mathcal{E}_W)$, such that the work unit $w_i \in W$ is being processed by node $n_i \in \mathcal{V}_W$, and $I_{w_i} = \{(n_j, w_j, Q^j_{val}), \ldots, (n_k, w_k, Q^k_{val})\}$ as the current set of QoS inputs of w_i, and given T as the set of changed QoS inputs in response to the coordination request, the function $update(I, T)$ updates I with the elements from T:

$$
\begin{aligned}
update(\emptyset, T) &= \emptyset \\
update(I, T) &= \{(n_i, w_i, Q^i_{val'})\} \cup update(I \setminus (n_i, w_i, Q^i_{val}), T), \text{ if } (n_i, w_i, Q^i_{val}) \in I \\
&\quad \text{and } (n_i, w_i, Q^i_{val'}) \in T \\
update(I, T) &= \{(n_i, w_i, Q^i_{val})\} \cup update(I \setminus (n_i, w_i, Q^i_{val}), T), \text{ if } (n_i, w_i, Q^i_{val}) \in I \\
&\quad \text{and } (n_i, w_i, Q^i_{val'}) \notin T
\end{aligned}
$$

Proposition 6.3.2.1 *Given two sets I and T, both with elements of the form (n_i, w_i, Q^i_{val}),* update(I,T) *terminates and returns a new set with the elements of I such that whenever $(n_i, w_i, Q^i_{current}) \in I$ and $(n_i, w_i, Q^i_{new}) \in T$ the pair stored in the returned set is (n_i, w_i, Q^i_{new}).*

Given a node n_i and a work unit w_i, we define the function $get\_input\_qos(n_i, w_i)$ as returning the set of elements (n_j, w_j, Q^j_{val}), where each of these elements represents a work unit w_j being executed at node n_j with an output QoS level of Q^j_{val} used as an input of the work unit w_i at node n_i.

6.3.2.1 Coordinating upgrades

Algorithm 12 Coordinating upgrades

```
temp := n_i
U := ∅
for each n_c ∈ C_W ∪ {n_u} do
    if upgrade(temp, n_c, G_W, Q^c_val') = (TRUE, T) then
        temp := n_c
        U = U ∪ T
    else
        U = ∅
        return
    end if
end for
for each (n_i, Q^i_val') ∈ U do
    Set the new QoS level Q^i_val' for work unit w_i ∈ S
end for
```

Given the connected graph $\mathcal{G}_W = (\mathcal{V}_W, \mathcal{E}_W)$ with a set of cut-vertices $\mathcal{C}_W$ and an end-user node n_u receiving the final outcome of the coalition's processing of service S,

whenever a node $n_i \in \mathcal{V}_W$ is able to upgrade the output of its work unit $w_i \in S$ to a QoS level $Q^i_{val'}$, the other nodes in the coalition respond to this upgrade request according to Algorithm 12.

Given the connected graph $\mathcal{G}_W = (\mathcal{V}_W, \mathcal{E}_W)$ with a set of cut-vertices $\mathcal{C}_W$ and the subgraph that connects node n_i to next cut-vertex $n_c \in \mathcal{C}_\mathcal{W}$, the function $upgrade(n_i, n_c, \mathcal{G}_W, Q^i_{val'})$ is defined by:

function UPGRADE$(n_i, n_c, \mathcal{G}_W, Q^i_{val'})$
  $T := \{(n_i, w_i, Q'_{val})\}$
  for each $n_j \in paths(n_i, n_c) \setminus \{n_i\}$ **do**
    $S := update(get\_input\_qos(n_i, w_i), T)$
    if $test\_feasibility(n_j, w_j, Q'_{val}, S) =$ **TRUE then**
      $T := T \cup \{(n_j, Q'_{val})\}$
    end if
  end for
  $S := update(get\_input\_qos(n_c, w_c), T)$
  if $test\_feasibility(n_c, w_c, Q'_{val}, S) =$ **TRUE then**
    return (**TRUE**, T)
  end if
  return (**FALSE**, $\emptyset$)
end function

Lemma 6.3.2.2 *Given the connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that $n_i \in \mathcal{V}$ and $n_j \in \mathcal{V}$ and a QoS level value Q'_{val}, the call to $upgrade(n_i, n_j, \mathcal{G}, Q'_{val})$ terminates and returns true if n_j is able to output a new QoS level Q'_{val} or false otherwise.*

Proof 6.3.2.2 *Since $\mathcal{V}$ is a finite set and since by Proposition 6.2.0.1 paths terminates and by Proposition 6.3.2.1 update terminates, the number of iterations is finite due to the finite number of elements in the path. Thus, upgrade terminates.*

For any element in the path between n_i and n_j, the new required QoS level Q'_{val} is tested and, by Lemma 6.3.2.1, the upgrade is possible if and only if the new local set of SLAs is feasible. After considering all nodes in the path, the upgrade function returns true and the set of nodes able to upgrade, if node n_j is able to upgrade to Q'_{val}, or false otherwise. Thus, the result follows by induction on the length of the set of elements in the paths between n_i and n_j.

Theorem 6.3.2.1 (Correctness of Upgrade) *Given the connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ representing the QoS inter-dependencies of a serviceS being executed by a coalition of*

nodes, such that $n_u \in \mathcal{V}$ is the end-user node receiving the service at a QoS level Q_{val}, whenever a node n_i announces an upgrade to Q'_{val}, Algorithm 12 changes the set of SLAs at nodes in $\mathcal{G}$ such that n_d receives S upgraded to the QoS level Q'_{val} or does not change the set of local SLAs at any node and n_u continues to receive S at its current QoS level Q_{val}.

Proof 6.3.2.3 *Termination comes from the finite number of elements in $\mathcal{C} \cup n_u$ and from Lemma 6.3.2.2.*

Algorithm 12 applies the function upgrade iteratively to all nodes in the subgraph starting with n_i and finishing in n_u. The base case is when there are no cut-vertices and there is only one call to upgrade. It is trivial to see that the result of upgrade will consist in true and a set of nodes that will upgrade for the new QoS level Q'_{val} or false and an empty set and, by Lemma 6.3.2.2, it is correct. The remaining cases happen when there are one or more cut-vertices between n_i and n_u. Here, upgrade will be applied to all subgraphs starting in n_i and finishing in n_d. Each of these subgraphs are sequentially tested and only if all of them can be upgraded the service S will be delivered to node n_u at the new upgraded QoS level Q'_{val}. The result follows by induction in the number of cut-vertices.

6.3.2.2 Coordinating downgrades

Given the connected graph $\mathcal{G}_W = (\mathcal{V}_W, \mathcal{E}_W)$ with a set of cut-vertices $\mathcal{C}_W$ and an end-user node n_u receiving the final outcome of the coalition's processing of service S, whenever a node $n_i \in \mathcal{V}_W$ needs to downgrade the quality of the output of a work unit $w_i \in S$ from its current QoS level of Q^i_{val} to a downgraded QoS level $Q^i_{val'}$, the other nodes in the coalition respond to this downgrade request according to Algorithm 13.

Algorithm 13 Coordinating downgrades

```
1: temp := n_i
2: for each n_c ∈ C_W ∪ {n_u} do
3:     if downgrade(temp, n_c, G_W, Q^c_val') = FALSE then
4:         temp := n_c
5:     else
6:         Downgrade was compensated and n_c continues to output Q^c_val
7:         break
8:     end if
9: end for
```

Given the connected graph $\mathcal{G}_W = (\mathcal{V}_W, \mathcal{E}_W)$ with a set of cut-vertices $\mathcal{C}_W$ and the sub-graph that connects node n_i to next cut-vertex $n_c \in \mathcal{C}_\mathcal{W}$, the function $downgrade(n_i, n_c, \mathcal{G}_W, Q^c_{val'})$ is defined by:

function DOWNGRADE$((n_i, n_c, \mathcal{G}_W, Q^c_{val'}))$
  $T := \{(n_i, Q^i_{val'})\}$
  for each $n_j \in paths(n_i, n_c) \setminus \{n_i\}$ **do**
    $D := update(get\_input\_qos(n_j, w_j), T)$
    if $test\_feasibility(n_j, w_j, Q^j_{val}, D) =$ **TRUE then**
      $T := T \cup \{(n_j, Q_{val})\}$
    else
      $set\_qos\_level(n_j, w_j, Q^j_{val'})$
    end if
  end for
  $D := update(get\_input\_qos(n_c, w_c), T)$
  if $test\_feasibility(n_c, w_c, Q^c_{val}, D) =$ **TRUE then**
    return TRUE
  else
    for each $n_j \in paths(n_i, n_c) \setminus \{n_i\}$ **do**
      $set\_qos\_level(n_j, w_j, Q^j_{val'})$
    end for
    return FALSE
  end if
end function

Lemma 6.3.2.3 *Given the connected graph $\mathcal{G}_W = (\mathcal{V}_W, \mathcal{E}_W)$ such that $n_i \in \mathcal{V}_W$ and $n_j \in \mathcal{V}_W$ and n_j currently outputs a QoS level Q^j_{val}, the call to $downgrade(n_i, n_j, \mathcal{G}_W, Q^i_{val'})$ terminates and returns true if n_j is able to keep its current output level Q^j_{val} or false otherwise.*

Proof 6.3.2.4 *Since $\mathcal{V}_W$ is a finite set and since, by Proposition 6.2.0.1, paths terminates and by Proposition 6.3.2.1 update terminates, the number of iterations is finite due to the finite number of elements in the path. Thus, downgrade terminates.*

For any element n_k in the possible paths between n_i and n_j, it is tested if that node, given its new set of inputs I_{w_k}, can continue to output its current QoS level Q^k_{val}. After considering all k nodes in the possible paths, the downgrade function returns true, if node n_j is able to continue to output Q^j_{val}, or sets all the k previous nodes in the possible paths between n_i and n_j to the downgraded QoS level $Q^k_{val'}$ and returns

false. Again the result follows by induction on the length of the set of elements in the paths between n_i and n_j.

Theorem 6.3.2.2 (Correctness of Downgrade) *Given the connected graph $\mathcal{G}_W = (\mathcal{V}_W, \mathcal{E}_W)$ representing the QoS inter-dependencies of a service S being executed by a coalition of nodes such that $n_u \in \mathcal{V}_W$ is the end-user node receiving S at the QoS level Q^u_{val}, whenever a node n_i is forced to downgrade the quality of the output of a work unit $w_i \in S$ from its current QoS level of Q^i_{val} to a degraded QoS level $Q^i_{val'}$, Algorithm 13 changes the set of SLAs at nodes in $\mathcal{G}_W$ such that n_u continues to receive S at its current QoS level Q^u_{val} or sets all nodes to a degraded QoS level of $Q^i_{val'}$.*

Proof 6.3.2.5 *Termination comes from the finite number of elements in $\mathcal{C}_W \cup n_u$ and from Lemma 6.3.2.3.*

The correctness trivially follows by the correctness of Lemma 6.3.2.3 and by induction on the number of elements in $\mathcal{C}_W \cup \{n_u\}$.

6.3.3 Number of exchanged messages

In the previous sections we presented the formalisation of the two main coordination operations, namely upgrades and downgrades of the currently supplied QoS level Q_{val} for a service S, as a reaction to a change in the quality of inter-dependent inputs sent by adjacent nodes. In this section we analyse the number of exchanged messages in such coordination operations.

We start with some definitions.

Definition 6.3.3.1 *Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the in-degree of a node $n_i \in \mathcal{V}$ is the number of edges that have n_i as their destination.*

Definition 6.3.3.2 *Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the out-degree of a node $n_i \in \mathcal{V}$ is the number of edges that have n_i as their starting node.*

Whenever an upgrade to a new QoS level Q'_{val} is requested by a node n_i, if the next cut-vertex n_c in the graph $\mathcal{G}$ on QoS inter-dependencies cannot supply the requested upgrade, then all the precedent nodes between n_i and n_c are kept in their currently supplied QoS level Q_{val}. Thus, the number of needed messages is given by the number of edges in the paths between the n_i and n_c where it was determined that the requested

upgrade was not possible. On the other hand, if the upgrade is possible, the number of needed messages is twice the number of edges between n_i and the end-user node n_u. This is because an upgrade is only possible after all the involved nodes are queried and the conjunction of their efforts results in a upgraded QoS level being delivered to n_u.

Whenever, due to resource limitations, a node n_i announces a downgrade to Q'_{val}, the next nodes nodes in the sub-graph from n_i to the next cut-vertex n_c try to compensate the downgraded input quality in order to keep outputting the previous QoS level Q_{val}. When the cut-vertex n_c is reached two scenarios may occur. In the first one, the cut-vertex cannot compensate the degradation, although some of its precedent nodes may. In this case, all the precedent nodes are informed that they can downgrade their current QoS level to Q'_{val} since their compensation effort is useless. Note that, in the worst case, this can be propagated until the final node n_u is reached and all the coalition members will downgrade their current QoS level. As such, in the worst case, a message is sent from each node to its adjacent ones and a reply is received, which demands a total number of messages of two times the number of edges between n_i and n_u. On the other hand, in the second possible scenario, some cut-vertex n_k may be able to compensate the downgraded input quality and continue to produce the current QoS level Q_{val}. In this case, the coordination process is restricted to the subgraph between n_i and n_k. As such, coordination messages are exchanged in this subgraph only.

Thus, in the worst case, the maximum number of exchanged messages in a coordination operation is given by Equation 6.1.

$$\sum_{n \in \mathcal{V}} (out\_degree(n) + in\_degree(n)) \tag{6.1}$$

6.4 Summary

This chapter addressed the challenging problem of providing support for runtime inter-dependent QoS adaptations that either span multiple local tasks and/or multiple hosts in a coalition. QoS inter-dependency relations specify that a task offers a certain level of QoS under the condition that some specified QoS will be offered by the environment or by other tasks. As such, the system has to adapt the QoS level of individual services according both to intra and inter-service QoS dependencies when searching for the best overall service utility.

Solving this dependency problem among local QoS dimensions is equivalent to finding an assignment of values to all QoS attributes such that all dependency constraints are locally satisfied. This chapter proposed anytime QoS optimisation and adaptation algorithms that track QoS dependencies and propagate the performed changes to all the affected attributes at each iteration, ensuring that a valid solution is available at any time.

Furthermore, whenever the effects of these autonomous individual adaptations span multiple nodes in a coalition, coordination is crucial to maintain the correctness of a service's execution and a desirable system's performance. As a result, each node, although autonomous, is influenced by, and can influence, the behaviour of other nodes in a coalition.

This chapter proposed an one-step decentralised coordination model based on an effective feedback mechanism to reduce the complexity of the needed interactions among nodes until a collective adaptation behaviour is determined. Positive feedback is used to reinforce the selection of the new desired global service solution, while negative feedback discourages nodes to act in a greedy fashion as this adversely impacts on the provided service levels at neighbouring nodes.

Chapter 7

Scheduling inter-dependent task sets

Most of the reservation-based scheduling algorithms proposed so far in the literature only support independent task sets. Tasks are not allowed to block or suspend their execution, otherwise certain properties, such as isolation among tasks, cannot be guaranteed. This is a major limitation for their implementation in several real-time scenarios where tasks communicate through shared memory regions, access mutually exclusive resources, or exhibit precedence constraints.

This chapter proposes the Capacity Exchange Protocol (CXP), a new strategy to schedule tasks that share resources and exhibit precedence constraints without any previous precise information on critical sections and computation times. The concept of bandwidth inheritance is combined with the greedy capacity sharing and stealing policy of CSS to efficiently exchange bandwidth among tasks, minimising the degree of deviation from the ideal system's behaviour caused by inter-application blocking.

7.1 Introduction

In Chapter 5 we assumed that tasks were independent, i.e. with no relationships between them. However, in many real-time systems, inter-task dependencies may appear: some tasks have to respect a processing order, data is exchanged among tasks, or they need to use some resources in exclusive mode.

From a modelling point of view, there are two kinds of typical dependencies that can be specified on real-time tasks: (i) precedence constraints, whenever a task must wait the completion of another task before beginning its own execution; and (ii) mutual exclusion constraints, to protect access to shared resources such as data structures, memory areas, external devices, registers, etc.

Until now, a great amount of work has been addressed to minimise the adverse effects of blocking when considering shared resources and precedence constraints among tasks. Resource sharing protocols such as the Priority Ceiling Protocol [SRL90], Dynamic Priority Ceiling [CL90], and Stack Resource Policy [Bak90] have been proposed to provide guarantees to hard real-time tasks accessing mutually exclusive resources. Solutions based on these protocols were already proposed [Jef92, CS01, CBT05, Bar06] but they all require a prior knowledge of the maximum resource usage and, as such, cannot be directly applied to open real-time systems.

The effectiveness and reduced complexity of CSS, proposed in Chapter 5, in managing unused reserved capacities without any previous complete knowledge about the tasks' runtime behaviour makes it appropriate to be used as the basis of a more powerful scheduler able to handle dependent tasks sets which share access to some of the system's resources and exhibit precedence constraints.

The purpose of this chapter is to address both problems, proposing the Capacity Exchange Protocol (CXP) which integrates the concept of bandwidth inheritance [LLA01] with the greedy capacity sharing and stealing policy of CSS. Rather than trying to account borrowed capacities and exchanging them later in the exact same amount, CXP focus on greedily exchanging extra capacities as early, and not necessarily as fairly, as possible. The achieved results suggest that the followed approach effectively minimises the impact of bandwidth inheritance on blocked tasks, outperforming other available solutions.

This work is partially presented in [NP07b, NP08c].

7.2 System model

This section extends the system model used in Chapter 5, where independent task sets were assumed.

A service can be composed by a set of dependent real-time and non-real-time tasks which can generate a virtually infinite sequence of jobs. The j^{th} job of task τ_i arrives at time $a_{i,j}$, is released to the ready queue at time $r_{i,j}$, and starts to be executed at

time $s_{i,j}$ with deadline $d_{i,j} = a_{i,j} + T_i$, with T_i being the period of τ_i. The arrival time of a particular job is only revealed at runtime and the exact execution requirements $e_{i,j}$, as well as which resources will be accessed and by how long they will be held, can only be determined by actually executing the job to completion until time $f_{i,j}$. These times are characterised by the relations $a_{i,j} \leq r_{i,j} \leq s_{i,j} \leq f_{i,j}$.

Tasks may simultaneously need exclusive access to one or more of the system's resources R, during part or all of their executions. If task τ_i is using resource R_i, it locks that resource. Since no other task can access R_i until it is released by τ_i, if τ_j tries to access R_i it will be blocked by τ_i. Blocking can also be indirect (or transitive) if although two tasks do not share any resource, one of them may still be indirectly blocked by the other through a third task.

Tasks may also exhibit precedence constraints among them. A task τ_i is said to precede another task τ_k if τ_k cannot start until τ_i is finished. Such precedence relation is formalised as $\tau_i \prec \tau_k$ and guaranteed if $f_{i,j} \leq s_{k,j}$.

Precedence constraints are defined in the service's description at admission time by a directed acyclic graph G, where each node represents a task and each directed arc represents a precedence constraint $\tau_i \prec \tau_k$ between two tasks τ_i and τ_k. Given a partial order $\prec$ on the tasks, the release times and deadlines are said to be consistent with the partial order if $\tau_i \prec \tau_k \Rightarrow r_{i,j} \leq r_{k,j}$ and $d_{i,j} < d_{k,j}$.

Each accepted real-time task τ_i is associated to a CSS server S_i characterised by a pair (Q_i, T_i), where Q_i is the server's maximum reserved capacity and T_i its period. Recall that these values are based on average estimations for soft real-time tasks.

The schedulability of hard real-time tasks can be guaranteed as long as it is possible to perform an accurate analysis and bound the execution times of hard tasks, their minimum inter-arrival times, and the duration of the accessed critical sections and maximum blocking time, independently of the behaviour of other tasks in the system. Please refer to Section 7.6 for a detailed analysis.

At any given time, it is selected for execution the server with the earliest deadline and pending work to do, based on the EDF priority assignment. When no server is selected, the processor is idle or it is executing non-real time tasks.

7.3 Sharing resources in open systems

As discussed in Chapter 5, CSS can effectively reduce the mean tardiness of periodic soft real-time tasks through an efficient management of unused reserved capacities under the assumption that tasks do not share any of the system's resources. In fact, if classic mutual exclusion semaphores are used with CSS, a particular problem arises, usually referred as priority inversion [SRL90]. If a higher priority task is blocked on a semaphore by a lower priority task and another medium priority arrives, the latter can preempt the lower priority task causing an unbounded blocking delay to the higher priority task.

Resource sharing among tasks of open real-time systems started to be addressed in [LLA01]. The proposed Bandwidth Inheritance (BWI) protocol extends the CBS scheduler to work in the presence of shared resources, adopting the Priority Inheritance Protocol (PIP) [SRL90] to handle tasks' blocking. Although the PIP was initially thought in the context of fixed priority scheduling, it has been shown that it can be applied to dynamic priority scheduling, holding its basic properties: it limits the worst-case blocking that must be endured by a job j to the duration of at most $min(n, m)$ critical sections where n is the number of jobs with lower priority than j and m the number of different semaphores used by j.

While BWI allows a shared access to the system's resources without requiring any prior knowledge about the tasks' structure and temporal behaviour it also guarantees that tasks that do not access shared resources are not affected by the behaviour of other tasks.

However, its main drawback is its unfairness in bandwidth distribution. A blocking task can use most (or all) of the reserved capacity of one or more blocked tasks, without compensating the tasks it blocked. Blocked tasks may then lose deadlines that could otherwise be met.

At the same time, servers keep postponing their deadlines and recharging their capacities on every capacity exhaustion, potentially severely delaying blocked tasks with earlier deadlines which will finish later than tasks with longer deadlines. It is known that allowing a task to use resources allocated to the next job of the same task may cause future jobs of that task to miss their deadlines by larger amounts [NP07a, LB05]. This violation of the original capacity distribution can have a huge negative impact in the overall system's performance.

Figure 7.1 illustrates these problems with a simple example. Three servers $S_1 = (2, 5)$, $S_2 = (1, 3)$, and $S_3 = (1, 4)$ serve three tasks with execution times equal to their

respective servers' capacity. Tasks τ_1 and τ_2 share access to resource R for the entire duration of their execution times, while τ_3 is independent from the other two.

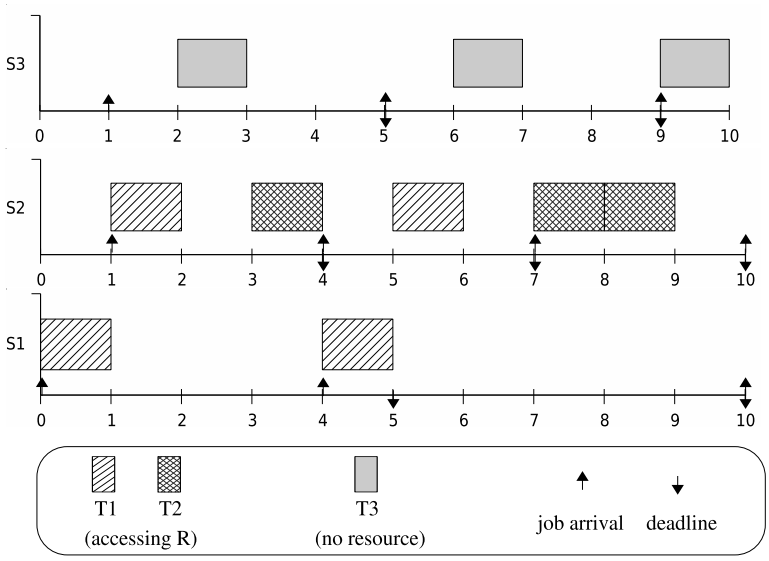

Figure 7.1: BWI's drawbacks

Note how an early arrival of the second job of task τ_1 at time $t = 4$ allows τ_1 to consume 3 units of reserved bandwidth in the interval $[0, 5]$, more than its initial reservation. The nonexistence of a compensation mechanism and the automatically deadline update are responsible for the deadline miss of the second job of task τ_2.

To address the lack of a compensation mechanism, BWE [WLP02] and CFA [SLS04] try to fairly compensate blocked servers in exactly the same amount of capacity that was consumed by a blocking task while executing in a blocked server. To achieve this, BWE maintains a global $n * n$ matrix (n is the number of servers in the system) in order to record the amount of capacity that should be exchanged between servers, a capacity list at each server to keep track of available budgets, and dynamically manages resource groups[1] at each blocking and releasing of a shared resource. CFA requires each server to manage two task lists with different priorities and a counter that keeps track of the amount of borrowed capacity from a higher priority server, converting the inheritor into a debtor. Contracted debts are payed by blocking servers, until the blocked servers' counters are successively decremented to zero.

The increased computational complexity of these attempts to fairly compensate borrowed capacities and the fact that CSS tends to fairly distribute residual capacities in the long run [NP07a], lead us to propose a simple and efficient capacity exchange

[1]Groups of tasks that access a particular resource R

protocol that merges the benefits of a smart greedy capacity reclaiming policy with the concepts of bandwidth inheritance and hard reservations. Adding to the lower complexity of our approach, taking advantage of all of the available capacity instead of only exchanging capacities within the same resource group leads to a better system's performance in dynamic open real-time systems.

7.4 The Capacity Exchange Protocol (CXP)

The Capacity Exchange Protocol (CXP) merges the benefits of the capacity sharing and stealing approach of CSS with the concept of bandwidth inheritance to allow a task τ_i to be executed on more than its dedicated server S_i, efficiently exchanging capacities among servers and reducing the undesirable effects caused by inter-application blocking.

CXP adds to a CSS server a list of served tasks ordered by the tasks' deadlines. Initially, each server has only its dedicated task in the task list and, as long as no task is blocked, servers behave as in the original CSS scheduler. With blocking, the following rules are introduced:

- **Rule E:** When a high priority task τ_i is blocked by a lower priority task τ_j when accessing the shared resource R, τ_j is inherited by server S_i. The execution time of τ_j is now accounted to the currently pointed server by S_i. If task τ_j has not yet released the shared resource R when S_i exhausts all the capacity it can use, τ_j continues to be executed by the earliest deadline server with available capacity that needs to access R, until τ_j releases R.

- **Rule F:** If a blocking task τ_j is inherited by a blocked server S_i, delaying the execution of its dedicated task τ_i, then τ_i is also added to S_j's task list. When task τ_i is unblocked it is executed by the earliest deadline server which has τ_i in its task list until it is finished or the server exhausts all the capacity it can use (whatever comes first).

- **Rule G:** If at time t, no active server with pending jobs can continue to execute through one of the rules B, C, or D, and there is at least one active server S_r with residual capacity greater than zero, it is possible to use those available residual capacities with deadlines greater than the one assigned to the current job $j_{p,k}$ of the earliest deadline server S_p with pending work to execute $j_{p,k}$ through bandwidth inheritance.

Rule E describes the integration of the bandwidth inheritance mechanism in the dynamic capacity accounting of CSS. The currently executing server always consumes the pointed capacity, either its own or another available valid capacity in the system.

Rule F proposes to exchange reserved capacities among servers due to blocking without the goal of a fair compensation, reducing the complexity and overhead of CXP. It allows a blocked task τ_i that has been delayed in its execution to be executed by the earliest deadline server with available capacity which has τ_i in its task list. Note that, with bandwidth inheritance, this server may now be different from S_i.

In general, the hard reservation approach may cause the loss of more deadlines since once a server's capacity is depleted, capacity recharging is suspended until the server's next activation. To minimise this and take advantage of a more constant rate in tasks' execution, Rule G allows the use of bandwidth inheritance to execute unfinished tasks, including those from servers that do not directly or indirectly share any resource with the selected server, if at a particular time no active server in the system is able to reclaim new residual capacities or steal inactive non-isolated capacities to continue executing its pending work after a capacity exhaustion.

Since the queue of active servers is ordered by deadlines, CXP easily keeps track of the earliest deadline server with pending work and no capacity left S_p, as well as the earliest deadline server with available residual capacity S_r, when traversing the queue to select the next running server. If the end of the active queue is reached without finding a server with pending work and available capacity, server S_r is selected as the running server and inherits the first task of S_p' list. S_r executes the task, consuming its own residual capacity. Since a server always starts to consume the earliest residual capacity available, no modification to the capacity accounting mechanism is needed to correctly account for the consumed capacity.

Note that Rules A and B of the original CSS scheduler ensure that residual capacities originated by earlier completions can be reclaimed by any active eligible server. Blocked servers can then take advantage of any residual capacity, even if it is released by a server that does not share any resource with the reclaiming server.

7.4.1 Minimising the cost of blocking with CXP

While preserving the isolation principles of independent tasks and inheritance properties of critical sections of BWI, CXP introduces significant improvements in the system's performance. Figure 7.2 illustrates how CXP can minimise the cost of blocking by efficiently exchanging reserved capacities among servers, scheduling the

same set of tasks used to analyse BWI's drawbacks in Figure 7.1.

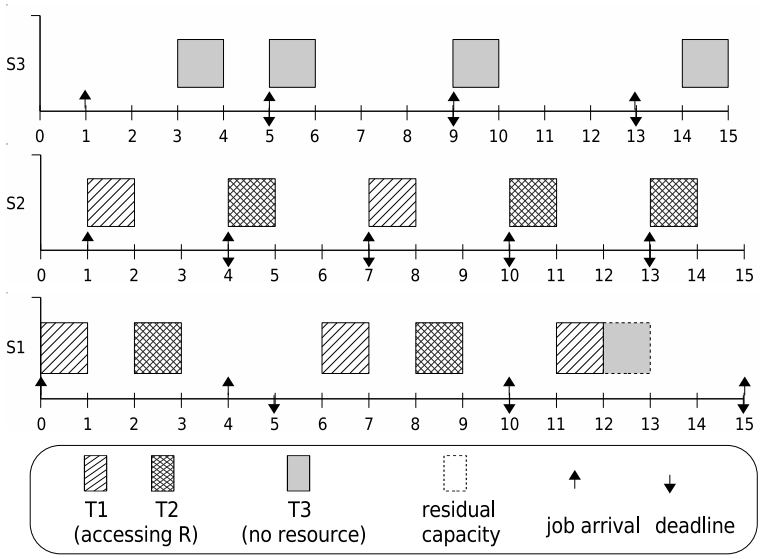

Figure 7.2: Sharing resources with CXP

At time $t = 1$, task τ_2 is added to the task list of server S_1 (Rule F). At time $t = 2$, task τ_2 is unblocked and it is executed by server S_1, since it is the earliest deadline server with remaining capacity with τ_2 in its task list (the same happens at time $t = 8$). Note that capacities are exchanged between all the system's servers and not only within a specific resource group, maximising the use of extra capacities to handle overloads and still meet deadlines. An overload of the independent task τ_3 was handled by reclaiming the residual capacity originated by an earlier completion of task τ_1 at time $t = 12$.

Since the execution and inter-arrival times of jobs are not known in advance it is important to minimise the impact of misbehaved tasks that exceed their expected execution times or have a shorter inter-arrival time of jobs. Note that despite the earlier arrival of the second job of task τ_1 at time $t = 4$, the deadline of server S_1 is not set to $d_{1,2} = 9$ and the job is only released at time $t = 5$, following a hard reservation approach.

7.5 Handling precedence constraints in open systems

Additional constraints that may affect the schedulability of real-time systems arise when the execution of the data's producer must precede the execution of the consumer

of that data. In more complex scenarios, both shared resources and precedence constraints can be present among tasks. This section describes an unified approach to handle these two types of constraints.

It is well known that precedence constraints can be guaranteed in real-time scheduling by priority assignment. In fact, with dynamic scheduling, any task will always precede any other task with a later deadline. This suggests that precedence constraints that are consistent with the tasks' deadlines do not affect the schedulability of the task set. In fact, the idea behind the consistency with the partial order is to enforce a precedence constraint by using an earlier deadline.

Formal work exists, showing how to modify deadlines in a consistent manner so that EDF can be used without violating the precedence constraints. Garey et al. [GJST81] show that the consistency of release times and deadlines can be used to integrate precedence constraints in the task model. Spuri and Stankovic [SS94] introduce the concept of quasi-normality to give more freedom to the scheduler so that it can also obey shared resource constraints, and provide sufficient conditions for schedules to obey a given precedence graph. The authors prove that with deadline modification and some type of inheritance it is possible to integrate precedence constraints and shared resources. Mangeruca et al. [MFSV06] consider situations where the precedence constraints are not all consistent with the tasks' deadlines and show how schedulability can be recovered by considering a constrained scheduling problem based on a more general class of precedence constraint.

However, all these works base their modifications of deadlines on a previous knowledge of the tasks' execution times. To make use of these previous results in open real-time systems, the consistency of release times and deadlines with the partial order must be enforced considering estimated execution times when applying some known technique [GJST81, SB94, MFSV06, Bla77, CSB90] at admission time.

However, such approach immediately raises two questions: (i) what happens if a precedent task requires more capacity than declared? (ii) how can a task know if all its predecessors have already finished?

CXP provides answers for both questions and can be used to handle blocking due to precedence violations in the same way as for a critical section blocking, minimising the impact of misbehaved tasks on the overall system's performance. We base our approach on the idea that if task $\tau_j \prec \tau_i$ has not yet finished at time $s_{i,k}$, when the k^{th} instance of τ_i is selected to execute, it is blocking its successor.

Given a partial order $\prec$ on the tasks, described by a directed graph G, servers' state

changes in CXP allow an easy verification of the current condition of a precedent task τ_j. Recall that a server that has completed its job is only kept active until its deadline if it is supplying some residual capacity originated by an earlier completion of its previous job. As such, by adding the following rule to CXP, we are able to handle precedence constraints among tasks of open real-time systems without any previous complete knowledge of their actual behaviour during runtime.

- **Rule H:** If a precedent server S_j is active at time $s_{i,k}$, whenever server S_i is scheduled for execution it must check the current value of S_j's residual capacity. If its equal to zero, then the current task τ_j of S_j has not yet been completed and must be added to S_i's task list.

Note that precedence constraints can then be handled by Rule H as an access to a shared resource without introducing overhead in the protocol. Since CXP reclaims available residual capacities as earlier as possible, whenever a server S_i is scheduled for execution it already checks the current state of the residual capacity of active earlier deadline servers.

7.5.1 Handling tasks' precedences with CXP

The next example illustrates how CXP can easily handle precedence constraints among tasks whose actual computation times are only revealed at run time. Figure 7.3 shows a possible scheduling of three servers $S_1 = (2, 8)$, $S_2 = (4, 10)$, and $S_3 = (3, 15)$ used to serve three periodic soft real-time tasks, based on their estimated average execution times and periods, exhibiting the following precedence constraints $\tau_1 \prec \tau_2 \prec \tau_3$.

At time $t = 3$, the successor server S_2 knows it has to complete its predecessor's task since S_1 is still active and its residual capacity is set to zero. As such, task τ_1 needs to be executed in server S_2, prior to the execution of τ_2.

On the other hand, at times $t = 6$ and $t = 10$, both servers S_3 and S_1 can start executing their dedicated tasks. At time $t = 6$, S_2 becomes inactive by completing τ_2 and exhausting its capacity. Its inactive state clearly indicates that task τ_2 has been completed. Similarly, at time $t = 10$, the predecessor server S_1 is active but with residual capacity available. This is only possible when a server has completed its current task using less that its budgeted capacity.

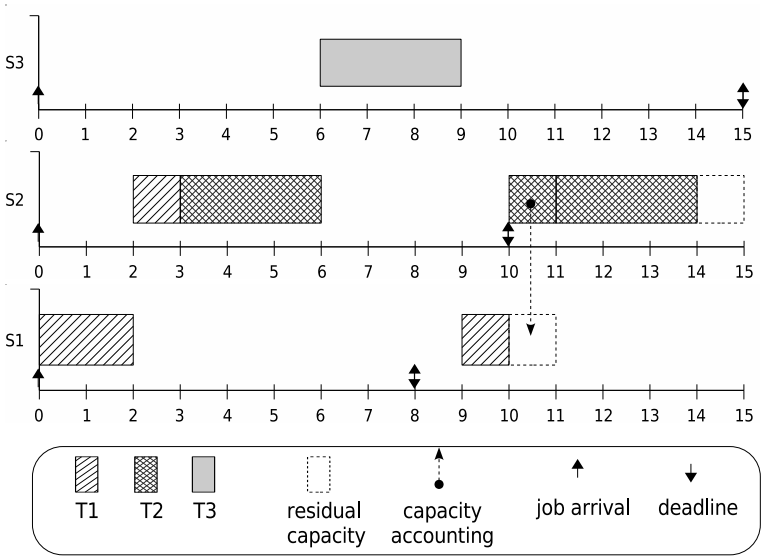

Figure 7.3: Handling tasks' precedences with CXP

7.6 Theoretical validation for dependent tasks

As shown, CXP is particularly suitable to schedule soft real-time tasks without requiring any offline knowledge of how many services will be concurrently executed, which resources will be accessed, nor by how long they will be held.

However, enabling resource sharing among hard (HRT) and soft real-time (SRT) tasks in open systems is not straightforward. Demanding that SRT tasks declare the maximum duration of the critical sections on each accessed resource at admission time is against the basic purpose of an open system itself. Nevertheless, HRT tasks still need to be guaranteed based on the knowledge of their worst-case behaviour.

One way to achieve such guarantee in an open system is to implement the critical sections as library functions whose WCET can be determined. Of course, this comes at the cost of some pessimism. Nevertheless, serving HRT tasks must always be based on a reserved capacity equal to their WCETs.

Furthermore, if nested critical sections are allowed, the system's libraries must also impose a totally ordered access to resources, since for a deadlock to be possible a blocking chain must exist in which there is a circular relationship. Deadlocks can be detected and exceptions raised if a misbehaving task attempts to acquire resources in an improper order, by following the chain of accessed resources and detecting a resource that is already in the list.

In the remaining of this section, we assume that resources are orderly accessed through shared libraries and discuss how to assign the maximum capacity Q_i and period T_i to an isolated server which has to serve a hard real-time task τ_i in an open system with n hard reservation servers with a total utilisation of $\sum_{i=1}^{n} \frac{Q_i}{T_i} \leq 1$.

We start by proving the correctness of the proposed capacity exchange mechanism of CXP.

Definition 7.6.0.1 *At a particular time instant t, the total amount of available execution capacity C_a in the system is the sum of the remaining reserved capacities greater than zero that can be used to execute a task (either the remaining execution or residual capacities of active servers or the remaining execution capacities of inactive non-isolated servers whose capacity can be stolen by active servers).*

Lemma 7.6.0.1 *Just after a task τ_i releases the shared resource R, the total amount of available execution capacity C_a in the system is the same as in the non-resource sharing case.*

Proof

While task τ_i is accessing the shared resource R during t units of time, it can block some other task. It follows from the bandwidth inheritance protocol that when a task blocks another one it inherits the latter's server. Furthermore, as proven by Theorem 5.4.0.1 in Chapter 5, the dynamic budget accounting mechanism used in CXP does not affect the system's schedulability.

Hence, the total amount of available system's execution capacity C_a when task τ_i releases the shared resource R is independent of whether the task was executed only by its dedicated server S_i or not. In the worst case, the longest time a server can be connected to another server is bounded by the currently pointed server's capacity and deadline.

□

Lemma 7.6.0.2 *No capacity is exchanged after its deadline*

Proof

Let $a_{i,k}$ denote the time instant at which the k^{th} instance of task τ_i arrives and its dedicated server S_i is inactive. At $a_{i,k}$, a new execution capacity $c_i = Q_i$ is generated. If S_i is a non-isolated server and some amount c_s of its reserved capacity was stolen while it was inactive, the server becomes active with the remaining execution capacity $c_i = Q_i - c_s$.

Let $\forall_{i,k}\ d_{i,k} = max\{a_{i,k}, d_{i,k-1}\} + T_i$ be the deadline and $\forall_{i,k}\ r_i = d_{i,k}$ be the replenishment time associated with capacity c_i.

Let L be the task list of server S_i. L is composed at least by jobs of task τ_i, but can also contain, due to blocking, inherited tasks (Rule E) and tasks that were delayed by the execution of τ_i in high priority servers (Rule F).

Let $[t, t + \Delta_t[$ denote a time interval during which server S_i is executing the earliest unblocked task of L, consuming its own reserved capacity c_i. Consequently, S_i has used an amount equal to $c'_i = c_i - \Delta_t \geq 0$ of its own capacity during this period. As such, c_i must be decreased to c'_i, until its value is equal to zero.

Let $f_{i,k}$ denote the time instant when server S_i completes the last job of L. The remaining execution capacity $c_i > 0$ is released as residual capacity $c_r = c_i$ and c_i is set to zero.

At the time instant $f_{i,k}$, the next active server S_j with pending work and remaining execution capacity is scheduled for execution, according to the EDF policy. If the inequality $d_{i,k} \leq d_{j,l}$ holds, server S_j can use the released residual capacity c_r until its deadline $d_{i,k}$ or $c_r = 0$.

Let $[t, t+\Delta_t[$ denote a time interval during which server S_j is executing, consuming the residual capacity c_r. Consequently, S_j has used an amount equal to $c'_r = c_r - \Delta_t \geq 0$ of S_i's residual capacity during this period. As such, c_r must be decreased to c'_r, until its value is equal to zero.

If at some instant t all active servers have exhausted the amount of execution capacities they can use and there are unfinished jobs, the job of the earliest deadline unfinished task τ_u is added to the task list of the earliest deadline active server S_r with residual capacity $c_r > 0$. Assume that S_i is the selected server.

Let $[t, t+\Delta_t[$ denote a time interval during which server S_i is executing, consuming its own residual capacity c_r. Consequently, S_i has used an amount equal to $c'_r = c_r - \Delta_t \geq 0$ of its residual capacity during this period. As such, c_r must be decreased to c'_r, until its value is equal to zero.

At replenishment time $t = r_i$ any remaining unused residual capacity c_r of server S_i is

discarded and c_r is set to zero.

□

Theorem 7.6.0.1 *Given a system with n servers with utilisation* $U = \sum_{i=1}^{n} \frac{Q_i}{T_i}$ *which uses CXP for accessing shared resources, it can be guaranteed that, at any time, the system's utilisation U is no more than the utilisation for the case when the served tasks do not share any resource.*

Proof

Without resource sharing, CXP ensures that no server consumes more than its reserved capacity Q_i every period T_i and the amount of capacity that can be reclaimed or stolen is limited in the worst case by the reserved capacity and deadlines of the pointed servers. By directly applying the results of Lemma 7.6.0.1 and Lemma 7.6.0.2, the same properties hold in CXP when tasks share access to resources.

□

Theorem 7.6.0.2 *A blocked task scheduled by CXP never has less available time to complete its execution than under the basic BWI protocol*

Proof

From Rule F, CXP guarantees that a blocked task τ_i resumes its execution in the earliest deadline server which has τ_i in its task list, which may be different from its dedicated server S_i. With BWI, however, the blocked task τ_i is only able to resume its execution when its dedicated server S_i has no more blocking tasks in its task list and is the earliest deadline among all active servers.

As a consequence, the time that is available for a blocked task τ_i to complete its execution may be increased with CXP but never reduced when compared to BWI.

□

After proving the correctness of the capacity exchange mechanism of CXP, we now discuss how to provide guarantees to hard real-time tasks, starting with some important definitions that help to clarify the following analysis.

Definition 7.6.0.2 *Two tasks are in the same resource group G if they directly or indirectly share some resource.*

Definition 7.6.0.3 *Given a task τ_i served by server S_i, the blocking time B_i is defined as the maximum time during which all other tasks can be executed by S_i, for each job of τ_i.*

Lemma 7.6.0.3 *Given a task τ_i served by server S_i, only tasks in the same resource group G can be added to S_i's task list and contribute to B_i, for each instance of τ_i.*

Proof

Initially, each active server has exactly one task in its task list. It follows from the bandwidth inheritance protocol that if a task τ_i is blocked by task τ_j when accessing a resource R, then τ_j is added to the task list of server S_i. If τ_j is also blocked on another resource, the chain of blocking is followed and all the blocked tasks are added to S_i until a non-blocked task is reached. The task list of all other servers remains unchanged. Hence, the number of tasks that can contribute to B_i is restricted to those tasks that belong to the same resource group G.

□

Theorem 7.6.0.3 *If a HRT task τ_i is served by an isolated server S_i with parameters (Q_i, T_i), where the reserved capacity $Q_i = C_i + B_i$ is determined by adding the WCET C_i of τ_i to the maximum blocking B_i that can be experienced by an instance of τ_i, and T_i is the minimum inter-arrival time of τ_i's jobs, then τ_i will meet its deadline, regardless of the behaviour of the other tasks in the system.*

Proof

From Theorem 7.6.0.1 it follows that each isolated server S_i always receives Q_i units of execution capacity every T_i units of time. Lemma 7.6.0.3 assures that the set of tasks that can be executed by S_i is restricted to those tasks in the same resource group G. Hence, if a HRT task τ_i does not access any shared resource it is not affected by the behaviour of other tasks. Therefore, if each instance of τ_i consumes up to $C_i \leq Q_i$ units of execution capacity and instances are separated at least by T_i, is is guaranteed that task τ_i finishes no later than S_i's capacity exhaustion and it will meet all its deadlines.

If a HRT task τ_i accesses some shared resources during its execution, we have to consider the maximum time during which other tasks can be executed by S_i through bandwidth inheritance. It follows from Lemma 7.6.0.3 that whether task τ_i meets its deadline depends only on the timing requirements C_i of task τ_i and on the maximum blocking time B_i that can be experienced by each instance of task τ_i. Hence, in order not to miss any deadline of a HRT task τ_i it is sufficient to assign a capacity of $Q_i = C_i + B_i$ to the isolated server S_i.

□

From Theorem 7.6.0.3 it is possible to derive sufficient conditions for the schedulability of HRT tasks. HRT tasks which do not access any shared resource can be guaranteed exactly like in the original CSS algorithm by assigning them to isolated servers with reserved capacities $Q_i = C_i$, where C_i is the WCET of task τ_i, and periods T_i equal to the minimum inter-arrival times of τ_i's jobs. A HRT task τ_i which accesses shared resources during its execution can be guaranteed if it is assigned to an isolated server S_i whose capacity $Q_i = C_i + B_i$ also accounts for the maximum blocking time B_i that can be experienced by each instance of τ_i.

7.6.1 Blocking time computation

An exact computation of the worst-case blocking time B_i for a HRT task τ_i is a complex problem in open systems where the unpredictable behaviour of SRT tasks may cause the associated servers to exhaust their capacities while inside the critical sections, causing many possible situations in which a SRT task can block a HRT task. Without a complete knowledge of the number, type, and behaviour of tasks that may, directly or indirectly, interact through shared resources with a HRT task τ_i it is impossible to perform an accurate offline analysis and compute the worst case blocking B_i that can be experienced by τ_i without imposing some pessimism.

The dynamic properties of an open real-time systems only allow us to assume that the WCET of the critical sections that may be accessed by any task through the system's libraries can be indirectly computed by an offline analysis of those shared libraries. With nested critical sections, the WCET must consider the worst possible path in the blocking chain. The reader may refer to [WEE+07] for an extensive survey of the current methods and tools to compute WCETs.

This may be considered too pessimistic since to guarantee a set of n HRT tasks the blocking times must all be summed together at admission time, but the dynamic nature

of an open system and lack of information impose such pessimism. It is impossible to completely identify the conditions under which any task that is dynamically admitted in the system can interfere with a HRT task. Of course, this comes at the cost of a lower system's utilisation to guarantee HRT tasks. However, with CXP, SRT tasks can benefit from the unused reserved capacities of HRT tasks, minimising this waste of resources.

If a resource group G is guaranteed to be composed only by HRT tasks, it is possible to explore all possible blocking situations and compute a more accurate and less pessimistic value for B_i, using, for example, an algorithm similar to the one presented in [LLA04].

7.7 Summary

The resource reservation approach is particularly interesting to open real-time systems where new services can enter the system at any time without any previous knowledge about their execution requirements and tasks' inter-arrival times. Tasks can be accepted based only on expected requirements and handled through dedicated servers that prevent the served tasks from demanding more than the reserved amount.

However, with a classic reservation-based approach, if tasks are allowed to block or suspend, inconsistencies can arise and real-time guarantees to hard tasks and probabilistic guarantees for soft tasks cannot be provided. Handling shared resources and precedence constraints among tasks in open systems is then a very challenging problem.

This chapter addressed both types of constraints and proposed the Capacity Exchange Protocol (CXP), a new strategy for open systems that integrates the concept of bandwidth inheritance with the efficient greedy capacity sharing and stealing policy of CSS to minimise the degree of deviation from the ideal system's behaviour caused by inter-application blocking. The reduced complexity of the proposed approach in CXP focus on greedily exchanging extra capacities as early, and not necessarily as fairly, as possible and introduces a novel approach to integrate precedence constraints into the task model.

In addition, a schedulability analysis for a hybrid set of inter-dependent hard and soft real-time tasks has been presented. The analysis is based on the most important formal properties of CXP, presented and proved in this chapter.

Chapter 8

Evaluation

The behaviour of the proposed CooperatES framework in dynamic open real-time scenarios was evaluated through extensive simulations, with a special attention being devoted to the introduction of a high variability in the characteristics of the used scenario. This chapter details the conducted evaluations and discusses the achieved results.

8.1 Introduction

The ideal way to evaluate the performance of the several algorithms proposed in this thesis would be to subject them to actual loads from a large portfolio of real world QoS-aware applications and embedded devices. Nevertheless, we have chosen to evaluate the effectiveness of the CooperatES framework by creating a broad collection of application and device profiles, chosen to cover the spectrum into which real applications and both embedded and their more powerful neighbour devices would fall or likely exhibit. Furthermore, the current scarcity of available QoS-aware applications that can be cooperatively executed by a dynamically formed coalition of nodes invalidates such approach.

The reported results were observed from multiple and independent simulation runs, with initial conditions and parameters, but different seeds for the random values[1] used to drive the simulations, obtaining independent and identically distributed variables. Although the outputs of individual simulation runs are not independent, it is still possible to obtain independent observations across the results of several simulation

[1]The random values were generated by the Mersenne Twister algorithm [MN98] with an uniform distribution.

runs (or simulation replicas) with a reasonably good statistical performance [LK00]. The mean values of all generated samples were used to produce the charts presented in this chapter, with a confidence level of 99,9% associated to each confidence interval [EM].

8.2 Evaluated scenario

The simulator used in the conducted experiments was custom built in Erlang [Lab], a functional programming language designed to be run in a distributed environment populated with resource constrained devices.

An application that captures, compresses and transmits frames of video to end users, which may use a diversity of end devices and have different sets of QoS preferences, was used as a scenario for the simulations. The application was composed by a set of source units to collect the data, a compression unit to gather and compress the data that came from the multiple sources, a transmission unit to transmit the data over the network, a decompression unit to convert the data into each user's specified format, and an user unit to display the data in the user's end device.

The number of simultaneous nodes in the system varied from 10 to 100, while the number of simultaneous users varied from 1 to 20, generating different amounts of load and resource availability. Each node had a fixed set of mappings between requested QoS levels and resource requirements and the code bases needed to execute each of the streaming application's units was loaded a priori in all the nodes.

The characteristics of end devices and their more powerful neighbour nodes was randomly generated from the set of characteristics described in Table 8.1, creating a distributed heterogeneous environment. This non-equal partition of resources affected the ability of some nodes to singly execute some of the application's units and has driven nodes to a coalition formation for a cooperative service execution.

Requested QoS levels were randomly generated, at randomly selected end devices and at randomly generated times, expressing the spectrum of acceptable QoS levels in a qualitative way, ranging from a randomly generated desired QoS level to a randomly generated maximum tolerable service degradation. The relative decreasing order of importance imposed in dimensions, attributes and values was also randomly generated. Similarly, inter-dependency QoS relations among tasks were randomly generated for each service.

The QoS domain used to generate the users' service requests was composed by the

Table 8.1: Possible characteristics of nodes

Resource	Type
cpu	400 MHz, 750 MHz, 1 GHz, 1.5 GHz, 2 GHz, 2.5 GHz, 3 GHz
memory	128 MB, 256 MB, 512 MB, 1 GB, 2 GB, 4 GB, 8 GB
storage	512 MB, 1 GB, 10 GB, 30 GB, 50 GB, 200 GB, 500 GB
network	10 Mbps, 11 Mbps, 54 Mbps, 100 Mbps, 1 Gbps
display	none,240x180,320x240,640x480,720x480, 1024x768, 1280x1024

following list of QoS dimensions, attributes, and possible values:

```
QoS dimensions = {Media Container,Video Quality, Audio Quality}
Media Container = {container format}
Video Quality = {color depth, frame size, frame rate}
Audio Quality = {sampling rate, sample bits}

container format = {3GP, ASF, AVI, QuickTime, RealVideo, WMV}
color depth (bits) = {1, 3, 8, 16, 24}
frame size (pixels) = {240x180, 320x240, 640x480, 720x480,
                       1024x768,1280x1024}
frame rate (per second) = {[1,30]}
sampling rate (kHz) = {8, 11, 32, 44, 88}
sample bits (bits) = {4, 8, 16, 24}
```

Based on each user's service request, coalitions of 4 to 20 nodes were formed using the anytime coalition formation and service proposal formulation algorithms proposed in Chapter 4. Each node was connected at least to another node in the coalition. The maximum degree of each node, that is, the maximum number of connections to a node was set to 3. After the coalition was formed, a random percentage of the connections among its members was selected as a QoS dependency among those work units.

To cope with such a dynamic environment, nodes were requested to adjust their local set of SLAs, either by lowering the currently provided QoS level of some services due to resource limitations or by (re)upgrading them when the needed resources become available.

8.3 Anytime approach's behaviour and overhead

Throughout this thesis, the notion that complex scenarios may prevent the possibility of computing optimal resource allocations was claimed and anytime algorithms that can tradeoff the needed deliberation time for the quality of the achieved results were proposed.

The first set of conducted studies had two main objectives: (i) to analyse the behaviour of the anytime algorithms proposed in Chapters 4 and 6 in highly dynamic scenarios; and (ii) to measure the computational cost of those algorithms when compared to their traditional versions proposed in Chapter 3.

The behaviour of an anytime algorithm is described by its performance profile, a representation of the relationship between processing time and result quality for a particular anytime algorithm and problem. Since there are many possible factors affecting the execution time of an algorithm, rather than measuring the algorithms' absolute execution time on every simulation run, we have normalised it with respect to its completion time [Zil93]. Nevertheless, all the algorithms needed an average time lower than 1 second to compute their optimal solutions on a Intel Core Duo T5500 at 1.66 GHz.

8.3.1 Coalition formation

The first conducted study determined the performance profile of the anytime coalition formation algorithm and compared the usefulness of two methods for selecting, at each iteration of the algorithm, the next service proposal to be evaluated. The first one, as proposed in Chapter 4, selects for evaluation the proposal sent by the node with the highest local reward, while the second one relies on the order of proposals' reception. Note that this second approach is the one followed by the traditional version of the coalition formation algorithm proposed in Chapter 3.

Figure 8.1 details the expected solution's quality as a function of the needed computation time.

The heuristic selection based on the nodes' local reward not only achieves a better performance, but also has a lower variation on the expected solution's quality. At only 20% of the completion time, the anytime coalition formation algorithm achieves a solution's quality of 83% $\pm$ 6% of its optimal solution, determined at completion time.

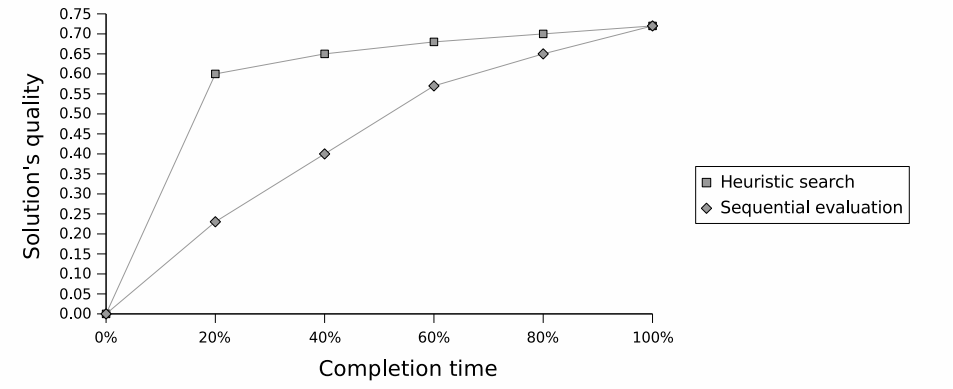

Figure 8.1: Coalition formation: Anytime behaviour

On the other hand, a poorer performance and higher variability is achieved if the algorithm relies on the order of proposal's reception. At 20% of the completion time, the algorithm achieves a solution's quality of 32% ± 25% of its optimal solution, determined at completion time.

A second study measured the computation time needed by both the anytime and traditional versions of the coalition formation algorithm to achieve their optimal solutions at completion time, as well as their first available solution. Figure 8.2 details the results, normalised to the longest approach.

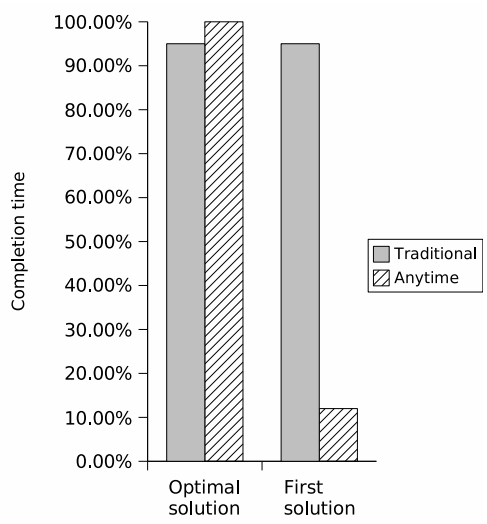

Figure 8.2: Coalition formation: Anytime vs Traditional

The traditional version is slightly faster than its anytime counterpart. It requires nearly 95% ± 2% of the time needed by the anytime approach to reach its optimal

solution at completion time. This difference is explained by the way both algorithms select the next proposal to evaluate. While the traditional version sequentially evaluates service proposals according to their order of arrival, the anytime approach selects proposals based on the nodes' local reward. This implies either to sort the proposals' set before starting the evaluation process or to search, at each iteration, for the maximum remaining local reward.

However, note that the anytime version needs as little as near 12% ± 2% of its completion time to be able to deliver a solution, whose quality is near 50% ± 6% of its optimal solution's quality. On the other hand, the binary notion of the solution's quality of the traditional version of the algorithm, only allows the algorithm to return its optimal solution at the end of its computation.

8.3.2 Service proposal formulation

A third study evaluated the behaviour of the anytime service proposal algorithm by measuring its performance profile as well as the impact generated by the arrival of a new service request on the QoS level of previously accepted tasks.

The results were plotted by averaging the results over several independent runs of the simulation, divided into two categories. Figure 8.3 presents the scenario where the average amount of available resources per node is greater than the average amount of resources demanded by the services being executed. The opposite scenario is represented in Figure 8.4, where the average amount of resources per node is smaller than the average amount of demanded resources.

In Figure 8.3, the increase in the solution's quality Q_{conf} results from the increase in the new task's reward (Step 1 of the algorithm). Recall that with spare resources the QoS levels of previously accepted tasks are kept the same. As such, this increase in the new service's reward also increases the node's local reward, that was affected by the initially proposed solution of serving the newly arrived service at the minimum requested QoS level.

However, due to resource limitations (Figure 8.4), when trying to upgrade the reward achieved by the new service, the generated configuration may result in an unfeasible set of SLAs. Whenever this happens, the algorithm iteratively selects the minimum utility's decrease, until a feasible solution is found that presents a higher satisfaction for the service request under negotiation, if it exists (Step 2 of the algorithm).

Note that, in both scenarios, the anytime service proposal formulation algorithm

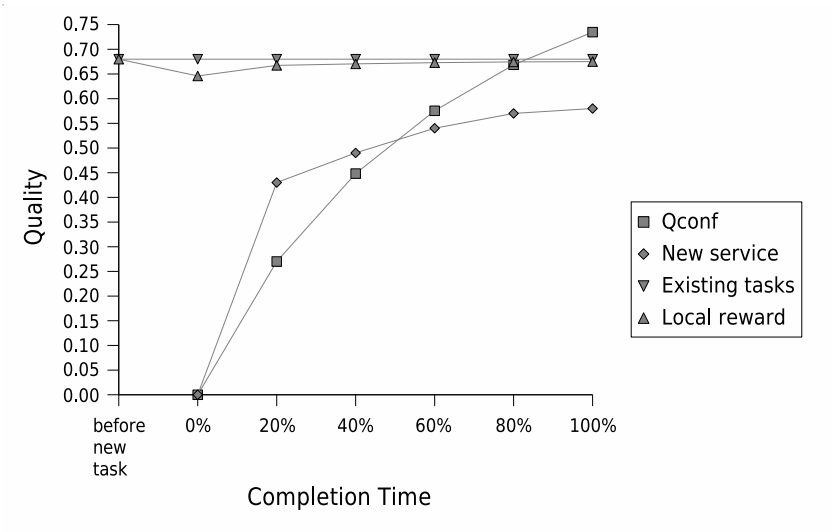

Figure 8.3: Proposal formulation: Anytime behaviour with spare resources

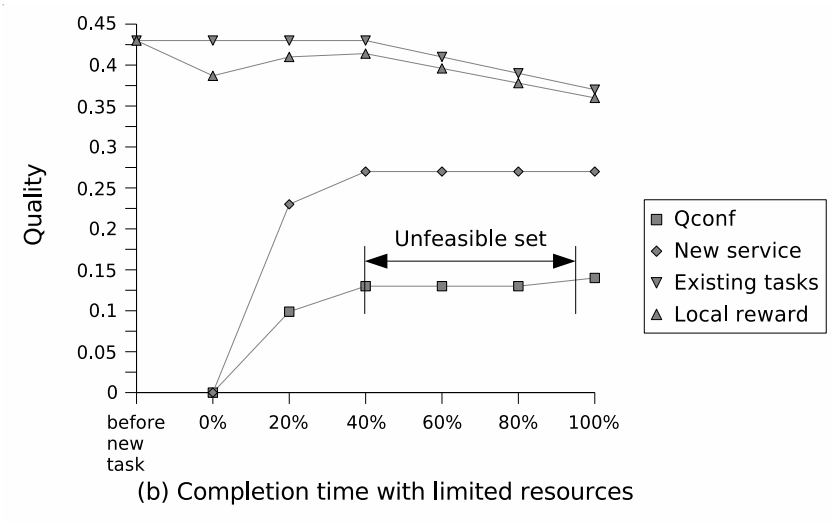

Figure 8.4: Proposal formulation: Anytime behaviour with limited resources

optimises the rate at which the quality of the current solution improves over time. With spare resources (Figure 8.3), at only 20% of the computation time, the solution's quality for the new arrived task is near 74% ± 5% of the achieved quality at completion time. When QoS degradation is needed to accommodate the new task (Figure 8.4), its service proposal achieves 85% ± 4% of its final quality at 20% of computation time.

Also note that the solution's quality, identified by Q_{conf} in both figures, quickly approaches its maximum value at an early stage of the computation.

A fourth study considered the same two scenarios of resource availability to measure the needed computation time of both the anytime and the traditional versions of the service proposal formulation algorithm to achieve their optimal solutions at completion time, as well as their first available solution.

Figure 8.5 compares the needed computation time when the average amount of resources per node is greater than the average amount of resources necessary for each service execution, while Figure 8.6 compares both versions when the average amount of resources per node is smaller than the average amount of resources necessary for each service execution, demanding QoS degradation of the previously accepted services.

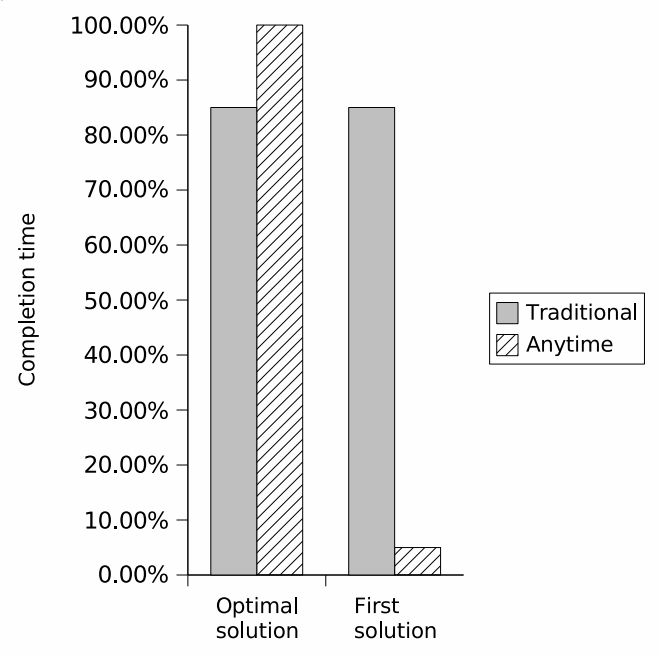

Figure 8.5: Proposal formulation: Anytime vs Traditional with spare resources

Both figures allow us to conclude that the traditional version of the service proposal formulation algorithm is also faster to achieve its optimal solution at completion time. While the anytime approach tries to quickly find an initial feasible solution by considering the worst QoS level requested by the user, the traditional version starts by selecting the user's preferred QoS level. As such, with spare resources the traditional version is faster to achieve the optimal resource allocation for the new set of tasks, while with limited resources both versions need almost the same time.

However, in both scenarios the anytime version is by far quicker to find a feasible solution. With spare resources, the anytime version needs near 5% $\pm$ 2% of its completion time to find the first feasible solution, with a quality near 10% $\pm$ 3%

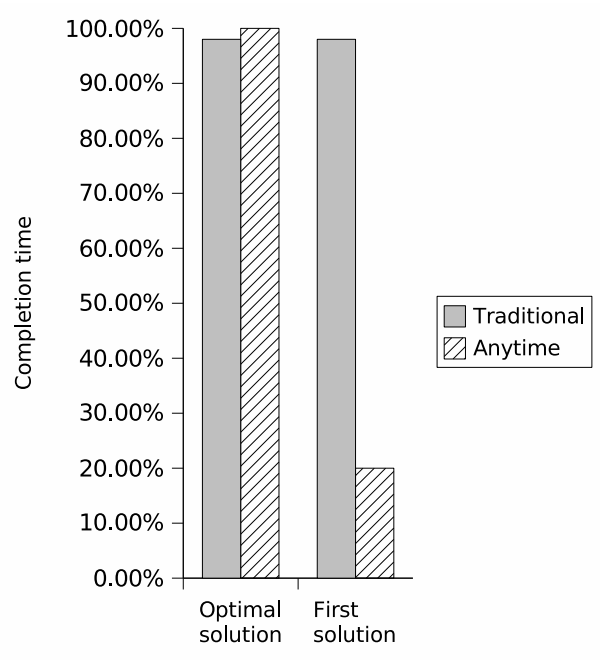

Figure 8.6: Proposal formulation: Anytime vs Traditional with limited resources

of the optimal solution. With limited resources, the anytime version takes about 20% ± 4% of its completion time to reach a feasible solution with 15% ± 3% of the optimal solution's quality.

8.3.3 Services' runtime adaptation

Whenever a system's utilisation below 60% was detected, an upgrade of previously downgraded SLAs whose stability period had already expired was done. Promised stability periods were determined by taking into consideration the observed variations in the tasks' traffic flow and correspondent resource usage, adapting the system to the observed environmental changes. The value of the smoothing factor α was optimised using the method of least squares.

The performance profile of the anytime QoS re-upgrade algorithm is plotted in Figure 8.7. At 20% of its computation time, the algorithm reaches near 60% ± 4% of its solution's quality at completion time. The increase in the solution's quality, identified by Q_{conf} in the figure is due to the increase in the tasks' reward determined by the possible upgrades. Recall that when determining the possible QoS upgrades for the previously downgraded tasks whose stability period has already expired, the QoS levels of all other tasks is kept the same. Naturally, this increase in those tasks' reward also

increases the node's local reward.

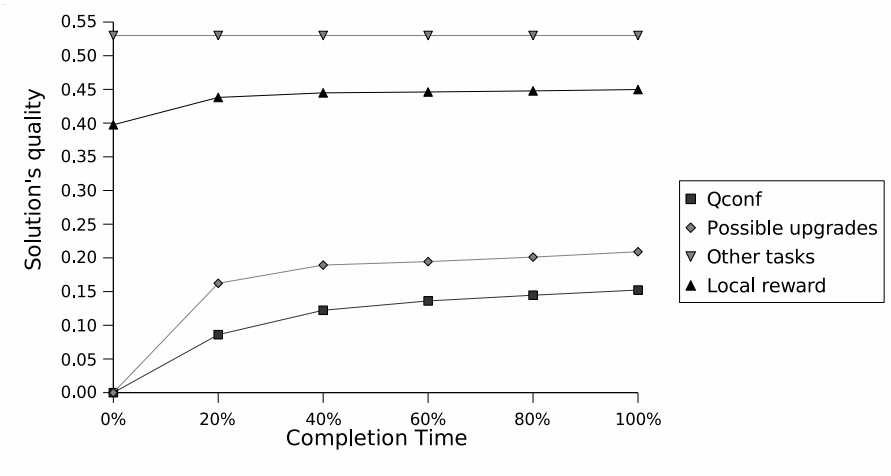

Figure 8.7: QoS re-upgrade: Anytime behaviour

The study also evaluated the users' influence on the services' adaptation behaviour. Three permanent service requests were added to the dynamic traffic randomly generated at each simulation run. All the three service requests were generated with the same random spectrum of acceptable QoS values, in the same decreasing preference order. They only differed on the users' QoS stability constraints for the minimum utility increase and stability period, $User_1 = \{0, 0s\}$, $User_2 = \{0.2, 10s\}$, $User_3 = \{0.3, 30s\}$, respectively.

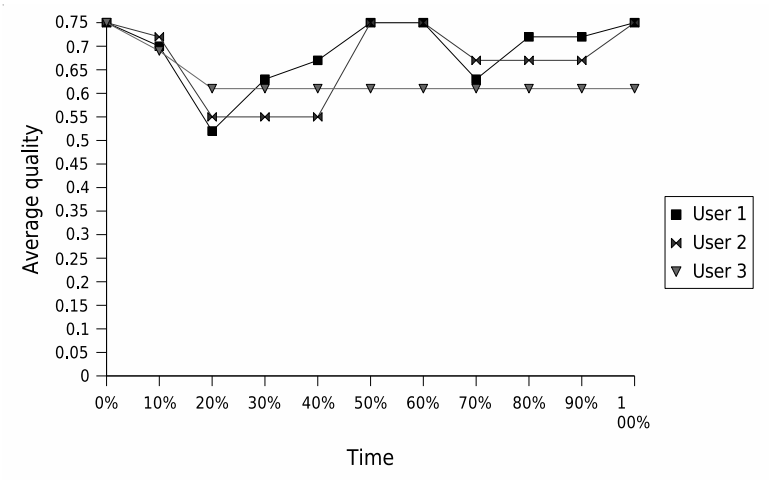

Figure 8.8: QoS re-upgrade: users' influence

The influence of personal constraints on the system's adaptation behaviour is clearly

observable in Figure 8.8. As the user's constraints for a service upgrade are harder to achieve there is less probability to change and stay in a better quality level. These results clearly demonstrate that the users' influence can be extended to the services' adaptation behaviour.

The needed computation time of both the anytime and the traditional versions of the service proposal formulation algorithm to achieve their optimal solutions at completion time, as well as their first available solution, is detailed in Figure 8.9.

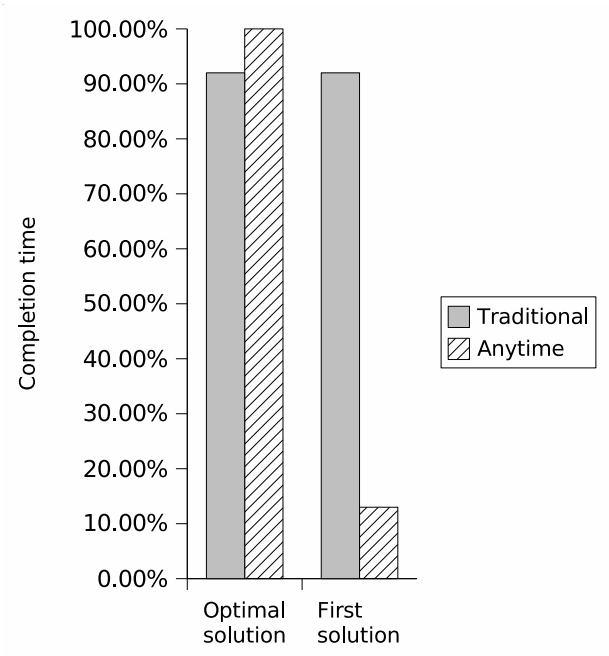

Figure 8.9: QoS re-upgrade: Anytime vs Traditional

Once again, similar conclusions can be taken. Due to the existence of spare resources, the traditional version of the QoS re-upgrade algorithm is slightly faster to complete its computation and return an optimal solution. However, the anytime version is able to almost immediately return a feasible service solution. At near 10% ± 2% of its completion time it finds the first feasible solution, with a quality near 35% ± 3% of the optimal solution achieved at completion time.

8.4 Coordinating distributed inter-dependent adaptations

The behaviour of the proposed decentralised one-step coordination model in highly dynamic scenarios was compared to a classic centralised optimal coordination model. With a centralised coordination model, all changes in the output quality of a work unit $w_{ij} \in S_i$ have to be communicated to a single entity with service-wide knowledge. Then, this central coordinator has to determine the impact of those changes in the overall coalition's QoS level and request the adaptation of the involved nodes. To evaluate the success or failure of such dependent adaptation, an adaptation request must be sequentially made along the dependency graph either until a common global service solution is found or one of the coalition member is unable to supply the new desired QoS values.

Optimallity comes from the fact that a node that cannot supply the requested coordinated QoS values is able to reply with a service counter-proposal, instead of replying with a negative feedback, as proposed in the distributed one-step coordination model. The goal is to find an optimal distributed service solution, after an unknown number of iterations among nodes. The involved nodes in the global coordinated adaptation either agree on the best possible common solution or the coordinated adaptation fails if one of the nodes is unable to find a compatible solution.

The conducted evaluation started by comparing the total number of messages that had to be exchanged among nodes when using both approaches to globally coordinate dependent autonomous self-adaptations. The average results of all simulation runs for different coalition sizes are plotted in Figure 8.10.

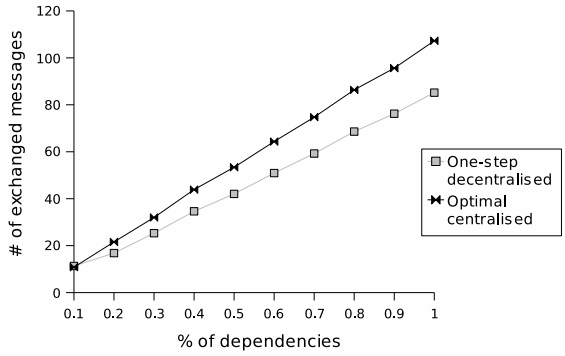

Figure 8.10: Average number of exchanged messages

As expected, both coordination approaches require more messages to be exchanged among nodes as the complexity of the service's topology increases. Nevertheless, the proposed decentralised coordination model requires around 80% of the needed number of messages required by the centralised model until all the affected coalition members become aware of the coordination request's result.

Less messages should result in a faster convergence to a global common solution. To verify the veracity of such assumption a second study measured the needed average time from the moment a node issued a coordination request until the outcome of the global adaptation process was determined. The deadline used for the anytime local QoS adaptation at each node was set to one second. At the end of the algorithm's execution, the feasibility or unfeasibility of the received coordination request was determined by the node.

The obtained results, on an Intel Core Duo T5500 at 1.66 GHz with 2 GB of RAM, are plotted in Figure 8.11.

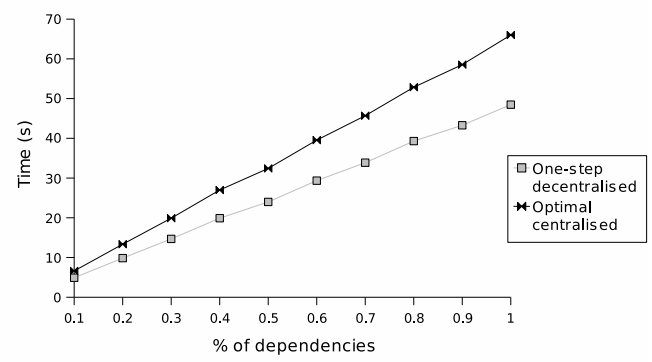

Figure 8.11: Needed time until the global adaptation result is determined

Clearly, the proposed decentralised coordination model is faster to determine the overall coordination result in all the evaluated services' topologies, needing approximately 75% of the time spent by the centralised optimal model.

Even if the proposed one-step decentralised model requires less messages and is faster than a centralised optimal model to determine a global solution it is still important to evaluate the impact of an one-step coordination model on the achieved service solution's quality. Recall that, when adopting the proposed one-step coordination algorithm, if some other dependent node in the coalition is unable to supply the new requested values no other alternative solution is tried and the global adaptation process fails. On the other hand, with the centralised optimal coordination model, a node is able to reply with a service counter-proposal whenever it is unable to coordinate with

the currently requested values. As such, it is possible that after some iterations, the node's best possible service solution can be accepted by all the dependent coalition partners as part of a global SLA. Note that such intermediate service solution would not be achieved with the proposed one-step coordination model.

The reward of each determined SLA after a successful global coordination process was evaluated by computing, for each service's QoS dimension, a weighted sum of the differences between the user's preferred quality values and the proposed values [NP05, NP06c]. The results were plotted, in Figure 8.12, by averaging the results over several independent runs of the simulation, divided into two categories: (i) when the average amount of available resources per node is greater than the average amount of resources demanded by the services being executed; and (ii) when the average amount of resources per node is smaller than the average amount of demanded resources.

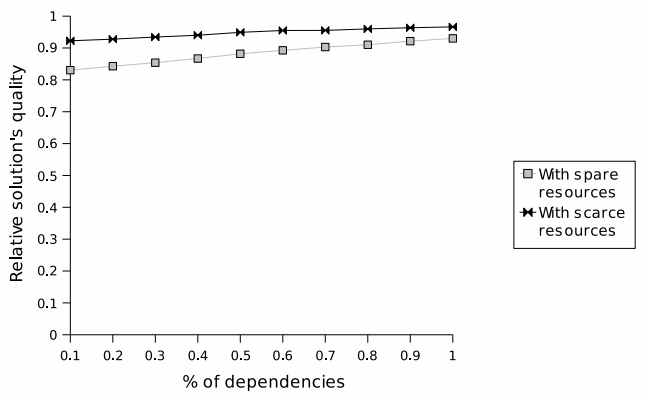

Figure 8.12: Relative solution's utility as a function of available resources

As the coalition's topology complexity increases it is clearly noticeable, in both scenarios, that a near-optimal service solution's quality is achieved when using the one-step coordination model, despite its simpler approach and faster convergence to a common solution. The achieved results can be explained by the fact that as the coalition's topology complexity increases it also increases the probability of one of the involved nodes in the global adaptation process to be unable to use more than its current level of reserved resources for a work unit $w_i \in S$. As the achieved results clearly demonstrate, such probability is even greater when the resources are scarce.

8.5 Efficiency of the proposed scheduling algorithms

The conducted experiments can be divided into two major sets. The first one evaluates the effectiveness of CSS in reducing the mean tardiness of independent periodic tasks. It starts by comparing the performance of CSS against other similar approaches considering only sets of isolated servers in Section 8.5.1, while Section 8.5.2 details the impact of allowing overloaded servers to steal inactive non-isolated capacities in the improvement of the overall system's performance.

The second set evaluates how the proposed flexible management of reserved capacities of CXP can minimise the degree of deviation from the ideal system's behaviour caused by inter-application blocking due to shared resources (Section 8.5.3) or precedence constraints (Section 8.5.4).

Each simulation replica ran until $t = 250000$, producing a large variety of inheritance and preemption situations among tasks, and was repeated several times to ensure that stable results were obtained.

8.5.1 Residual capacity reclaiming

Since the actual execution time of tasks often varies in data-, time-, or system-dependent ways, servers frequently use less computation time than they have reserved, dynamically originating residual capacity, that is, reserved but unused capacity. The efficient reclamation and redistribution of such residual capacity to tasks whose current needs exceed their reservations can significantly improve the performance of both soft-real time and best-effort tasks.

Similarly to CSS, CASH [CBS00] and BACKSLASH [LB05] also greedily assign residual capacities as early as possible to the highest priority server but propose different approaches to deal with a server's capacity exhaustion. The first conducted study evaluated the effect of those approaches in lowering the mean tardiness of independent periodic jobs. The mean tardiness was determined by $\sum_{i=0}^{n} trd_i/n$, where trd_i is the tardiness of task τ_i, and n the number of periodic tasks. For a fair comparison, only isolated servers were used with CSS.

Random workloads were created in order to evaluate the performance of each algorithms when the tasks' parameters differ in dynamic real-time scenarios. Different sets of 6 periodic servers, with varied capacities ranging from 20 to 50, and period distributions ranging from 60 to 600 were used, creating different types of load, from short to long deadlines and capacities. The execution time of each job varied in the

range $[0.7Q_i, 1.4Q_i]$ of its dedicated server's reserved capacity Q_i.

Figure 8.13 shows the performance of the three algorithms as a function of the system's load, measuring the mean tardiness of periodic tasks under random workloads for different probabilities of jobs' overload.

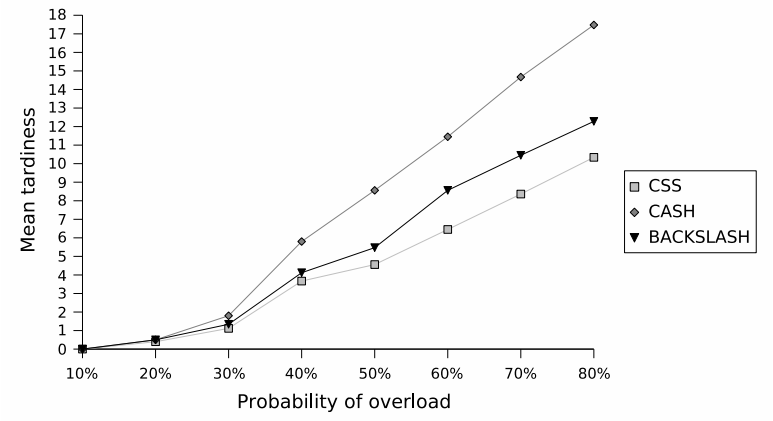

Figure 8.13: Performance in dynamic scenarios

As expected, all the algorithms perform better when there is more residual capacity available to handle overloads. Furthermore, they all behave very similarly when tasks have a lower probability (until near 30%) of exhausting their servers' reserved capacity. The behaviour of a server is determined by two parameters: (i) the server's reserved capacity, which defines the fraction of the processor allocated to the task it is serving; and (ii) the server's period, which defines the time granularity of the allocation. As such, without applying any technique to dynamically adapt the server's parameters based on the average response times of the served tasks, like the one proposed in [BB06] for example, it is clear that the system's performance will severely decrease as the probability of tasks' overloads increases.

Nevertheless, CSS outperforms the other algorithms in lowering the mean tardiness of periodic jobs with increased probabilities of jobs' overloads. Recall that CASH and BACKSLASH automatically update a server's capacity and deadline on every capacity exhaustion. As these results clearly demonstrate, allowing a task to use resources allocated to the next job of the same task may cause future jobs to miss their deadlines by larger amounts.

Even if BACKSLASH and CSS share the same concept of using original deadlines for residual capacity reclaiming, since CSS follows a hard reservation approach, a server whose capacity has been exhausted is kept active until its currently assigned

deadline. As proven by the achieved results, this approach effectively minimises the mean tardiness of periodic jobs. The reason is that a server is able to use residual capacities released after it has exhausted its capacity to advance its execution, without using capacities reserved for future jobs of the same task.

8.5.2 Allowing capacity stealing

A second study evaluated the impact of non-isolated capacity stealing on the performance of soft real-time tasks, either with short or long variations from mean execution times.

The workload consisted of a hybrid set of periodic isolated and non-isolated servers. The maximum capacity and inter-arrival times of the isolated servers were randomly generated in order to achieve a desired processor utilisation factor of $U_{isolated}$. The maximum capacity and period of the non-isolated servers were uniformly distributed in order to obtain an utilisation of $U_{non-isolated} = 1 - U_{isolated}$.

To evaluate the weight of non-isolated capacity stealing in lowering the mean tardiness of tasks, the probability of arrival of new jobs to non-isolated servers varied in the range $[0.1, 1.0]$. The mean tardiness of isolated and non-isolated jobs was measured when using both residual capacities and non-isolated capacity stealing or when only using residual capacities.

In the first simulation, periodic tasks were served by 1 non-isolated server $S_1 = (2, 10)$ and 4 isolated servers $S_2 = (3, 15)$,$S_3 = (4, 20)$,$S_4 = (5, 25)$, $S_5 = (6, 30)$, with utilisation of $U_{non-isolated} = 0.2$ and $U_{isolated} = 0.8$. The execution time of each job shortly varied in the range $[0.8Q_i, 1.2Q_i]$ of its dedicated server's reserved capacity Q_i.

The achieved results are shown in Figure 8.14. As expected, when overloaded active servers have more opportunities to steal non-isolated capacities, the obtained mean tardiness lowers accordingly. When only using residual capacities, the mean tardiness is higher as the probability of non-isolated jobs' arrival lowers, since there is less residual capacities available, released by active non-isolated servers. The experiment shows that with a low variation in the jobs' computation times, the ability to steal non-isolated capacity achieves better results, although the single use of an efficient residual capacity reclaiming mechanism is able to achieve a similar, albeit lower, performance.

Furthermore, Figure 8.14 also shows that the performance of non-isolated servers is worse than the achieved performance of isolated servers. Two reasons explain this behaviour. The first one is that when a new job arrives for a inactive non-isolated

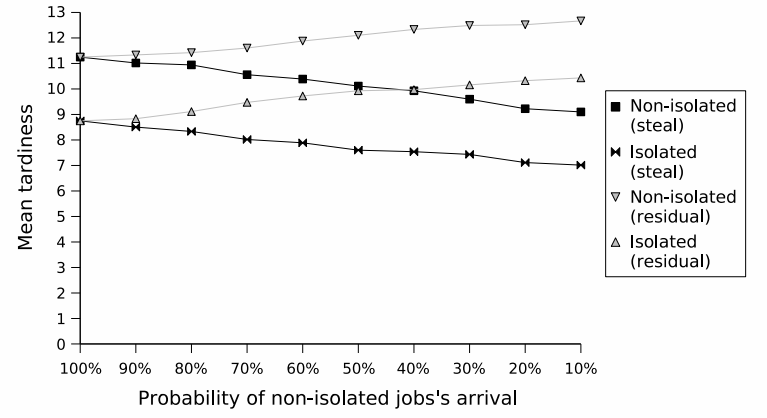

Figure 8.14: Small variation in execution times

server, some of its reserved capacity might have been stolen by a needed active overloaded server. As such, if the now active non-isolated server cannot reclaim any available residual capacity, the job must be executed with less capacity than expected, probably resulting in a deadline miss. The second one is that there is a big difference on the performance of a server for different configurations of Q_i and T_i, even if they result in the same server utilisation [BB02]. It is well known that the higher the priority the smaller the capacity available, since there is a tradeoff between capacity size and interference. A server with parameters $(2Q_i, 2T_i)$ has the same utilisation but a higher probability of using residual capacities and steal inactive non-isolated time due to the increased period.

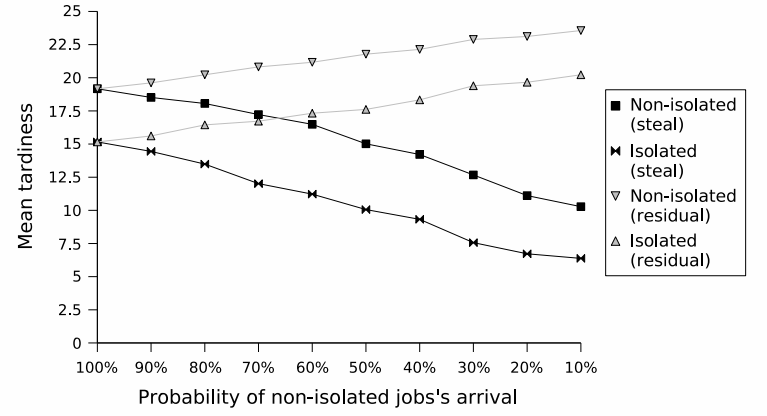

Figure 8.15: Large variation in execution times

The second simulation has been generated with the same characteristics of the first one, except that a greater variance of jobs' execution time was introduced, ranging

from $[0.6Q_i, 1.8Q_i]$ of the dedicated server's reserved capacity Q_i. Note that in this experiment the average value of the jobs' execution requirements is greater than the reserved capacity of their servers, necessarily leading to a greater tardiness. Figure 8.15 clearly shows a perceptibly improved system's performance when it is possible to steal inactive non-isolated capacities in the presence of a large variation in jobs' computation times. One can conclude that, with CSS, severe overloads can be efficiently handled through a residual capacity reclaiming and non-isolated capacity stealing approach, reducing the mean tardiness of periodic jobs.

8.5.3 Sharing resources among tasks

The first conducted study compared the cumulative capacity that was consumed by the shortest period (SP) and longest period (LP) tasks of a randomly generated task set when tasks share resources to the amount of capacity that would be consumed if the same set of tasks did not shared any resource. The cumulated capacities consumed by the SP and LP tasks were recorded every 200 time ticks and the mean values of all generated samples plotted in Figures 8.16 and 8.17, respectively.

Different sets of 5 tasks were randomly generated, with varied execution requirements ranging from 20 to 60 units, and period distributions ranging from 100 to 300 time units, always ensuring a system's utilisation $U \leq 1$. An isolated server was assigned to each task, with a reserved capacity Q_i equal to the task's execution requirements and period T_i equal to the task's period. The execution requirements of each job were always equal to the reserved capacity of its dedicated server and all jobs accessed the shared resource R during all their executions, with a new job being released immediately after a task has completed its current job.

The achieved results show that with BWI, and due to blocking, while higher priority tasks can consume less than their initial allocations, tasks with longer deadlines can consume more than their reserved capacities since BWI is affected by the absence of a compensation mechanism. In contrast, the efficient capacity exchange mechanism of CXP ensures that both tasks are able to get their allocated capacities even when accessing shared resources thus providing a better fairness than BWI and confirming the conclusions drawn from the examples in Section 7.3.

A second study compared the efficiency of the studied protocols BWI, BWE, CFA and CXP in lowering the mean tardiness of a set of periodic jobs with variable execution times in highly dynamic scenarios. At each simulation run, a random number of servers with a system's utilisation up to 70% contended for the system's resources

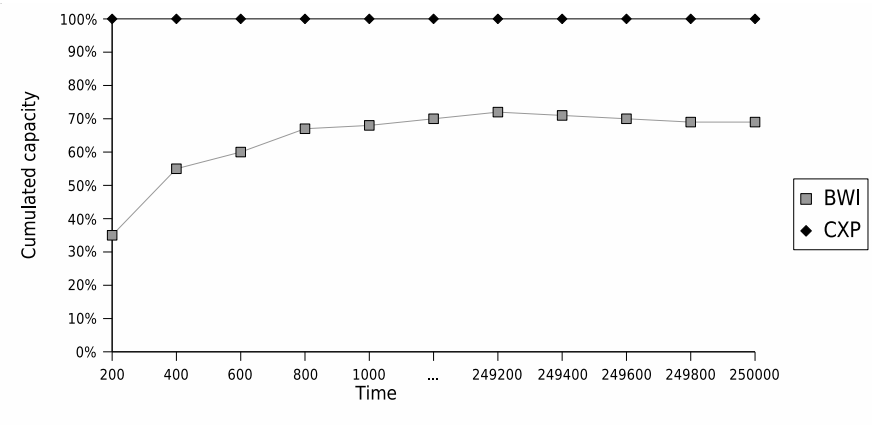

Figure 8.16: Capacity consumed by the SP task

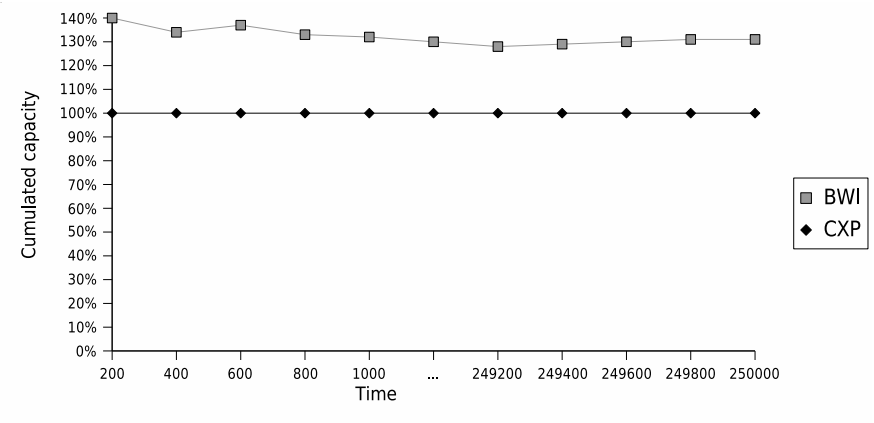

Figure 8.17: Capacity consumed by the LP task

with a dynamic traffic that demanded up to 30% of the system's capacity. Resource sharing protocols that require a prior knowledge of the maximum resource usage time for each task such as the Priority Ceiling Protocol, the Dynamic Priority Ceiling, or the Stack Resource Policy were not considered in the studies since they cannot be directly applied to open real-time systems.

All servers were generated with varied reserved capacities Q_i ranging from 15 to 50 units of execution and period distributions ranging from 50 to 500 time units, creating different types of load, from short to long deadlines and capacities. Tasks arrived at randomly generated times and remained in the system for a variable period of time with each job having an execution time in the range $[0.8Q_i, 1.2Q_i]$ of its dedicated server's reserved capacity Q_i, originating both overloads and residual capacities due to early completions. There were 6 resources, whose access and duration of use was randomly distributed by the servers, creating direct and transitive blocking situations and distinct resource groups. For a fair comparison, only isolated servers were used in CXP.

Figure 8.18 illustrates the performance of the evaluated protocols as a function of the system's load, measuring the mean tardiness of periodic tasks under random workloads for different probabilities of jobs' overload.

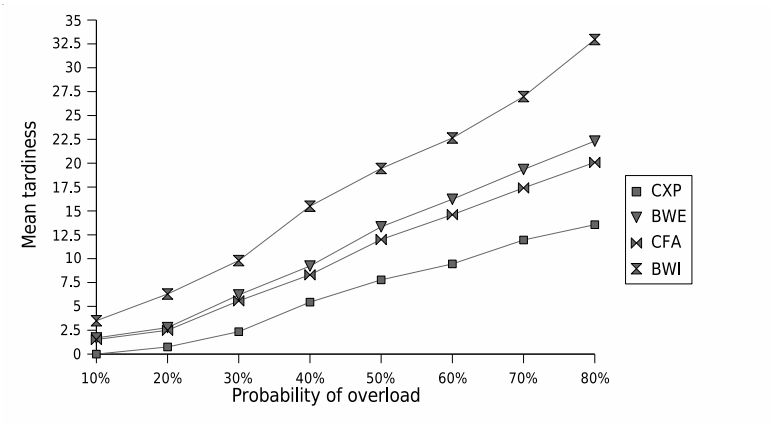

Figure 8.18: Performance in dynamic scenarios

As expected, the achieved results clearly justify the use of a capacity exchange mechanism to minimise the impact of blocking on the system's performance. Without any compensation for the extra work on blocked servers, BWI obtains the poorest result. Recall that with BWI, a blocked task is only able to use the remaining capacity of its dedicated server, if any.

BWE and CFA achieve similar performances when handling tasks with variable execution times. Both algorithms are unable to reclaim residual capacities originated by early completions, wasting available resources to handle overloads and minimise the number of deadline misses. Also, both algorithms immediately recharge a server's capacity and extend its deadline at every capacity exhaustion, allowing a task to use resources allocated to a future job, contributing for future jobs of that task to miss their deadlines by larger amounts.

On the other hand, by reclaiming as much extra capacity as possible, CXP outperforms BWE and CFA in lowering the mean tardiness of periodic tasks in highly dynamic scenarios. CXP not only exchanges capacities between all active servers, not restricting capacity exchange to the same resource group, but it also reclaims all the available residual capacity to handle overloads of soft real-time tasks.

Furthermore, these better results in highly dynamic scenarios were achieved with a less complex approach to exchange reserved capacities among servers. Figure 8.19 illustrates the average overhead introduced by the optimisations of BWE, CFA, and CXP in terms of the needed scheduling time and memory consumption during the previous study, using the base BWI protocol as a reference.

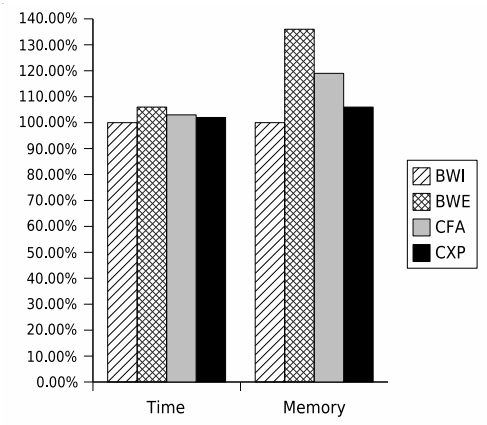

Figure 8.19: Overhead using BWI as reference

As expected, the optimisations performed by BWE, CFA, and CXP introduce some overhead when compared against BWI in terms of needed time and memory. Although all the three algorithms need only slightly more time than BWI to determine which capacity is going to be accounted by the currently executing server, they substantially differ in terms of storage information demands. BWE requires a global $n * n$ matrix to record the amount of capacity that must be exchanged between servers and an extra list at each server to keep track of available capacities. CFA enhances BWI by

adding a new task queue to each server and one extra variable for each contracted debt between servers S_i and S_j. On the other hand, CXP focuses on minimising the cost of blocking by exchanging reserved capacities as early, and not necessarily as fairly, as possible. As such, it does not not account the amount of borrowed capacity on each server neither manages individual resource groups.

8.5.4 Imposing precedence constraints among tasks

Another study compared the time and memory needed by CXP to schedule the same task set with and without precedence constraints among its tasks. 10000 tasks sets were randomly generated, with different system's utilisation in the range $[0.6, 1.0]$. For each task set, a random set of precedence constraints consistent with the tasks' deadlines was determined. Each job had random execution requirements in the range $[0.7Q_i, 1.3Q_i]$ of its dedicated server's reserved capacity Q_i.

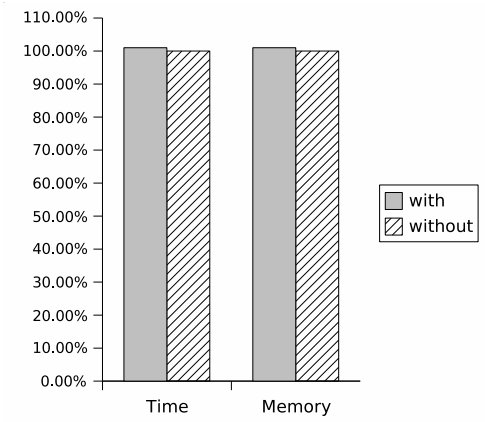

Figure 8.20: Overhead of handling precedence constraints

The achieved results, plotted in Figure 8.20, allow us to conclude that CXP is able to efficiently handle precedence constraints among tasks whose exact behaviour is not known beforehand without any significant overhead. Recall that precedence constraints are handled by CXP as an access to a shared resource and the proposed residual capacity reclaiming policy already checks the current state of earlier deadline servers, since residual capacities are consumed before the server's reserved capacity.

8.6 Summary

This chapter evaluated, through extensive simulations, the behaviour and overhead of the algorithms proposed in this thesis when operating in highly dynamic open real-time scenarios.

After the formal analysis of the desirable properties of anytime algorithms discussed in Chapter 4, the empirical evaluation detailed in Section 8.3 further strenghts the pratical usefulness of the proposed anytime algorithms. Note that the solution's quality measure of the three algorithms is a non-decreasing function of time. Only a better service proposal for each specific task under negotiation updates the currently found solution, increasing its quality. Also, the improvement in the solution's quality is larger at the early stages of the computation and diminishes over time. All the three algorithms quickly determine a service solution whose quality is expected to be sufficiently close to their optimal solution's quality, determined at completion time.

Section 8.4 evaluated the proposed one-step decentralised coordination of autonomous dependent adaptations of resource constrained devices. As the achieved results demonstrate, the proposed coordination model has a reduced overhead and enables a faster convergence to a new global service solution, whenever the needed downgrade or desired upgrade in one coalition member has an impact on or depends on the quality of the inputs sent by other work units being executed on other coalition members.

The results reported in Section 8.5 clearly demonstrate that the proposed Capacity Sharing and Stealing (CSS) approach for independent task sets is able to efficiently reclaim residual capacities originated by earlier completions and steal reserved unused capacities from inactive non-isolated servers, effectively reducing the mean tardiness of soft real-time tasks. It is clear that the exact performance depends upon the ratio of residual capacity donating and residual capacity consuming tasks, but in general CSS outperforms the other evaluated algorithms.

When considering inter-depedent task sets, the achieved results clearly justify the use of a capacity exchange mechanism that reclaims as much capacity as possible and does not restrict itself to exchange capacities only within a resource sharing group. The proposed Capacity Exchange Protocol (CXP) achieves a better system's performance when compared against other available solutions and has a lower overhead.

Chapter 9

Conclusion

This thesis provides support for adaptive cooperative coalitions of possibly inter-dependent nodes, able to autonomously organise, regulate and optimise themselves without the intervention of a user or any other central entity, even when services exhibit unrestricted QoS inter-dependencies. This chapter resumes its most relevant contributions and highlights some lines of future work.

9.1 Introduction

As the complexity of open embedded real-time systems increases, driven by the need to boost their capabilities and scope, the ability to support predictable, reliable Quality of Service (QoS) must keep pace. This calls for a different and more flexible approach than those typically used today for building fixed-purpose real-time systems, since the set of applications to be executed and their aggregate resource and timing requirements are unknown until runtime, implying that accurate optimisation models are then difficult to obtain and quickly become outdated.

The challenge is how to efficiently execute applications in these new open real-time systems while meeting non-functional requirements arising from the operating environment, the users, and applications. This thesis advocates that the complex demands of such systems are adequately handled through a cooperative decentralised model, supported by anytime QoS optimisation algorithms and effective flexible scheduling mechanisms.

In the previous chapters, we proposed and evaluated the mechanisms to achieve

this goal. The CooperatES framework allows services to be executed by temporary coalitions of nodes whenever a particular set of user-imposed QoS constraints cannot be satisfyingly answered by a single node. Users encode their own relative importance of the different QoS parameters for each service they want to execute and the framework uses this information to determine the distributed resource allocation that maximises the satisfaction of those constraints and minimises the impact on the current QoS levels of previously accepted tasks.

Thanks to the anytime nature of the proposed QoS optimisation approach, it is possible to interrupt the optimisation process, done according to each user's specific QoS and stability preferences, at any point in its execution and still be able to obtain a service solution and a measure of its quality, which is expected to improve as the run time of the algorithms increases. The binary notion of correctness associated with traditional QoS optimisation algorithms is then replaced by a set of quality measured outputs.

Furthermore, in order to reduce the needed interactions among nodes until a collective adaptation behaviour is determined, this thesis proposes an one-step decentralised coordination model based on an effective feedback mechanism. Positive feedback is used to reinforce the selection of the new desired global service solution, while negative feedback discourages nodes to act in a greedy fashion as this adversely impacts on the provided service levels at neighbouring nodes.

In addition, a new scheduling approach is proposed to handle the dynamic changes of services' requirements in a predictable fashion, enforcing timing constraints with a certain degree of flexibility, aiming to achieve the desired tradeoff between predictable performance and an efficient use of resources. CSS is a dynamic server-based scheduler that supports the coexistence of guaranteed and non-guaranteed bandwidth servers to efficiently handle soft-tasks' overloads by making additional capacity available from two sources: (i) residual capacity allocated but unused when jobs complete in less than their budgeted execution time; (ii) stealing capacity from inactive non-isolated servers used to schedule the anytime algorithms devoted to the framework's management.

Another algorithm able to handle dependent tasks sets which share access to some of the system's resources and exhibit precedence constraints is proposed. CXP merges the benefits of CSS with the concept of bandwidth inheritance to allow a task to be executed on more than its dedicated server, efficiently exchanging capacities among servers and reducing the undesirable effects caused by inter-application blocking.

The next sections summarise the most relevant contributions of this thesis to the development of cooperative open real-time systems and outline areas for future research.

9.2 General conclusions

Throughout the previous chapters of this thesis we have shown that:

- A semantically rich QoS specification interface for multidimensional QoS provisioning allows users to define fine-grained service requests that are later used to (i) dynamically select the cooperative set of nodes that maximises the satisfaction of those service constraints; and (ii) adapt the services' QoS configuration during runtime.
- An anytime adaptive QoS management that quickly achieves a reasonable solution's quality is a powerful approach to ensure a timely answer to events, despite the imprecision, uncertainty, and complexity of open real-time environments, where actual resource needs are only known at runtime and tasks may even exhibit unrestricted QoS inter-dependency relations.
- A simple and effective feedback mechanism can be used to reduce the complexity of the needed interactions among nodes until a collective adaptation behaviour is determined, whenever the autonomous self-adaptations to the changing environmental conditions have an impact on other coalition members.
- Unused reserved capacities in server-based schedulers can be more efficiently used to meet deadlines of tasks whose resource usage exceeds their reservations and minimise the degree of deviation from the ideal system's behaviour, caused by inter-application blocking due to shared resources or precedence constraints.

9.3 Summary of the main contributions

This section presents a summary of the main contributions of this thesis.

The fundamental basis of a QoS-aware cooperative framework

To tackle the increasing demand for performance and resources in open embedded environments, this thesis has proposed the main architecture of the CooperatES framework, a cooperative computing approach based on the concept of dynamically formed coalitions of neighbour nodes. By redistributing the computational load across a set of nodes, a cooperative environment enables the execution of far more complex and resource-demanding services that otherwise would be able to be executed on a standalone basis.

Utility-based resource allocation and adaptation policies which take into consideration the increasing demand for customisable service provisioning, tailored to each user's specific QoS preferences and needs, was explored. The proposed QoS negotiation approach goes beyond the basic QoS scheme of delivering service in a prioritised fashion. Instead, it allows users and applications to specify, through a QoS specification interface semantically rich both in terms of expressiveness and customisation, the QoS dimensions subject to negotiation, their attributes and the quality constraints in terms of possible values for each attribute, as well as inter-dependency relations between some of those QoS parameters. Then, the coalition formation and service proposal formulation algorithms fully explore the QoS tradeoffs that maximise the satisfaction of the QoS constraints associated with new services and minimise the impact on the global QoS caused by a new service's arrival.

Particular attention was devoted in also maximising the users' influence on their services' adaptation behaviour at runtime, proposing that the dynamic QoS arbitration among competing services should be done under the control of the user. While some users may prefer to always get the best possible instantaneous QoS, independently of the reconfiguration rate of their requested services, others may find that frequent QoS reconfigurations are undesirable. This suggests that while a resource constrained device may not be able to avoid a downgrade of the currently provided QoS level of some services in order to accommodate a new service with a higher utility if its granted stability period has already expired, upgrades to a higher QoS level can and should be controlled by each user's stability requirements. Upgrades of currently provided QoS levels are subject to a comparison against each user's stability requirements, namely a minimum utility increment and minimum stability period. Promised stability periods are periodically updated in response to fluctuations in the tasks' traffic flow, relating observations of past and present environmental conditions. The achieved results clearly demonstrate that such influence can be achieved.

An anytime QoS optimisation and adaptation approach

The increased complexity and dynamism of open real-time environments may prevent the possibility of computing both optimal local and global resource allocations within a useful and bounded time. This is true for many soft real-time applications, where it may be preferable to have approximate results of a poorer but acceptable quality delivered on time to late results with the desirable optimal quality. Anytime algorithms have shown themselves to be particularly appropriate in such settings, as they usually provide an initial, possibly highly sub-optimal, solution very quickly and then concentrate on improving this solution until the time available for planning runs

out. Nevertheless, there as been relatively little interaction between QoS management and anytime algorithms. QoS management research has been concentrated on finding single optimal, or with a fixed sub-optimality bound, solutions.

This thesis has reformulated the distributed resource allocation problem for sets of both independent and dependent task sets as a heuristic-based anytime optimisation problem in which there is 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. The achieved results clearly demonstrate that proposed anytime algorithms are able to quickly find a good initial solution and effectively optimise the rate at which the quality of the current solution improves at each iteration of the algorithms, with an overhead that can be considered negligible when compared to the introduced benefits.

Support for unrestricted QoS inter-dependencies among tasks

While runtime adaptation is widely recognised as valuable, adaptations in most existing systems are limited to changing independent execution parameters. However, embedded applications increasingly consist of interacting components that may exhibit unrestricted QoS inter-dependencies among them. This thesis provided support for dependent runtime adaptations that span multiple hosts and multiple components in open distributed systems.

Whenever the outputted QoS of some task depends not only on the amount and type of used resources but also on the quality of the received inputs sent by other tasks, the QoS negotiation process was extended to ensure that a source task provides a QoS which is acceptable to all consumer tasks and lies within the QoS range supported by the source task. A service's feasible QoS level was then defined as the set of compatible QoS regions provided by all the dependent components that compose the service.

With inter-dependencies among local tasks, it was ensured that a valid service solution was available at any time by tracking QoS dependencies and propagating the performed changes to all the affected attributes, at each iteration of the proposed anytime QoS optimisation and adaptation algorithms.

With inter-dependencies that span multiple hosts, runtime autonomous adaptations were coordinated in order to maintain the service's correctness. Note that a lack of coordination among nodes in a cooperative distributed system can then lead to interference between the different nodes' self-management behaviour, conflicts over shared resources, sub-optimal system performance and hysteresis effects. This thesis has proposed an one-step decentralised coordination model, based on an effective

feedback mechanism to reduce the complexity of the needed interactions until a collective adaptation behaviour is determined. Positive feedback was used to reinforce the selection of the new desired global service solution, while negative feedback discouraged nodes to act in a greedy fashion as this adversely impacts on the provided service levels at neighbour nodes.

The achieved results clearly demonstrate that, while achieving similar coordinated global QoS levels, the proposed one-step decentralised coordination model is faster and requires few messages to be exchanged among nodes than other possible approaches for coordinating autonomous adaptations of nodes in cooperative environments.

Novel scheduling algorithms for open systems

Predictability in open real-time environments is strictly related to the capacity of controlling the incoming workload, preventing abrupt and unpredictable performance degradations. As such, it is necessary to prevent a service that needs more than the expected resource reservations to introduce unbounded delays on other services' execution, jeopardising their performance. On the other hand, unused reserved capacities, originated whenever a task needs less than its budgeted execution time, should be donated to overloaded servers, in order to improve the response times of soft real-time tasks, particularly in systems where the needed computation time is highly variable and data dependent.

Based upon a careful study of the ways in which unused reserved capacities can be more efficiently used to meet deadlines of tasks whose resource usage exceeds their reservations, this thesis has presented the Capacity Sharing and Stealing (CSS) scheduler. CSS considers the coexistence of the traditional *isolated* servers with a novel *non-isolated* type of servers, combining an efficient reclamation of residual capacities with a controlled isolation loss. As such, it handles overloads with additional capacity available from two sources: (i) by greedily reclaiming unused allocated capacity when jobs complete in less than their budgeted execution time; and (ii) by stealing allocated capacities to inactive non-isolated servers used to schedule aperiodic best-effort jobs.

By giving priority to the overload control of guaranteed services, the achieved results demonstrate that CSS has a better performance than other available scheduling solutions, particularly when tasks' computation times have a large variance.

The effectiveness and reduced complexity of CSS in managing unused reserved capacities, without any previous complete knowledge about the tasks' runtime behaviour, was used as the basis of a more powerful scheduler, able to handle dependent tasks sets which share access to some of the system's resources and exhibit precedence

constraints. Rather than trying to account borrowed capacities and exchanging them later in the exact same amount, the proposed Capacity Exchange Protocol (CXP) focus on greedily exchanging extra capacities as early, and not necessarily as fairly, as possible and introduces a novel approach to integrate precedence constraints into the task model.

The achieved results clearly justify the use of a capacity exchange mechanism that reclaims as much capacity as possible and does not restrict itself to exchange capacities only within a resource sharing group. It was proven that CXP achieves a better system's performance when compared to other available solutions and has a lower overhead.

9.4 Future research directions

In this thesis, most of the effort has been spent on the theoretical formalisation and evaluation of adaptive QoS management and scheduling policies for cooperative open real-time systems. The next important step is to continue the development of the CooperatES framework and subject the proposed algorithms to actual workloads from real QoS-aware applications. This means that some existing applications need to be modified, or new adaptive QoS-aware applications that can be distributed across a set of nodes need to be developed.

Given the widespread use of embedded systems and the trend to rely more and more on them, increased demands for dependability are expected to arise. Therefore, the issue of dependability should play a very important role in future developments of the CooperatES framework in order to exhibit crucial attributes such as availability, reliability, or safety. Particular attention should be devoted into devising what will be the future applications' needs and what kind of support and technologies of dependable embedded systems must be provided.

While both CSS and CXP focus on temporal aspects and constraints, they can also include non-temporal objectives. With the current trends towards higher integration and embedding processors in battery-powered devices, energy consumption becomes an increasingly important issue. Dynamic Power Management (DPM) and dynamic Voltage Scaling (DVS) have both proven to be highly effective techniques for reducing power dissipation. DPM refers to a selective shut-off of idle system components, while DVS slows down underutilised resources and decreases their operating voltages. From this perspective, the goal of an energy-aware version of the proposed scheduling

algorithms should not be only to select the task to be scheduled and which reserved capacity to use, but also the CPU's operating frequency, in order to minimise the consumed energy but without jeopardising the schedulability of real-time tasks.

The proposed CooperatES framework assumes nodes are willing to cooperate, enabling the formation of coalitions of nodes that share computation power for free. In a business environment, mechanisms are needed to provide incentive for both consumers and resource owners for being part of a shared environment. There are various economic models for setting the price of services based on suply-and-demand and their value to the user. It would be interesting to integrate them into the CooperatES framework and operate in environments populated by economically motivated nodes.

Bibliography

[AAS00] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. Qos negotiation in real-time systems and its application to automated flight control. *IEEE Transactions on Computers, Best of RTAS '97 Special Issue*, 49(11):1170–1183, November 2000.

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE RTSS*, page 4, Madrid, Spain, December 1998.

[Abe98] Luca Abeni. Server mechanisms for multimedia applications. Technical report, Scuola Superiore S. Anna, 1998.

[ACH+01] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a sentient computing system. *IEE Computer*, 34(8):50–56, August 2001.

[ACS03] M. Agrawal, D. Cofer, and T. Samad. Real-time adaptive resource management for advanced avionics. *IEEE Control Systems Magazine*, 23(1):6–86, February 2003.

[AL97] Emile Aarts and Jan Karel Lenstra, editors. *Local search in combinatorial optimization*. John Wiley & Sons, 1997.

[ART04] ARTIST (IST-2001-34820). *Selected topics in Embedded Systems Design: Roadmaps for Research. Part III - Adaptive Real-Time Systems for Quality of Service Management*, May 2004. Available at http://www.artist-embedded.org/.

[AS98] T. Abdelzaher and K.G. Shin. End-host architecture for qos-adaptive communication. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, pages 121–130, Denver, Colorado, USA, June 1998.

[AS99] T.F. Abdelzaher and K.G. Shin. Qos provisioning with q contracts in web and multimedia servers. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 43–53, Phoenix, Arizona, USA, December 1999.

[Bak90] Theodore P. Baker. A stack-based resource allocation policy for realtime processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 191–200, Lake Buena Vista, Florida, USA, December 1990.

[Bak01] D. E. Bakken. *Middleware*. Kluwer Academic Press, 2001.

[Bar06] Sanjoy K. Baruah. Resource sharing in edf-scheduled systems: A closer look. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 379–387, Rio de Janeiro,Brazil, December 2006.

[BB02] Guillem Bernat and Alan Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Systems*, 22(1-2):49–75, 2002.

[BB04] R. Bhattacharya and G. J. Balas. Anytime control algorithm: Model reduction approach. *Journal of Guidance, Control, and Dynamics*, 27(5):767–776, October 2004.

[BB06] Giorgio Buttazzo and Enrico Bini. Optimal dimensioning of a constant bandwidth server. In *Proceedings of the 27th IEE International Real-Time Systems Symposium*, pages 169–177, Rio de Janeiro, Brasil, December 2006.

[BBB04] G. Bernat, I. Broster, and A. Burns. Rewriting history to exploit gain time. In *Proceedings of the 25th IEEE RTSS*, pages 328–225, December 2004.

[BCA+01] G. S. Blair, G. Coulson, A. Andersen, M. Clarke, F. M. Costa, R. Moreira H. A. Duran, N. Paralavantzas, and K. B. Saikoski. The design and implementation of open orb version 2. *IEEE Distributed Systems Online Journal*, 2(6), June 2001.

[BCD98] G. Blair, G. Coulson, and N. Davies. Adaptive middleware for mobile multimedia applications. In *Proceedings of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 259–273, St. Louis, Missouri, USA, May 1998.

[BCHS01] P.G. Bridges, Wen-Ke Chen, M.A. Hiltunen, and R.D. Schlichting. Supporting coordinated adaptation in networked systems. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 162, Oberbayern, Germany, May 2001.

[BCRP98] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 191–206, The Lake District, England, September 1998.

[BCS94] R. Branden, D. Clark, and S. Shenker. IETF RFC 1633: Integrated services in the internet architecture: an overview, 1994.

[BDK+03] Craig Boutilier, Rajarshi Das, Jeffrey O. Kephart, Gerald Tesauro, and William E. Walsh. Cooperative negotiation in autonomic systems using incremental utility elicitation. In *In Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*, pages 89–97, Acapulco,Mexico, August 2003.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BJ90] George Edward Pelham Box and Gwilym Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[Bla77] Jacek Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proceedings of the International Workshop on Modelling and Performance Evaluation of Computer Systems*, pages 57–65, Ispra,Italy, October 1977.

[BMB+00] J. Bacon, K. Moody, J. Bates, Chaoying Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *IEE Computer*, 33(3):68–76, March 2000.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, March 1984.

[BNBM98] S. Brandt, G. Nutt, T. Berk, and J. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 307–317, December 1998.

[BP04] Ricardo Barbosa and Luís Miguel Pinho. Mechanisms for reflection-based monitoring of real-time systems. In *Work-In-Progress Session of the 16th ECRTS*, June 2004.

[Bro63] Robert Goodell Brown. *Smoothing, forecasting and prediction of discrete time series*. Prentice-Hall, Englewood Cliffs, NJ, 1963.

[BSLH05] Henrike Berthold, Sven Schmidt, Wolfgang Lehner, and Claude-Joachim Hamann. Integrated resource management for data stream systems. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 555–562. ACM Press, 2005.

[Bur03] M. Burgess. On the theory of system administration. *Science of Computer Programming*, 49:1, 2003.

[CBCP01] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. *Lecture Notes in Computer Science*, 2218:160–178, 2001.

[CBS00] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Proceedings of 21th IEEE RTSS*, pages 295–304, Orlando, Florida, 2000.

[CBT05] Marco Caccamo, Giorgio C. Buttazzo, and Deepu C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, February 2005.

[CKK+04] 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 Transactions on Parallel and Distributed Systems*, 15(9):795–809, 2004.

[CL90] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: a concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, 1990.

[Cora] Microsoft Corporation. Distributed component object model. Available at `http://msdn.microsoft.com/en-us/library/ms809340.aspx`.

[Corb] Microsoft Corporation. Microsoft .net framework. Available at `http://msdn.microsoft.com/en-us/netframework/default.aspx`.

[CP03] Antoine Colin and Stefan M. Petters. Experimental evaluation of code properties for wcet analysis. In *Proceedings of the 24th IEEE RTSS*, pages 190–199, December 2003.

[CPM+04] Tommaso Cucinotta, Luigi Palopoli, Luca Marzario, Giuseppe Lipari, and Luca Abeni. Adaptive reservations in a linux environment. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 238–245, Toronto, Canada, May 2004.

[Crn02] Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[CS01] Marco Caccamo and Lui Sha. Aperiodic servers with resource constraints. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 161–170, London, UK, December 2001.

[CSB90] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[CSKO02] A. Corsaro, D. Schmidt, R. Klefstad, and C. O'Ryan. Virtual component - a design pattern for memory constrained embedded applications. In *Proceedings of the 9th Conference on Pattern Language of Programs*, Monticello, Illinois, September 2002.

[CSZ92] D. Clark, S. Shenker, and L. Zhangn. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proceedings of the SIGCOMM'92 Symposium on Communications Architectures and Protocols*, pages 14–26, October 1992.

[CT94] C.L. Compton and D.L. Tennenhouse. Collaborative load shedding for media-based applications. *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 496–501, May 1994.

[Dav93] R. I. Davis. Approximate slack stealing algorithms for fixed priority preemptive systems. Technical report, Department of Computer Science, University of York, November 1993.

[DB88] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 49–54, 1988.

[DC99] Marco Dorigo and Gianni Di Caro. The ant colony optimization meta-heuristic. *New ideas in optimization*, pages 11–32, 1999.

[DC07] Ivana Dusparic and Vinny Cahill. Research issues in multiple policy optimization using collaborative reinforcement learning. In *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, page 18, Washington, DC, USA, 2007. IEEE Computer Society.

[DdIA] Politecnico di Milano Dipartimento di Ingegneria Aerospaziale. Realtime application interface for linux. Available at `http://www.rtai.org/`.

[DH08] Jim Dowling and Seif Haridi. *Decentralized Reinforcement Learning for the Online Optimization of Distributed Systems*, chapter in Reinforcement Learning: Theory and Applications, pages 142–167. I-Tech Education and Publishing, Vienna, Austria, 2008.

[DLS97] Z. Deng, J.W.-S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, pages 191–199, Toledo, Spain, June 1997.

[DTB93] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings of the 14th RTSS*, pages 222–231, 1993.

[DWH03] T. De Wolf and T. Holvoet. Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control. *Proceedings of the IEEE International Conference on Industrial Informatics*, pages 470–479, August 2003.

[EEGL03] Viktor S. Wold Eide, Frank Eliassen, Ole-Christoffer Granmo, and Olav Lysne. Supporting timeliness and accuracy in distributed real-time content-based video analysis. In *Proceedings of the 11th ACM international conference on Multimedia*, pages 21–32. ACM Press, 2003.

[EM] Valerie J. Easton and John H. McColl. Statistics glosarry. Available at `http://www.stats.gla.ac.uk/steps/glossary/index.html`.

[Emb] Embedded.com. The rtos buyers guide. Available at `http://www.embedded.com/`.

[Emm00] W. Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the Conference on the future of software engineering*, pages 117–129, Limerick, Ireland, June 2000.

[Fan95] Changpeng Fan. Realizing a soft real-time framework for supporting distributed multimedia applications. In *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, page 128, Washington, DC, USA, 1995. IEEE Computer Society.

[FDC02] A. Friday, N. Davies, and K. Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*, page 13, Washington, DC, USA, 2002. IEEE Computer Society.

[FFR+04] Ian Foster, Markus Fidlerc, Alain Royd, Volker Sandere, and Linda Winkler. End-to-end quality of service for high-end applications. *Elsevier Computer Communications Journal*, 27(14):1375–1388, September 2004.

[Foh95] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, page 152, Pisa, Italy, December 1995.

[Fou] Real-Time Linux Foundation. List of real-time linux variants. Available at `http://www.realtimelinuxfoundation.org/variants/variants.html`.

[FWMM97] K. Fukuda, N. Wakamiya, M. Murata, and H. Miyahara. Qos mapping between user's preference and bandwidth control for video transport. In *Proceedings of the 5th International Workshop on Quality of Service*, pages 291–302, New York,USA, 1997.

[GAKT03] Sven Graupner, Artur Andrzejak, Vadim Kotov, and Holger Trinks. Adaptive control overlay for service management. In *First Workshop on the Design of Self-Managing Systems*, San Francisco, USA, June 2003.

[Gar99] Goutham Garimella. Advance cpu reservations with the dynamic soft real-time scheduler. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, 1992.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[GHRL04] Sourav Ghosh, Jeffery Hansen, Ragunathan (Raj) Rajkumar, and John Lehoczky. Adaptive qos optimizations with applications to radar tracking. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, Gothenburg, Sweden, August 2004.

[GJST81] M. R. Garey, D. S. Johnson, B. B. Simons, and R. E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal on Computing*, 10(2):256–269, May 1981.

[GK04] Dina Goldin and David Keil. Toward domain-independent formalization of indirect interaction. In *Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 393–394, Washington, DC, USA, 2004. IEEE Computer Society.

[GMG+04] Xiaohui Gu, Alan Messer, Ira Greenberg, Dejan Milojicic, and Klara Nahrstedt. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing Magazine*, 3(3):66–73, 2004.

[GP99] Vera Goebel and Thomas Plagemann. Mapping user-level qos to system-level qos and resources in a distributed lecture-on-demand system. In IEEE Computer Society, editor, *Proceedings of The 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, page 197, 199.

[GRH+03] Sourav Ghosh, Ragunathan Rajkumar, Jeffery P. Hansen, , and John P. Lehoczky. Scalable resource allocation for multi-processor qos optimization. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 174, Rhode Island, USA, May 2003. IEEE Computer Society.

[GRHL04] Sourav Ghosh, Ragunathan (Raj) Rajkumar, Jeffery Hansen, and John Lehoczky. Integrated resource management and scheduling with multi-resource constraints. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 12–22, Lisbon, Portugal, December 2004.

[Groa] Object Management Group. The common object request broker: Architecture and specification revision 3.1. Available at `http://www.omg.org/technology/documents/corba_spec_catalog.htm`.

[Grob] Object Management Group. The common object request broker architecture for embedded specification. Available at `http://www.omg.org/docs/formal/08-11-06.pdf`.

[Groc] Object Management Group. Real-time corba specification version 1.2. Available at `http://www.omg.org/docs/formal/05-01-04.pdf`.

[Gro97] Open Group. Dce 1.1: Remote procedure calls, 1997.

[GS96] L. Gilman and R. Schreiber. *Distributed Computing with IBM MQSeries*. Wiley, 1996.

[GVARG02] M. García-Valls, A. Alonso, J. F. Ruiz, and A. Groba. An architecture of a quality of service resource manager middleware for flexible multimedia embedded systems. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware*, pages 39–57, Orlando, Florida, USA, May 2002.

[Hal96] C. L. Hall. *Building Client/Server Applications Using TUXEDO*. Wiley, 1996.

[Haw03] Nicholas Hawes. *Anytime Deliberation for Computer Game Agents*. PhD thesis, School of Computer Science, The University of Birmingham, November 2003.

[Hay97] R. Hayton. Flexinet open orb framework. Technical report, APM Ltd, Cambridge, UK, October 1997.

[HB01] J. Hightower and G. Borriello. Location systems for ubiquitous computing. *IEE Computer*, 34(8):57–66, August 2001.

[HBS99] M. Hapner, R. Burridge, and R. Sharma. Java message service specification. Technical report, Sun Microsystems, 1999.

[HFL95] D. L. Hull, W. Feng, and J. W.-S. Liu. Enhancing the performance and dependability of real-time systems. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 174–182, Erlangen, Germany, April 1995.

[HLR01] Jeffery P. Hansen, John Lehoczky, and Ragunathan Rajkumar. Optimization of quality of service in dynamic systems. In *Proceedings of the 9th International Workshop on Parallel and Distributed Real-Time Systems*, April 2001.

[HLS97] T. Harrison, D. Levine, and D. Schmidt. The design and performance of a real-time corba event service. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications*, pages 184–200, Atlanta, Georgia, USA, October 1997.

[Hor88] Eric J. Horvitz. Reasoning under varying and uncertain resource constraints. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 111–116, 1988.

[Hud94] E. S. Hudders. *CICS: A Guide to Internal Structure*. Wiley, 1994.

[Iba88] T. Ibaraki. Enumerative approaches to combinatorial optimization - part i. *Annals of Operation Research*, 10(1-4):3–342, 1988.

[IK75] O. Ibarra and C. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of ACM*, 22:463–468, 1975.

[IT95] ITU-T/ISO. *Reference Model for Open Distributed Processing, RM-ODP (ISO/IEC 10746) X.901-X.904*, 1995.

[JB95] 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*, pages 64–75, London, UK, 1995. Springer-Verlag.

[Jef92] Kevin Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 89–99, Phoenix, Arizona, USA, December 1992.

[JLDB95] M. Jones, P. Leach, R. Draves, and J. Barrera. Modular real-time resource management in the rialto operating system. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, page 12. IEEE Computer Society, 1995.

[JLT85] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, December 1985.

[JMB04] Mark Jelasity, Alberto Montresor, and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *In Engineering Self-Organising Systems, G. Di Marzo Serugendo*, pages 265–282. Springer, 2004.

[JN04] Jingwen Jin and Klara Nahrstedt. Qos specification languages for distributed multimedia applications: A survey and taxonomy. *IEEE MultiMedia*, 11(3):74–87, 2004.

[JRR97] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. Cpu reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, October 1997.

[KBH+01a] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu. Infopipes for composing distributed information flows. In *Proceedings of the International Workshop on Multimedia Middleware*, pages 44–47, Ottawa, Ontario, Canada, October 2001.

[KBH+01b] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu. Thread transparency in information flow middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 121–140, Heidelberg, Germany, November 2001.

[Kha98] S. Khan. *Quality Adaptation in a Multisession Multimedia System: Model, Algorithms and Architecture*. PhD thesis, University of Victoria, 1998.

[KHR01] Ulrich Kermer, Jamey Hicks, and James Rehg. A compilation framework for power and energy management on mobile computers. In *14th International Workshop on Parallel Computing*, pages 115–131, 2001.

[Kos02] R. Koster. *A Middleware Platform for Information Flows*. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, July 2002.

[KR06] K. Kwiat and Shangping Ren. A coordination model for improving software system attack-tolerance and survivability in open hostile environments. In *Proceedings of the IEEE International Conference on*

Sensor Networks, Ubiquitous, and Trustworthy Computing, pages 394–402, Taichung, Tawain, June 2006.

[KRL+00] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhaes, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pages 121–143, New York, USA, 2000.

[KRP+93] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Lab] Ericsson Computer Science Laboratory. Open source erlang. Available at `http://www.erlang.org/`.

[LB00] Giuseppe Lipari and Sanjoy Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the 12th ECRTS*, pages 193–200, Stockholm, Sweden, 2000.

[LB05] Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE RTSS*, pages 410–421, 2005.

[LK00] Averill M. Law and W. David Kelton. *Simulation modeling and analysis*. McGraw-Hill, 3rd edition, 2000.

[LKRM96] C. Lee, Y. Katsuhiko, R. Rajkumar, and C. Mercer. Predictable communication protocol processing in real-time mach. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1996.

[LL73] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):40–61, 1973.

[LL07] Chunlin Li and Layuan Li. Utility-based qos optimisation strategy for multi-criteria scheduling on the grid. *Journal of Parallel and Distributed Computing*, 67(2):142–153, 2007.

[LLA01] Gerardo Lamastra, Giuseppe Lipari, and Luca Abeni. A bandwidth inheritance algorithm for real-time task synchronization in open systems.

In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 151–160, London, UK, December 2001.

[LLA04] Giuseppe Lipari, Gerardo Lamastra, and Luca Abeni. Task synchronization in reservation-based real-time systems. *IEEE Transactions on Computers*, 53(12):1591–1601, 2004.

[LLB+94] 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. *Foundations of Dependable Computing: Paradigms for Dependable Applications*, pages 157–182, 1994.

[LLS+91] Jane W.S. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, 1991.

[LLS+99] Chen Lee, John Lehoczky, Dan Siewiorek, Ragunathan Rajkumar, and Jef Hansen. A scalable solution to the multi-resource qos problem. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 315–326, 1999.

[LRT92] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In *Proceedings of the 13th RTSS*, pages 110–123, December 1992.

[LW82] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[LWX01] 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*, pages 238–246. ACM Press, 2001.

[LWX02] Zhiyuan Li, Cheng Wang, and Rong Xu. Task allocation for distributed multimedia processing on wirelessly networked handheld devices. In *Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, page 79. IEE Computer Society, 2002.

[MAC+82] S. Makridakis, A. Andersen, R. Carbone, R. Fildes, M. Hibon, R. Lewandowski, J. Newton, E. Parzen, and R. Winkler. The accuracy of

extrapolation (time series) methods: Results of a forecasting competition. *Journal of Forecasting*, 1:111–153, 1982.

[MC94] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[MC02] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad-hoc networks. In *Proceedings of the International Workshop on Distributed Event-Based Systems*, pages 639–644, Vienna, Austria, July 2002.

[MFSV06] L. Mangeruca, A. Ferrari, and A. L. Sangiovanni-Vincentelli. Uniprocessor scheduling under precedence constraints. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 157–166, San Jose, CA, USA, April 2006.

[Mica] Sun Microsystems. Java remote method invocation. Available at `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`.

[Micb] Sun Microsystems. Jini. Available at `http://www.jini.org/`.

[MJ95] S. McCanne and V. Jacobson. Vic: A flexible framework for packet video. In *Proceedings of the ACM Multimedia' 95*, November 1995.

[MLBC04] Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. Iris: A new reclaiming algorithm for server-based real-time systems. In *Proceedings of the 10th IEEE RTAS*, page 211, Toronto, Canada, 2004.

[MMB03] Alberto Montresor, Hein Meling, and Özalp Babaoglu. Toward self-organizing, self-repairing and resilient distributed systems. In *Future Directions in Distributed Computing*, pages 119–126, 2003.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[MOFR01] L. Marcenaro, F. Oberti, G. L. Foresti, and C. S. Regazzoni. Distributed architectures and logical-task decomposition in multimedia surveillance systems. *Proceedings of the IEEE*, 89(10):1419–1440, October 2001.

[Mok83] A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[MR02] Sebastian Möller and Alexander Raake. Telephone speech quality prediction: towards network planning and monitoring models for modern network scenarios. *Speech Communication*, 38(1):47–75, 2002.

[MST94] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, 1994.

[MT90] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, 1990.

[Nak98] Kaoru Nakazono. Frame rate as a qos parameter and its influence on speech perception. *Multimedia Systems*, 6(5):359–366, 1998.

[NBBB98] K. Nichols, S. Blake, F. Baker, and D. Black. IETF RFC 2474: Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers, 1998.

[NhCN98] K. Nahrstedt, H. hua Chu, and S. Narayan. Qos-aware resource management for distributed multimedia applications. *Journal of High Speed Networks*, 7(3-4):229–257, December 1998.

[NL97] Jason Nieh and Monica Lam. The design, implementation and evaluation of smart: a scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197, October 1997.

[NP05] Luís Nogueira and Luís Miguel Pinho. Dynamic qos-aware coalition formation. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, page 135, Denver, Colorado, April 2005.

[NP06a] Luís Nogueira and Luís Miguel Pinho. Building adaptable, qos-aware dependable embedded systems. In *Proceedings of the 3rd International Workshop on Dependable Embedded Systems*, pages 72–77, Leeds, United Kingdom, October 2006.

[NP06b] Luís Nogueira and Luís Miguel Pinho. Dynamic adaptation of stability periods for service level agreements. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 77–81, Sydney, Australia, August 2006.

[NP06c] Luís Nogueira and Luís Miguel Pinho. Iterative refinement approach for qos-aware service configuration. *IFIP From Model-Driven Design to Resource Management for Distributed Embedded Systems*, 225:155–164, 2006.

[NP07a] 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.

[NP07b] Luís Nogueira and Luís Miguel Pinho. Handling shared resources and precedence constraints in open systems. In *Proceedings of the WiP session of the 19th Euromicro Conference on Real-Time Systems*, Pisa, Italy, July 2007.

[NP08a] Luís Nogueira and Luís Miguel Pinho. Dynamic qos adaptation of inter-dependent task sets in cooperative embedded systems. In *Proceedings of the 2nd ACM International Conference on Autonomic Computing and Communication Systems*, page 97, Turin,Italy, September 2008.

[NP08b] Luís Nogueira and Luís Miguel Pinho. Handling qos dependencies in distributed cooperative real-time systems. *IFIP Distributed Embedded Systems: Design, Middleware and Resources*, September 2008.

[NP08c] Luís Nogueira and Luís Miguel Pinho. Shared resources and precedence constraints with capacity sharing and stealing. In *Proceedings of the 22th IEEE International Parallel and Distributed Processing Symposium*, page 97, Miami,Florida,USA, April 2008.

[NP09a] Luís Nogueira and Luís Miguel Pinho. 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.

[NP09b] 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.

[Off] DARPA Information Technology Office. An integrated multi-layer approach to software enabled control: Mission planning to vehicle control. Available at `http://www.aem.umn.edu/people/faculty/balas/darpa_sec/index.html`.

[OH98] Mazliza Othman and Stephen Hailes. Power conservation strategy for mobile computers using load sharing. *SIGMOBILE Mobile Computing Communications Review*, 2(1):44–51, 1998.

[PNB05] Luís Miguel Pinho, Luís Nogueira, and Ricardo Barbosa. An ada framework for qos-aware applications. In *Proceedings of the 10th Ada-Europe International Conference on Reliable Software Technologies*, pages 25–38, York, UK, June 2005.

[PVC+05] 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 Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 67–75, Palma de Mallorca, Spain, July 2005.

[Riva] Wind River. Real-time linux. Available at `http://www.windriver.com/products/platforms/real-time_core/`.

[Rivb] Wind River. Vxworks rtos. Available at `http://www.windriver.com/products/vxworks/`.

[RJM+98] R. Rajkumar, K. Juvva, A. Molano, , and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[RLLS97] 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*, page 298. IEEE Computer Society, 1997.

[Rod02] G. Rodosek. Quality aspects in it service management. In *Proceedings of the 13th IFIP/IEEE Internation Workshop on Distributed Systems: Operations and Management*, pages 82–93, Montereal, Canada, October 2002.

[RRPK98] 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(1):19–26, 1998.

[RS94] K. Ramamritham and J.A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, January 1994.

[RST06] Shangping Ren, Limin Shen, and J. Tsai. Reconfigurable coordination model for dynamic autonomous real-time systems. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pages 60–67, Taichung, Tawain, June 2006.

[RtMSL] Real-time and Carnegie Mellon University Multimedia Systems Laboratory. Linux/resource kernel. Available at `http://www.cs.cmu.edu/~rajkumar/linux-rk.html`.

[Sah75] S. Sahni. Approximation algorithms for the 0-1 knapsack problem. *Journal of ACM*, 23:555–565, 1975.

[SB94] Marco Spuri and Giorgio Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Proceedings of the 15th IEEE Real-Time System Symposium*, pages 2–11, San Juan, Puerto Rico, December 1994.

[SCC04] John Shackleton, Darren Cofer, and Saul Cooper. Anytime scheduling for real-time embedded control applications. In *Proceedings of the 23rd Digital Avionics Systems Conference*, volume 2, pages 101–110, Salt Lake City, UT, USA, October 2004.

[SCFJ96] H. Schulzrinne, S. Casner, R. Frederic, and V. Jacobson. IETF RFC 1889: Rtp: A transport protocol for real-time applications, 1996.

[Sch93] D. C. Schmidt. The adaptive communication environment: An object-oriented network programming toolkit for developing communication software. *Concurrency: Practice and Experience*, 5(4):269–286, 1993.

[SCZ05] Michael Stonebraker, Ugur Cetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[SdML99] Mallikarjun Shankar, Miguel de Miguel, and Jane W. S. Liu. An end-to-end qos management architecture. In *Proceedings of the 5th*

IEEE Real-Time Technology and Applications Symposium, pages 176–191, Washington, DC, USA, 1999. IEEE Computer Society.

[Ser06] G. Di Marzo Serugendo. *Autonomous Systems with Emergent Behaviour*, chapter Handbook of Research on Nature Inspired Computing for Economy and Management, pages 429–443. Idea Group, Inc., Hershey-PA, USA, September 2006.

[SH02] D. C. Schmidt and S. D. Huston. *C++ Network Programming: Mastering Complexity Using ACE and Patterns*. Addison-Wesley, 2002.

[SLM98] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the tao real-time object request broker. *Computer Communications*, 21:294–324, April 1998.

[SLS04] Rodrigo Santos, Giuseppe Lipari, and Jorge Santos. Scheduling open dynamic systems: The clearing fund algorithm. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 114–129, Gothenburg, Sweden, August 2004.

[SLS+06] Praveen Kaushik Sharma, Joseph Loyall, Richard E. Schantz, Jianming Ye, Prakash Manghwani, Matthew Gillen, and George T. Heineman. Managing end-to-end qos in distributed embedded applications. *IEEE Internet Computing*, 10(3):16–23, 2006.

[SLSL05] Sven Schmidt, Thomas Legler, Daniel Schaller, and Wolfgang Lehner. Real-time scheduling for data stream management systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 167–176, 2005.

[SMK03] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten. Architecture and operation of an adaptable communication substrate. In *Proceedings of the 9th IEE International Workshop on Future Trends of Distributed Computing Systems*, pages 46–55, May 2003.

[Sri95] R. Srinivasan. IETF RFC 1831: Open network computing remote procedure call – version 2, 1995.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronisation. *IEEE Transaction on Computers*, 39(9):1175–1185, 1990.

[SS94] Marco Spuri and John A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers*, 43(12):1407–1412, 1994.

[SSL89] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, 1989.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[SWM95] Richard Staehli, Jonathan Walpole, and David Maier. A quality-of-service specification for multimedia presentations. *Multimedia Syst.*, 3(5-6):251–263, 1995.

[TK93] H. Tokuda and T. Kitayama. Dynamic qos control based on real-time threads. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 113–122, November 1993.

[TUoA] Computer Science Department The University of Arizona. The cactus project. Available at `http://www.cs.arizona.edu/projects/cactus/`.

[vdBFK06] Jur van den Berg, David Ferguson, and James Kuffner. Anytime path planning and replanning in dynamic environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2366– 2371, Orlando, Florida, USA, May 2006.

[vHtT00] F. van Harmelen and A. ten Teije. Describing problem solving methods using anytime performance profiles. In *Proceedings of ECAI'00*, pages 181–186, Berlin, August 2000.

[VN97] Nalini Venkatasubramanian and Klara Nahrstedt. An integrated metric for video qos. In *MULTIMEDIA '97: Proceedings of the fifth ACM international conference on Multimedia*, pages 371–380, New York, NY, USA, 1997. ACM.

[WEE+07] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Muller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 2007.

[WF08] G. R. Wiedenhoft and A. A. Fröhlich. Using imprecise computation techniques for power management in real-time embedded systems. In *Proceedings of the 6th IFIP Working Conference on Distributed and Parallel Embedded Systems*, pages 121–130, Milano, Italy, September 2008.

[WL04] 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.

[WLP02] Song Wang, Kwei-Jay Lin, and Song Peng. Bwe: A resource sharing protocol for multimedia systems with bandwidth reservation. In *Proceedings of the 4th IEEE International Symposium on Multimedia Software Engineering*, pages 158–165, New-port Beach,CA,USA, December 2002.

[WS00] X. Wang and H Schulzrinne. An integrated resource negotiation, pricing, and qos adaptation framework for multimedia applications. *IEEE Journal on Selected Areas in Communications*, 18(12):2514–2529, December 2000.

[WSK03] N. Wang, D. C. Schmidt, and M. Kircher. Towards an adaptive and reflective middleware framework for qos-enabled corba component model applications. *IEE Distributed System Online Special Issue on Reflective Middleware*, 2003.

[WWGF08] Geovani Ricardo Wiedenhoft, Lucas Francisco Wanner, Giovani Gracioli, and Antônio Augusto Fröhlich. Power management in the epos system. *ACM SIGOPS Operating Systems Review*, 42(6):71–80, October 2008.

[YVH08] Heng Yu, Bharadwaj Veeravalli, and Yajun Ha. Dynamic scheduling of imprecise-computation tasks in maximizing qos under energy constraints for embedded systems. In *Proceedings of the 13th Conference on Asia and South Pacific Design Automation*, pages 452–455. IEEE Computer Society Press, January 2008.

[ZBS97] J. Zinky, D. Bakken, and R. Schantz. Architecture support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1), January 1997.

[ZDE+93] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. Rsvp: A new resource reservation protocol. *IEEE Network*, pages 8–18, September 1993.

[Zil93] Shlomo Zilberstein. *Operational Rationality Through Compilation of Anytime Algorithms*. PhD thesis, Department of Computer Science, University of California at Berkeley, 1993.

[Zil96] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *Artificial Inteligence Magazine*, 17(3):73–83, 1996.

[ZR96] Shlomo Zilberstein and Stuart Russel. Optimal composition of real-time systems. *Artificial Inteligence Magazine*, 82(1-2):181–213, 1996.