COORDINATED MANAGEMENT OF RESOURCE ALLOCATIONS AND APPLICATION QUALITY OF SERVICE LEVEL ADAPTATION FOR REAL-TIME SYSTEMS

A Thesis Presented to
The Faculty of the
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirement for the Degree
Master of Science
by
Shikha Jain

November, 2002
ACKNOWLEDGEMENTS

I would like to thank my advisors Dr. Carl Bruggeman and Dr. Lonnie Welch for their continuing support and their invaluable contribution to this thesis and also for their constant faith in me. Among the many things I have learnt from them, the most important one has been the ability to look beyond mere numbers and understand the real implication and meaning of the work that I have been involved in.

I would also like to thank Dr. David Chelberg for his suggestions, help and comments on my research, and also for serving on my examination committee. I would like to thank Ms. Barbara Pfarr of the NASA Goddard Space Flight Center for her helpful suggestions and financial support. This project is sponsored by NASA under the contract HART Grant 10349. I also thank Mr. Chris Shuler, Ms. Mei Liu and Mr. Matt McCoy of Computer Sciences Corporation for their inputs and ideas.

I also thank David Fleeman, David Parrott, Zhenyu Tan, Anbu, Purvi, Lu and all the other members of the Laboratory for Intelligent Real-Time Secure Systems (LIRTSS) for all the unstinting assistance that they offered.

Lastly, I would like to thank Seenu, my parents and all my friends for their support and confidence in me.
# TABLE OF CONTENTS

 CHAPTER 1 .............................................................................................................................................................................. 1  
 INTRODUCTION ........................................................................................................................................................................... 1  
 CHAPTER 2 ...................................................................................................................................................................................... 6  
 BACKGROUND .................................................................................................................................................................................. 6  
 2.1 DeSiDeRaTa Resource Manager ....................................................................................................................................... 13  
 2.1.1 Object Oriented DeSiDeRaTa Resource Manager ......................................................................................................... 16  
 CHAPTER 3 .................................................................................................................................................................................... 21  
 SERVICE LEVELS AND UTILITY BASED PARADIGM .................................................................................................................... 21  
 3.1 Motivating Example ................................................................................................................................................................. 22  
 3.2 Service Level and Utility Definitions ....................................................................................................................................... 28  
 3.3 DeSiDeRaTa Approach with Service Levels and Utility ....................................................................................................... 29  
 3.3.1 UML Model .............................................................................................................................................................................. 30  
 3.4 Use Cases .................................................................................................................................................................................... 31  
 3.4.1 Load a real-time software system specification file ............................................................................................................. 31  
 Load a real-time software system specification file: Requirements ............................................................................................ 32  
 3.4.2 Maintain a feasible allocation .................................................................................................................................................. 35  
 3.4.3 Observe the Service Level and Utility ................................................................................................................................ 41  
 3.5 Mathematical System Model ..................................................................................................................................................... 44  
 3.6 Specification Language and Grammar ...................................................................................................................................... 48  
 3.7 Algorithm to Determine the Service Levels ............................................................................................................................. 51  
 CHAPTER 4 .................................................................................................................................................................................... 56  
 EXPERIMENTS AND RESULTS ...................................................................................................................................................... 56  
 4.1 HART System Test Bed .............................................................................................................................................................. 56  
 4.1.1 Architecture of the Test Bed prototype ................................................................................................................................. 59  
 4.2 Experiments .................................................................................................................................................................................. 62  
 4.2.1 Experiment 1: ......................................................................................................................................................................... 64  
 4.2.2 Experiment 2: ......................................................................................................................................................................... 68  
 4.2.3 Experiment 3: ......................................................................................................................................................................... 73
4.2.4 Experiment 4: ................................................................................................. 75
4.3 Results ................................................................................................................ 76
CHAPTER 5 ............................................................................................................... 78
CONCLUSIONS AND FUTURE WORK ................................................................... 78
BIBLIOGRAPHY ....................................................................................................... 81
APPENDIX A ........................................................................................................... 87
1. Example Hardware Configuration Specification File .......................................... 87
2. Example Real-Time Software System Specification File .................................... 87
3. Example Real-Time Software System Specification File with service level threshold ................................................................. 89
APPENDIX B .......................................................................................................... 93
LIST OF FIGURES

Figure 2.1. A real-time subsystem.................................................................14
Figure 2.2. Logical architecture of the resource and QoS management software........15
Figure 2.3. Architecture of DeSiDeRaTa................................................................17
Figure 2.4. Subsystem dependencies and layers ..............................................19
Figure 3.1. Motivating example scenario.......................................................25
Figure 3.2. HART prototype system diagram.................................................27
Figure 3.3. Use-case diagram .......................................................................31
Figure 3.4. Analysis collaboration diagram ....................................................34
Figure 3.5. Design class diagram ....................................................................35
Figure 3.6. Analysis collaboration diagram ....................................................39
Figure 3.7. Design class diagram ....................................................................41
Figure 3.8. Analysis collaboration diagram ....................................................43
Figure 3.9. Design class diagram ....................................................................44
Figure 3.10. Software specification file ..........................................................49
Figure 3.11. Grammar for Application............................................................50
Figure 3.12. Grammar for Service Levels.......................................................51
Figure 4.1. HART high level conceptual diagram ..........................................57
Figure 4.2. Data Flow ....................................................................................59
Figure 4.3. Program flow of the IPA...............................................................60
Figure 4.4. Cloud Display (Demonstrator)......................................................61
Figure 4.5. Camera Simulator program flow ....................................................62
Figure 4.6. Experimental Set-up: H/W: 2 Sun workstations N/W 100 Mbps........64
Figure 4.7. IPA and camera simulator successfully started on Water................65
Figure 4.8. Observed Latency and Service Levels for IPA...............................66
Figure 4.9. Application IPA is stopped............................................................66
Figure 4.10. IPA misses an image.................................................................67
Figure 4.11 Observed latency and service levels for IPA (with 40% load).........69
Figure 4.12. Cloud cover percentage with Algorithm 1 ........................................ 70
Figure 4.13. Cloud cover display with Algorithm 4 .............................................. 71
Figure 4.14 Observed latency and service levels of IPA (with 60% load) ................. 72
Figure 4.15: Performance at the lowest service level ........................................... 73
Figure 4.16. Performance of IPA with modified run-time allocation algorithm ......... 75
Figure 4.17. Performance of IPA service levels with varying CPU idle time .......... 76
LIST OF TABLES

Table 3.1: Generic Components.........................................................................................45
Table 3.2: Path attributes..................................................................................................46
Table 3.3: Hardware Attributes.........................................................................................46
Table 3.4: QoS Parameters.............................................................................................47
Table 3.5: Utility Model..................................................................................................48
CHAPTER 1

INTRODUCTION

The canonical definition of a real-time system (from Donald Gillies), is the following “A real-time system is one in which the correctness of the computations depends not only upon the logical correctness but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred”. Real-time then, is a property of systems where time is literally “of the essence.” In a real-time system, the value of a computation depends on how timely the answer is. There are three types of real-time systems: hard, firm and soft.

Hard real time is a property of the timeliness of a computation in a system. A hard real-time constraint in a system is one for which there is no value to a computation if it is late, and the effects of a late computation may be catastrophic to the system. For such a system, all of the activities must be completed within the specified deadline. A firm real-
time system is one in which the deadlines are not as strict as hard real-time. Such systems can afford to miss a few deadlines, generating useful results. Soft real time is a property of the timeliness of a computation where the utility (value) diminishes according to its tardiness. A soft real-time system can tolerate some late answers to soft real-time computations, as long as the utility hasn't diminished to zero. The utility of the soft real-time system is directly proportional to the quality of the output. One of the ways to quantify the quality of the output is to discretely characterize the Quality of Service (QoS) levels based on a user specified utility value. This quantization is termed as service levels for a real-time system (or application). At each of these levels, the resource requirements and execution times for the application are different and provide a certain utility to the user. Applications that can be characterized using QoS levels are said to execute at multiple levels of fidelity. This thesis focuses on introducing the concepts of service levels and utility to provide a coordinated management of resource allocations and application QoS level adaptation for real-time systems.

Problems arise when there is competition for resources in a system, and resources are shared among many applications. Such kinds of applications place a burden on the operating system. Though operating systems have built-in resource management policies, they are very general and try to provide a fair share of resources to all competing applications. This is essentially a best-effort approach to satisfy the resource needs of as many applications as possible, but it does not have a notion of real-time constraints. These resource management policies do not provide a mechanism by which the
applications can specify their timing requirements, verify that the timing requirements are met, or react to missed deadlines. Further, they do not guarantee the availability of resources until the critical real-time applications have finished their processing. Moreover, most real-time applications exhibit dynamic behavior and hence their characteristics cannot be captured completely in a static model. This is where the need arises to apply resource management to an operating system. A system is real-time to the degree that it employs real-time resource management -- its resources are explicitly managed for the purpose of operating in real time in dynamically varying environments. The degree to which a system is real-time can be quantified in different ways but has no widely accepted metric; thus, it is usually subjective. One such quantification is based on the Quality of Service (QoS) levels.

Existing approaches to adaptive resource and QoS management typically perform either reallocation of resources or adaptation of application QoS levels in response to dynamic environment changes. While success has been achieved with both these approaches individually, the potential synergy of a combined technique has not been explored. The DesiDeRata (Dynamic Scalable Dependable Real-Time System) approach uses adaptive resource allocation to provide real-time guarantees. The DesiDeRata approach supports the dynamic path concept for automated QoS assessment and resource allocation. Path-level QoS specification is used by the adaptive resource allocator to determine if the current configuration is achieving the desired QoS levels and to assist in selecting new configurations to improve QoS.
The DeSiDeRaTa approach may not scale very well with real-time systems that are capable of running at multiple levels of fidelity. In addition, some of these applications are also capable of monitoring their own performance. When a QoS violation occurs, these applications try to adapt themselves by changing their QoS level and might use significantly more resources in order to meet their deadlines. Since these applications do not have a global perspective, the decisions made to adapt themselves might be selfish. Moreover, one of the corrective actions that DeSiDeRaTa takes to restore QoS is to move some of the applications to another host in order to reduce overload. Moving an application can cause an additional overhead of storing the state of the application and keeping track of its dependencies. Thus, by introducing the concepts of service levels and utilities, DeSiDeRaTa could manage the real-time system and application resources in a more effective way. In the case of applications that have service levels, DeSiDeRaTa can change the service level instead of moving the application to a new host. The applications would adapt to best take advantage of the global set of resources available to the real-time systems. Though these two approaches have been applied independently to real world problems [1][20], the idea of an integrated approach for real-time systems has not been explored.

The concept of service levels and utility is used in integrating the two approaches. Each application is associated with a service level that provides a certain utility to the user. The utility describes the benefit provided by that application. Since the utility at a particular service level is fixed, the application is said to provide static benefit. In
addition to other QoS parameters like real-time deadlines, execution time, and period, DeSiDeRaTa also considers the service level information for the applications that can operate at multiple levels of fidelity. In order to accommodate these features, the DeSiDeRaTa specification language has been enhanced.

The remainder of this thesis is organized as follows. Chapter 2 provides a historical perspective of research in the field of real-time systems and resource management techniques. A brief background of the DeSiDeRaTa resource manager is also presented. Chapter 3 presents the concept of service levels and utility, which is the main contribution of this thesis. The current DeSiDeRaTa architecture and the enhancements made to the resource allocation algorithms to deal with soft real-time systems are also described. The mathematical model and specification language for the DeSiDeRaTa resource management system are discussed. Chapter 4 presents the architecture of the test bed used to validate the approach. The experiments and results derived are also explained. Finally, Chapter 5 summarizes the conclusions derived from this work and details future work.
CHAPTER 2

BACKGROUND

Real-time process scheduling and resource management is a well-established research field. Scheduling and resource management are needed to provide guaranteed services to applications. The real-time research community is investigating methods to provide support for distributed real-time systems to meet their quality of service requirements such as timeliness and availability of resources in hostile environments. Although the motivation for the problem has the same root, all the approaches differ in their conceptual model of the system.

Classical real-time research focuses on hard real-time systems for synchronous monitoring and control – usually centralized but occasionally distributed. In most real-time computing models the Worst-Case Execution Time (WCET) is used to characterize the workload a priori. In [12] Puschner and Burns provide a review of worst-case analysis techniques. They consider WCET analysis to be hardware dependent, making it
an expensive operation on distributed heterogeneous clusters. In most cases, a worst-case approach is not optimal since a lot of resources go unutilized. One of the original papers on real-time scheduling by Liu and Layland [9] proposes a scheduling technique based on Rate Monotonic Analysis (RMA) and Earliest Deadline First (EDF) approach. This approach considers the worst-case execution time to be constant and defined \textit{a priori}. Characterizing workload of real-time systems \textit{a priori} based on worst case execution times can lead to poor resource utilization, particularly if the average case is significantly less than the worst case. Some of the drawbacks of using a WCET based approach are cited in [10][11]. Classical real-time research focuses on static scheduling based on a priori workload characteristics. Thus it does not work very well with dynamic distributed systems.

Modern real-time applications, including multimedia systems, mobile robotics, satellites, and distributed monitoring architectures, often operate in highly dynamic environments where workload conditions are difficult to predict in advance. In addition, real-time activities may have variable computational requirements and are typically characterized by more flexible timing constraints than classical real-time theory permits. Handling such systems according to a hard real-time paradigm (based on worst-case assumptions) is inappropriate, because it would cause a waste of resources and would dramatically increase the cost. Moreover it is difficult to employ or adapt such techniques for systems that are non-periodic and distributed. Non-periodic real-time computer systems are inherently posteriori in terms of their workload characteristics and hence
require adaptive real-time resource management. Recently, significant work has been devoted to increasing the flexibility and the efficiency of dynamic real-time systems, still providing a form of performance guarantee.

Research in the field of imprecise computation [14][18] examines the strategy of having two parts to a task: a mandatory part and an optional part. The optional part merely refines the result obtained by the mandatory part and is less important. The optional part is used to minimize the computational error. The scheduling algorithm used statically allocates resources for mandatory part and tries to grant as many CPU cycles as possible (dynamically) to the optional part in such a way as to minimize the total error. This approach provides a guarantee of meeting the deadlines for only mandatory parts. Though this work presents an approach for managing real-time systems, there is no mechanism to detect missed deadlines and notify the system about the missed deadlines. While in this approach there are no real-time guarantees for the optional part, the approach presented in this thesis provides a mechanism to specify the real-time constraints for the system as a whole (both mandatory and optional part) and tries to satisfy the real-time performance requirements by using the concept of service levels and utility.

Jensen, et al. [15] propose a benefit based real-time scheduling model for the applications. The applications specify a benefit curve that indicates the relative benefit to be obtained by scheduling the application at a particular time with respect to their deadlines. Here, the completion of a process or a set of processes typically has a value or
benefit to the system, which can be expressed as a function of time. The goal of his research was to maximize the overall system benefit. The difference between this approach and ours is that in our approach the benefit or utility is associated with the different service levels of the application. These service level settings for the applications are used to finish a task within the required time constraints.

A new concept of value [7] was introduced by using a scheduling algorithm for tasks described by time value functions. In [11][28] Buttazzo et al. study the effect of value based against deadline based scheduling algorithms under overload conditions. They conclude that the best approach should be an integrated approach where EDF is used to schedule before the overload conditions and use value-based scheduling after overload conditions so that the real-time guarantees for tasks with higher priority or value are satisfied.

Burns et al. [29] presents yet another example of using value in scheduling flexible real-time systems. The notion of value is used to gain run-time control during the decision process. This not only provides a method to use the resources effectively, but also reacts gracefully to failures and overloads. Although these approaches are very appealing, they require a complex heuristic algorithm to be executed in order to dynamically compute the time-value function and determine the best schedule. This will increase the computational complexity of the problem.

Howes et al. [25] present a scheduling method based on end-to-end deadlines instead of artifact deadlines introduced by designers. There are no priorities associated
with any task and hence are known as peer tasks. While all the above approaches present solutions for dynamic real-time systems, each of them uses some kind of scheduling algorithm to be used with real-time operating system.

In 1998, Welch et al. [1] developed an adaptive QoS management system, DeSiDeRaTa, which incorporates the knowledge of resource needs in the distributed computer control system and supports dynamic path based systems. Instead of performing real-time scheduling, this approach uses a metric to predict the real-time deadlines and uses this predicted value to predict a quality of service violation. Based on this prediction DeSiDeRaTa takes corrective action to restore the real-time constraints before an actual violation occurs. This model is designed for hard real-time systems. The dynamic distributed real-time systems can be modeled using the specification language [2][5]. This specification language is application-language-independent. In [6][8][13][23] Welch et al. present an adaptive resource management technique based on real-time paths using a posteriori workload characteristics. Under overload conditions, DeSiDeRaTa reallocates resources in such a way so as to satisfy the real-time requirements of the system under its control.

In the field of soft real-time systems, Brandt et al. [16][19][20][21][22] provide techniques for soft real-time application execution with dynamic quality of service assurance. The technique developed is called Dynamic Quality of Service Management (DQM), which mediates application resource usage so as to ensure that applications get the resources they need in order to provide required performance. The applications need
to notify the DQM when they miss a deadline. This could be a potential bottleneck when there are a large number of applications that miss their deadlines. Thus this approach may not scale very well for large, complex distributed systems. Although the techniques presented by Brandt et al [16] and this thesis are similar in that they handle the service levels of an application, there are a few differences. The main difference is that in DQM, the applications monitor themselves and are responsible for notifying the DQM when a deadline is missed whereas in the DeSiDeRaTa approach, the resource manager is responsible for both detection and diagnosis of the violation of QoS.

Rodrigues et al. [32][33] present an adaptive quality of service framework based on Quality Objects (QuO) and the CORBA A/V streaming service. In their approach the applications monitor themselves and adapt themselves to changes in resource availability to meet QoS requirements. One drawback of such an approach is that the applications do not have knowledge about the other applications in the system. The decisions made by such applications are based solely on their needs. Such decision-making might cause system instability or even complete breakdown.

Rajkumar et al. [27][30][31] have developed a model based on QoS levels called Q-RAM. It uses benefit functions to specify application benefit as a function of resource allocations. The authors present algorithms that dynamically select or negotiate the levels of each application or enable the application to select their levels such that the total benefit is maximized. Our approach differs from the approach presented by Rajkumar et al. in the sense that in their approach the application have some control over the service
level at which they execute while in our approach, the decisions regarding the service levels are made by RM.

Atkins et al. [26] present a technique that uses discreet QoS levels to provide a graceful degradation under overload conditions. They deal with applications that have the capability to request resources from the resource manager. If the request is not granted, then a rejection penalty is applied. The QoS Provider RTPOOL presented in [26] is used for scheduling the real-time tasks. While this approach requires a scheduler to maintain the QoS of a system, in our approach, the QoS Manager triggers the action to be taken. The RM components carry out the actions and let the OS scheduler handle the CPU level scheduling.

Welch et al. [4][34] propose that the utility theory could be used to capture the hard-, firm- and soft- real-time requirements and produce a resource manager for a mixed real-time environment. It discusses how benefit theory can be applied to real-time computing paradigms and categorizes the different types of benefits. In our approach the benefit is a function of output quality and resource allocation. While there have been techniques to handle resource management for hard and soft real-time systems independently, no methodology has been developed to integrate the two approaches to have a coordinated resource management for mixed real-time systems.

DeSiDeRaTa’s (Dynamic Scalable Dependable Real-Time System) capability to effectively manage the quality of service and resources of large distributed hard real-time systems has been established in [17]. This research is aimed at producing a methodology
to guarantee the quality of service for soft real-time systems along with hard real-time systems.

2.1 **DeSiDeRaTa Resource Manager**

*DeSiDeRaTa* (Dynamic Scalable Dependable Real-Time System) is an adaptive resource management technique [1] that provides middleware services for distributed dynamic real-time systems that cannot be characterized *a priori*. It provides innovative QoS management technology that incorporates the knowledge of needs in the distributed real-time mission-critical systems domain. The technology includes a specification language [2][5] and dynamic QoS management software that support dynamic, path-based systems. The specification language is used to describe the various environment dependent features, hardware and software configurations and the structural and QoS properties of the software.

*DeSiDeRaTa* technology supports the dynamic path concept for automated QoS assessment and resource allocation. An example of a dynamic path is shown in Figure 2.1. A dynamic path is a very large-grain entity, which typically consists of sensors, actuators, and control software for filtering, evaluating, and acting. The paths may have timing constraints, widely varying dynamic behavior, and may be scalable and fault tolerant. A generic path design pattern contains: (1) a data (and/or event) source, (2) a data (and/or event) stream and (3) a data (and/or event) consumer. The data (and/or event) *source* produces a *stream* of data (and/or event), which cause the consumer to perform processing. A data (and/or event) source typically consists of one or more
sensors, but may also be a clock or a software entity. A consumer often contains one or more actuators. A data consumer evaluates each datum or event to decide whether the actuators should perform actions.

Path-level QoS specification is used by the adaptive resource allocator to determine if the current configuration is achieving the desired QoS and to assist in selecting new configurations to improve QoS. This is significantly different than other approaches for assessment in the sense that it is performed dynamically, and is performed at a much larger granularity. Moreover, DeSiDeRaTa differs from previous work in that it accounts for the complex features of dynamic real-time systems. These features include previously overlooked issues with respect to granularity, variable periods, priorities, fault management and scalability. The logical architecture of the DeSiDeRaTa QoS management software is shown in Figure 2.2.

Figure 2.1. A real-time subsystem
Figure 2.2. Logical architecture of the resource and QoS management software.

The application programs of real-time control paths send time-stamped events to the QoS metrics component (1), which calculates path-level QoS metrics and sends them to the QoS monitor component (2). The monitor checks for conformance of observed QoS to required QoS, and notifies the QoS diagnosis component (3) when a QoS violation occurs. The diagnoser notifies the action selection component (4) of the cause(s) of poor QoS and recommends actions (e.g., move a program to a different host or LAN, shed a program, or replicate a program) to improve QoS. Action selection ranks the recommended actions, identifies redundant actions, and forwards the results to the allocation analysis component (7); this component consults resource discovery (6) for host and LAN load index metrics (5), and determines an efficient method for allocating the hardware resources to perform the actions, and requests that the actions be performed by the allocation enactment component (8).
2.1.1 Object Oriented DeSiDeRaTa Resource Manager

Using the described architecture as a guideline, the Resource Manager (RM) was remodeled to follow an object-oriented design. The resource manager is implemented using the unified software development process, which is use-case driven, architecture-centric, iterative and incremental. The four main use cases for the system are:

1. Load the hardware configurations file: This use-case is used to describe the hardware configurations like the hosts and network available for the real-time system.

2. Load the real-time software system specification file: This use-case is used to describe the real-time constraints and application specific parameters for a real-time system.

3. Start a real-time system: This use-case is used by the real-time system developer to start a real-time system via RM.

4. Maintain a feasible allocation: This use-case is used by RM to monitor and guarantee the real-time performance of a real-time system.

The user (or RM Installer) obtains and installs RM on a selected configuration of hosts and starts the middleware system. Then, the real-time system developer uses the “Start a real-time System” use-case to cause RM to find a feasible allocation for a Real-Time (RT) system and to start the applications of the RT system on the hosts indicated in the feasible allocation. As a next step, the dynamic real-time system initiates the “Maintain a feasible allocation” use-case to inform RM of its real-time performance and of its
resource needs. This causes RM to monitor the real-time performance as required by the real-time system developer. If the QoS is not met, then the RM performs a set of reallocation actions that will restore the required real-time performance.

The architecture of the DeSiDeRaTa QoS management system is shown in Figure 2.3. It consists of five subsystems: User Management, Real-Time System Management, Allocation Management, Resource Instrumentation and Control Management, and Specification File Management. These subsystems are used to implement the various resource management functions. Each subsystem is implemented as a set of classes

Figure 2.3. Architecture of DeSiDeRaTa
whose functionalities are closely related and an interface is provided for subsystem interactions.

The Specification File Management subsystem contains classes that parse user specification files and build the data objects that are used by the other subsystems.

The Resource Instrumentation and Control Management subsystem contains two service packages: Resource Monitor and Application Control. Resource Monitor has objects that gather resource usage and availability on a particular host. Application Control has objects that start and stops applications. The User Management subsystem contains classes that handle all user inputs.

The Allocation Management subsystem contains classes that handle all of the tasks relating to allocations. It finds feasible allocations using the information provided by Specification File Management and Resource Instrumentation and Control Management subsystems.

The Real-Time System Management subsystem contains classes that monitor and diagnose a Real-Time System. It gets the Real-Time System information from the Specification Management and Resource Instrumentation and Control subsystems. It then activates the Allocation Management to perform any changes needed in allocation.

The various subsystems are interdependent on each other. The subsystem dependency diagram is shown in Figure 2.4. All the communication between the different components of RM is via the communication middleware - CORBA. The TAO/ACE flavor of CORBA is used.
The User Management subsystem displays the allocation information and the real-time performance information for a real-time system. Thus, it is dependent on the Allocation Management and the Real-Time System Management subsystems to keep it informed of the current state of all real-time systems. The Real-Time System Management loads real-time system information (software) when it is activated via the software specification file. Allocation Manager loads resource information (hardware) and real-time software system information (software) files upon user's request. These
information sets are handled by the Specification File Management subsystem. Therefore Real-Time System Management and Allocation Management subsystems are dependent on the Specification File Management subsystem. The Real-Time System Management subsystem needs to monitor and diagnose the QoS for a real-time system. If there is a QoS violation, it relies on the services of Allocation Management for allocations to be performed. In order for Allocation Management to find and maintain feasible allocations it needs the services of the Resource Instrumentation and Control subsystem. The Resource Instrumentation and Control Subsystem further decomposes into two service packages: Resource Monitor and Application Control. Both of these packages depend on the Sun Solaris operating system to gather resource usage information and to start or stop an application.
CHAPTER 3

SERVICE LEVELS AND UTILITY BASED PARADIGM

For real-time applications that provide a variety of algorithmic alternatives that differ in their processing requirements and in the quality of their results, it is possible to defer the important decision of which algorithm to execute until run-time and to dynamically select the algorithm at run-time based on the available resources. This provides an alternate solution in addition to the DeSiDeRaTa’s approach of moving the applications onto another host in order to restore the QoS. The advantage of using such a technique is that the resources can be utilized more effectively, more data can be processed overall, and an overall higher benefit can be achieved from the system. These were the driving reasons to introduce the concept of service levels and utility within the DeSiDeRaTa resource management technique.
This is the structure of this chapter. The motivating example for this research is presented in section 3.1. Sections 3.2 and 3.3 describe, in detail, the concept of service levels and utility. The mathematical model and specification language were enhanced in order to accommodate these new features. The enhancements made are explained in sections 3.4 and 3.5. The algorithm used to determine the service level of an application is outlined in section 3.6.

### 3.1 Motivating Example

The motivation for this thesis comes from the problem statement for The Tenth International Workshop on Parallel and Distributed Real-Time System, 2002. Processing, archiving, accessing, visualizing and communicating Earth Science data from space and ground based sensors to the research community presents many challenging problems in distributed, real-time computing. In the future, NASA envisions a system where autonomous satellites will perform much of the event detection and response processing which ground-based stations presently perform.

The observing system of the future will include satellites in a variety of orbits. These will include a “sensor web” of small smart satellites with distributed processes that will dynamically respond to the commands of users interested in measuring important terrestrial events (e.g., a volcanic event).

A typical scenario with the sensor web is as follows: A scientist becomes aware of a “hotspot” on earth like an erupting volcano or a forest fire. There are many research satellites in orbit -- some of which are capable of observing this hotspot. Determining
which satellites can observe this hotspot, notifying each satellite of the event, and modifying the observation schedule of each satellite is a difficult and time-consuming job. It may take so long that the hotspot has disappeared. The scientist wishes to notify just one satellite in orbit, whichever comes over the nearest ground station first. Then the satellites perform "collaborative problem solving", working together autonomously to perform the notification, selection and rescheduling necessary to observe this hotspot with as many resources as possible, and send all the data to the scientist. The onboard real-time systems of the future should enable this kind of collaborative problem solving. While these satellites will be able to collaborate and come up with an observation schedule, they do not have the capability of managing the onboard computing and network resources. These satellites will not have a global view of the pool of computing resources at their disposal.

Changing the parameters of the onboard instruments dynamically based on the need is another area of research. These satellites form a network that consists of distributed processes that need to respond to perceived scientific events, the spacecraft environment, spacecraft anomalies and user commands. The requests and responses exhibit dynamic behavior. In order to handle such dynamic environments, a method is needed to guarantee the real-time quality of service constraints. The DeSiDeRaTa resource management approach is being enhanced to characterize the dynamic aspects of intra-constellation topologies and to accommodate the concept of service levels and utility.
The following scenario (from [3]) illustrates the operation of a constellation of earth observing (EO) satellites. Several sentinel satellites provide a line-of-sight view of all instrumented satellites in the constellation; each sentinel knows the precise location of all members in the constellation. The scenario involves the handling of a significant terrestrial event. A synthetic aperture radar satellite detects a volcanic event; the satellite brings the event into focus by rotating its instruments and altering its coverage area; onboard feature detectors analyze the data and assign priorities to different parts of the image; data compression is employed; to communicate the data to the ground station, another member of the constellation must be used as a relay, since the ground station is out of view; more important segments of data are sent back first, at high resolution; less important segments are sent back subsequently; scientists are alerted and assume control of the spacecraft; they direct its instruments for specific follow-up measurements.

To better equip the scientist with the capability of measuring significant terrestrial events with accuracy, the RM can be used. Consider the following scenario as shown in Figure 3.1. There are four satellite constellations, C1, C2, C3, and C4 each consisting of three satellites. Each of the satellites is busy performing some kind of earth observing activity. The scientist on earth becomes aware of an approaching tornado and is interested in monitoring the event. He sends an observation request to the RM. The RM uses the information in the observation request, the amount of resources available and the position of the various satellites to determine which satellite can observe the hot spot and sends a command to that satellite (in this example Satellite C). Now there can be a situation
where the satellite that is selected to observe the event becomes overloaded due to sudden increase in workload (for example, in the processing of a high resolution image). Due to the overload, the satellite misses a deadline and is not able to meet its real-time requirements. Under such circumstances, RM detects the QoS violation immediately and takes corrective action to restore the quality of service.

Figure 3.1. Motivating example scenario.
The Hierarchical Agent based Real-time Technology (HART) prototype system that is being developed at NASA efficiently models a typical satellite presented in the hot-spot example. Figure 3.2 shows the high level conceptual diagram of the prototype system.

The key components of the prototype system are:

**Camera Simulator (CS):** It sends the calibrated images to the IPA to be processed. It gets the data rate (the rate at which to send the images) from the CS GUI.

**IPA:** It is the image-processing agent that detects the cloud cover percentage present in an image.

**Download Agent:** It performs the transfer of data from the spacecraft to the ground whenever the network is available.

**Compressor Agent:** It performs the compression of the data based on the results from IPA, service level received from RM and the amount of space available in the buffer and then transfers the data to the buffer.

This prototype models a typical scenario on a satellite. Each of the key components is under RM control. RM is responsible for starting and stopping all the key components. All the components under RM control report their performance by sending a time-stamped event in each processing cycle to the RM. The camera simulator sends an image to the IPA at a specified rate. Upon receiving the image, IPA performs cloud cover analysis using the algorithm based on the service level received from RM and generates a report. IPA sends the cloud cover results to the compression agent. The compression
Camera sends Image Service Level and Image priority to IPA.

IPA Sends Start/Stop Timestamps, results, priority and Image Path to RM.

RM sends Service Level, Image Priority and Image Path to CA.

RM notifies Download Agent when network is available/unavailable.

User sets Frequency and Priority from GUI in Real-Time.

CA transfers Image to Spacecraft Simulated Buffer performing compression as specified by service level from RM.

CA Sends Start/Stop Timestamps to RM.

Compressor Agent

DA Sends Start/Stop Timestamps to RM.

Download Agent

DA - Performs the transfer from the spacecraft to the ground when the network is available. Also, deletes the downloaded files from the simulated buffer.

Simulated Data Transmitter

Buffer to Simulate an onboard system, with fixed size.

Simulated Data Receiver

Spacecraft Simulated Buffer

Figure 3.2. HART prototype system diagram
agent uses the utility algorithm to determine the best action based on the service level received from RM, the results obtained from IPA, the space available in the data storage buffer, and the time to next download window. If needed, the compression agent performs compression and transfers the data to the storage buffer. The download agent is responsible for transferring the data from the spacecraft to the ground whenever the network is available. It receives the data from the simulated buffer. This data is received at the ground station via the simulated data receiver.

3.2 Service Level and Utility Definitions

Service levels define the different Quality of Service (QoS) levels at which an application can operate. A level is a strategy of doing the application’s work and is characterized by a utility. Each level uses a different amount of resources and provides different utility to the user. Most of the multimedia applications fall under this model. Let us consider an example of a video application. Typical soft real-time approaches have the application use fewer resources by slowing it down, primarily by allowing it to miss its deadline by greater or lesser degrees. Although it may be acceptable for non-critical applications, such an approach might prove fatal for time and mission critical applications. For such applications, it is better to change the processing than to delay the result. Since QoS levels can characterize application specific soft real-time attributes, altering the frame rate, color depth, image resolution, size, or a combination of any of the variables can change the resource usage of video application. The choice of soft real-time
attributes and the corresponding QoS levels is up to the application developer and
depends on the specific requirements of the application.

Utility is defined as the user perceived benefit achieved by performing a certain
action with a particular fidelity within the given time constraints. It is a function of
quality of the output and is explicitly specified by the user. The utility value is defined in
the range $[0, 1]$.

### 3.3 DeSiDeRaTa Approach with Service Levels and Utility

DeSiDeRaTa is an adaptive resource management technique that provides
middleware services for distributed dynamic real-time systems that cannot be
categorized a priori. The DeSiDeRaTa system is based on the dynamic path paradigm
that provides automatic QoS assessment and resource allocation. A dynamic path
typically consists of sensors, actuators, and control software for filtering, evaluating and
acting. Path-level QoS specification is used by the adaptive resource allocator to
determine if the current configuration is achieving the desired QoS and to assist in
selecting new configurations to improve the QoS. A specification language is used for
specifying both static and dynamic attributes of an application.

Currently the DeSiDeRaTa system does not have any notion of service levels and
cannot accurately characterize the applications that are capable of running at multiple
levels of fidelity. Moreover, in the case where the resource manager determines a latency
violation, the only possible action that it can take to restore the QoS is to move the
application to another host. If the application is not movable, the resource manager
cannot do anything. Under such circumstances, the concept of service levels and utility can be of great benefit. If an application with service levels misses a deadline, we can lower the service level of the application and thus restore the QoS of the system. In some cases, moving the application to another host might cause additional overhead or missed deadlines, since the state of the application must be saved before killing and restarting the application on another host. This might also cause some loss of data. Under such circumstances, it is more beneficial to adjust the service level of the application rather than move it to a different host.

3.3.1 UML Model

The DeSiDeRaTa resource management system has been implemented using the Unified Modeling Language (UML), which is a well-defined software development tool. Its main characteristics are that it is use-case driven, architecture-centric, iterative and incremental. Introducing new concepts in DeSiDeRaTa affects its UML model, either modifying an existing use-case or introducing a new use-case. The concept of service level and utility results in modifying three existing use cases: “Load a real-time software system specification file”, “Start a real-time system” and “Maintain a feasible allocation” and introduces one new use-case called “Observe the service level and utility”. The two use cases, “Load a real-time software system specification file” and “Maintain a feasible allocation” had been modified significantly to handle the new concepts, and are discussed separately. The use-case model for the DeSiDeRaTa Resource Manager (RM) is shown in
Figure 3.3. The interaction of each actor with each RM use-case is depicted in the diagram.

![Use-case diagram](image)

**Figure 3.3. Use-case diagram**

### 3.4 Use Cases

#### 3.4.1 Load a real-time software system specification file

This use-case is used by the RM Operator (user) actor to load the software specification file, which provides all the information about the real-time constraints.
Load a real-time software system specification file: Requirements

Initial step-by-step description

Before this use-case can be initiated, the RM Installer has started the RM executable(s) and loaded the configuration specification files. The Real-Time System (RTS) Developer has profiled and described the characteristics of the RTS in a software specification file.

Pre-Condition

1. The RM Installer has started the RM executable(s) and loaded the configuration specification file.

2. The RTS Developer has profiled and described the characteristics (real-time deadlines, service levels) of the RTS in a software specification file.

Flow of Events

Basic Path

1. The RM Operator asks RM to load the software specification file using the interface.

2. RM reads and loads the software specification file. RM displays a message of success and waits for user input.

3. RM updates interface menus with the software system information.

Alternative Paths

1. In step 2, if the files are not found, RM displays the error back to the user and takes no action.
2. In step 2, if there are any syntax errors in loading the specification file, then RM displays the message through the interface and discards any stored information.

3. In step 2, if there are duplicate names found in the specification file, then RM displays the error back to the user and discards any stored information.

Post-Condition

RM has stored the information about each mission, application, path, and application found in the software specification file.

Load a real-time system software specification file: Analysis

Figure 3.4 describes the object interaction for the “Load a real-time software system specification file” use-case. The RM Operator through the RM Operator Interface specifies the RTS specification files to load (1). The Interface passes this information to the Software Parser (2). It creates an RTS Information object (3) and adds Mission, Software System, Path, Datastream, Datastream Element, Application Profiles, Application, Service Level Information, and Startup Information classes as introduced in the specification file (4, 5, 6, 7, 8, 10, 11, 12, 13). The Software System, Path, Datastream, Datastream Element, Application Profiles, Application, and Service Level Information are sent to the RM Operator Interface by the Software Parser (14). The RM Operator Interface updates its menus to allow the user to start the software systems in the software specification file.
Figure 3.4. Analysis collaboration diagram

Load a real-time software system specification file: Design

As seen from Figure 3.5 the RM Operator requests that RM loads a software specification file through the HCI and AllocationManagerInterface. The AllocationManager creates an RTSInformation class that parses the software specification file and stores the information in the Mission, SoftwareSystem, Path, CommunicationSubPath, Datastream, DatastreamElement, Application, StartupInformation, ServiceLevelInformation, and ApplicationProfile classes. AllocationManager then sends the specification file data to the HCIInterface so that HCI
can update its menus by updating the MissionInformationNode, PathInformationNode, and ApplicationInformationNode classes.

![Design class diagram](image)

**Figure 3.5. Design class diagram**

### 3.4.2 Maintain a feasible allocation

The use-case Maintain a Feasible Allocation is used by the Real-Time System (RTS) to provide an update to RM about its real-time performance and to request that RM check that any given allocation is still feasible. An allocation is feasible if the RTS's
performance meets its constraints. If RM discovers that the allocation is no longer feasible, it calls the diagnosis algorithm to find the reason, and the reallocation algorithm to find possible reallocation actions that restore or balance the RTS QoS. RM then performs a set of reallocation actions that should result in a feasible allocation.

**Maintain a feasible allocation: Requirements**

**Initial step-by-step description**

Before this use-case can be initiated, at least one RTS must be started. RTS applications must have been profiled.

The use-case is performed as follows:

**Pre-Condition**

1. At least one real-time software system has been started.

2. Real-time system applications have been profiled and the profile data should be available in the software spec file.

**Flow of Events**

**Basic Path**

1. RM receives and stores a timestamp event from the real-time system. The timestamp event includes the application ID (application name, application PID, host name), type (start/end), cycle (number), time, and workload information. RM stores the event information into the application event history.

2. If the event is an end event from the last application in a real-time software system path, and a start event from the first application of that path for the same cycle is
found in event history, RM computes the observed latency for that real-time software system path. RM then stores this information into the path’s state history.

3. RM obtains the required latency of the real-time software system path from the software specification file, and compares it with the observed latency.

4. If the observed latency for the path is worse than the required latency, the state for that path is marked as unhealthy in the path’s state history. RM then calls a pre-chosen diagnosis algorithm, which analyzes the applications' workload, latency and the profiles provided in the software specification file, to discover cause of the path's poor performance and to discover the names of the unhealthy applications. RM then calls a pre-chosen reallocation algorithm to find possible reallocation actions that can restore the QoS of the path.

5. RM then executes any of the following reallocation actions:

   a. Move an application to a different host. RM first kills the unhealthy application and its dependent applications. Then RM restarts the unhealthy application on the recommended host and restarts dependent applications on their current hosts.

   b. Decrease the service level of an application if RM does not find a suitable host. RM then notifies the application of the change in service level. The application uses the changed service level in the next cycle.

   c. Replicate an application. RM starts a new copy of the application on the recommended host.
6. RM checks observed performance for the path to see if its latency is below a minimum threshold relative to the required latency.

7. If yes, the algorithm identifies the causes of the path's very good performance and discovers the names of the applications with more than adequate resources. RM calls a pre-chosen reallocation algorithm to find possible reallocation actions that can balance the RTS QoS. RM then does one of the following actions:
   a. Moves an application to a different host.
   b. Increases the service level of an application. RM then notifies the application of the change in service level. The application uses the changed service level in the next cycle.
   c. Deletes a replica of the application.

Alternate Paths

1. In step 3, If RM does not find a matching event (for that cycle), it completes the use-case.

2. In step 4, if the performance is meeting the required QoS, RM does nothing and the use-case completes.

Post-Condition

1. Real-time software system quality of service (QoS) is improved.
Maintain a feasible allocation: Analysis

Figure 3.6. Analysis collaboration diagram

In figure 3.6, the RTS sends a timestamp event through the RTS Interface (1). This event could be of type START or DONE and contains the application PID, cycle number, current workload, and time. RTS Interface requests that RTS Manager computes application performance (2). RTS Manager gets the required latency from the RTS Information class to compute the performance of the RTS (3). RTS Manager stores the event data in the RTS Event class and searches for the matching start or end event for a particular cycle of an application (4) and updates the RTS State class with the cycle number, latency and health information of the application (5). If a violation of real-time requirements is detected, RTS Manager performs diagnosis to find unhealthy applications.
and requests reallocation of the applications (6,7). The Allocation Manager gathers the suitable hosts for an application from the RTS Information class (8), the resource information about a particular host from the Resource Information class (9), the allocation state information of a particular application from the Allocation State class (10), resource usage information on all hosts from the Resource Monitor class (11, 12, 13, 14), and network usage information from the Network Monitor class (15, 16, 17, 18, 19). The Allocation Manager makes reallocation decisions and asks the Application Control to start, stop or move the applications accordingly (20, 21, 22, 23). Allocation Manager then updates the Allocation State with the new host information depending on the reallocation action taken (24). The Allocation Manager will also change the service level of the application if necessary (25, 26).

*Maintain a feasible allocation: Design*

Figure 3.7 demonstrates the participating design classes and their associations. An RTS sends event data through the RTSInterface to RTSManager. RTSManager gives control to RTSMonitor which uses the RTSPath and RTSApplication classes to generate an RTSEvent, store it in RTSEventHistory, and then update the RTSSState and RTSSStateHistory classes. If QoS violations have occurred, then RTSDiagnosis analyzes the RTSSStateHistory and RTSSState classes to find the problem applications. RTSDiagnosis asks AllocationManager to perform a reallocation of the problem applications. AllocationManager needs the current AllocationState and the host and network resource performance metrics from resource monitors to create a new allocation
plan. It then asks AppControl to start/stop applications according to the new allocation plan. AllocationManager may also need to send new service levels to applications.

Figure 3.7. Design class diagram

3.4.3 Observe the Service Level and Utility

This use-case is used to observe the service level and utility of a particular application or real-time system.

Observe Service Level and Utility: Requirements

Initial step-by-step description

Before this use-case can be initiated the RM Operator has started at least one RTS with initial utility and service levels.

The use-case is performed as follows:
Pre-Condition

1. RM Operator has started at least one RTS with defined utility and service levels.

Flow of Events

Basic Path

1. The RM Operator selects the "View" menu on the interface.

2. The RM Operator drags down to "Utility".

3. The RM Operator chooses the RTS(s) to view the utility graph window for from the sub-menu under "Utility".

4. The interface gathers the utility data from RM for the RTS application to be graphed and graphs it.

5. The interface places labels of which service level is being used on the utility graph at regular intervals.

Post-Condition

The use-case ends with any of the following:

1. The RM Operator has started the graphing of an application's utility in a utility graphing window

2. The RM Operator has stopped the graphing of an application's utility in a utility graphing window

3. The application being viewed stops.
**Observe Service Level and Utility: Analysis**

Figure 3.8 describes the object interaction for Observe the Utility and Service Levels of an RTS Application. The RM Operator requests to observe the utility and service levels of an RTS application (1). The Interface requests the utility and service level data from Allocation Manager (2). Allocation Manager sends the utility and service level data to RM Operator Interface as the utility and service level data is updated within RM (3). The RM Operator Interface displays the utility and service level for RM Operator and updates it at regular intervals using the last utility and service level data it received (4).

![Figure 3.8. Analysis collaboration diagram](image)

**Observe Service Level and Utility: Design**

Figure 3.9 shows the interactions of participating design classes. Upon the request from RM Operator to view the utility and service levels, RMOperatorInterface displays the utility and service level for RM Operator. AllocationManager sends utility and service level data updates to RMOperatorInterface.
3.5 Mathematical System Model

In this section we present the system model to capture the dynamic path paradigm with the notion of utility. The complete specification for DeSiDeRaTa is given in [2,6], where a software subsystem is specified to include a set of applications, devices and paths. Using [2,6] as references, the system model has been further simplified for ease of implementation. The current model has been extended to incorporate the notion of service levels and utility. Each application has a set of acceptable service levels and the utility provided at each level. Table 3.1 describes the generic components of a real-time system. Each real-time system is described within a set of software systems SS. SS\textsubscript{i} describes a particular instance ‘i’ of a real-time system. A software system consists of
one or more applications and/or paths. ‘SS_i.P’ and ‘SS_i.A’ represent the set of all paths and the set of all applications in software system i respectively.

**Table 3.1: Generic Components**

<table>
<thead>
<tr>
<th>GENERIC COMPONENTS</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS</td>
<td>Set of all Software Systems</td>
</tr>
<tr>
<td>SS_i</td>
<td>Software System i</td>
</tr>
<tr>
<td>SS_i.P</td>
<td>Set of all Paths in Software System i</td>
</tr>
<tr>
<td>SS_i.A</td>
<td>Set of all Applications in Software System i</td>
</tr>
<tr>
<td>SS_i.P_j.A</td>
<td>Set of all Applications in Path j of Software System i</td>
</tr>
<tr>
<td>SS_i.P_j</td>
<td>Path j of Software System i</td>
</tr>
<tr>
<td>SS_i.A_j</td>
<td>Application j of Software System i</td>
</tr>
<tr>
<td>SS_i.P_j.A_k</td>
<td>Application k of Path j of Software System i</td>
</tr>
</tbody>
</table>

Table 3.2 specifies the path attributes. A path consists of one or more applications. The real-time performance of a path is monitored for each cycle of execution. Thus ‘c’ represents the current cycle number of the path (or application, if there is just one application in the path). The workload for a path is measured per cycle and is defined by workload (P, c). The required latency for a path is given by \( \lambda_{\text{REQ}}(P_i) \) and the observed latency of a real-time path for a given cycle is denoted by \( \lambda_{\text{OBS}}(P_i, c) \). In order to maintain the QoS for a system, \( \lambda_{\text{OBS}}(P_i, c) \leq \lambda_{\text{REQ}}(P_i) \).
Table 3.2: Path attributes

<table>
<thead>
<tr>
<th>PATH ATTRIBUTES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Current cycle number of path/application</td>
</tr>
<tr>
<td>workload(P_i, c)</td>
<td>Workload of path i for cycle c</td>
</tr>
<tr>
<td>$\lambda_{REQ}(P_i)$</td>
<td>Required latency of path i</td>
</tr>
<tr>
<td>$\lambda_{OBS}(P_i, c)$</td>
<td>Observed latency of path i for cycle c</td>
</tr>
</tbody>
</table>

Table 3.3 models the hardware attributes in a system. HS denotes the set of hardware systems. Each hardware system consists of a set of hosts.

Table 3.3: Hardware Attributes

<table>
<thead>
<tr>
<th>HARDWARE ATTRIBUTES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>HS</td>
<td>Set of all Hardware Systems</td>
</tr>
<tr>
<td>$HS_i$</td>
<td>Hardware System i</td>
</tr>
<tr>
<td>$HS_i(H)$</td>
<td>Set of all Hosts in Hardware System i</td>
</tr>
<tr>
<td>$HS_i(H_j)$</td>
<td>Host j of Hardware System i</td>
</tr>
<tr>
<td>$H(P_i,A_j, c)$</td>
<td>Host of Application j of Path i during cycle c</td>
</tr>
<tr>
<td>allocation(H_i)</td>
<td>The allocation of Host i (list of applications on host H_i)</td>
</tr>
<tr>
<td>allocation(P_i)</td>
<td>The allocation of Path i (list of application-host pairs)</td>
</tr>
</tbody>
</table>

Table 3.4 describes the Quality of Service (QoS) parameters for a real-time system. In this and the following tables the changes to the existing model are in bold type. Set of all QoS parameters in a software system is defined by $SS_i QOSP$. The service levels are associated with an application. Hence we model them as $SS_i A_j QOSU$, which defines the
set of service levels and utility associated for each level for the j\textsuperscript{th} application in the i\textsuperscript{th} software system. In order to manage the service levels, we need to remember the current service level of an application. This is denoted by $SS_i.A_j.QOS_{qc}$, where $q_c$ denotes the current service level. RM must make sure that the application satisfies at least its minimum service level requirements.

**Table 3.4: QoS Parameters**

<table>
<thead>
<tr>
<th>QoS Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SS_i.QOSP$</td>
<td>Set of all QoS parameters in software system I</td>
</tr>
<tr>
<td>$SS_i.QOSP_k$</td>
<td>QoS parameter k in software system I A tuple made up of a name, type, domain and resource_usage(x in domain).</td>
</tr>
<tr>
<td>$SS_i.A_j.QOSU$</td>
<td>A set of QoS levels and utility associated with each level for the j\textsuperscript{th} application in i\textsuperscript{th} software system. $QOSU = {(q_1, u_1), (q_2, u_2), \ldots}$</td>
</tr>
<tr>
<td>$SS_i.A_j.QOS_{qc}$</td>
<td>QoS level at which application $SS_i.A_j$ was running in cycle ‘c’</td>
</tr>
<tr>
<td>$\tau(SS_i,QOSP_k)$</td>
<td>Type of QoS parameter can be “Continuous”, “Discrete”.</td>
</tr>
<tr>
<td>Usage($SS_i.A_k, SS_i.QOSP_m$)</td>
<td>Usage of QoS parameter m of software system I with respect to application k. Usage values can be “IN”, “OUT”, or “ADJUSTABLE”.</td>
</tr>
</tbody>
</table>

Table 3.5 describes the utility model associated with the software system and path based on the available resources. The utility function is defined by the user and is specified in the software specification file. Given the available resources, the RM tries to maximize
the overall system utility. In order to do so, RM adjusts the service levels of the applications such that the global system utility is maximized.

**Table 3.5: Utility Model**

<table>
<thead>
<tr>
<th>UTILITY MODEL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>utility(SS_i.A_k, SS_i.QOSP, R)</td>
<td>The user-defined utility function of application k for a set of QoS parameters given a set of available resources R. Returns the maximum utility achievable.</td>
</tr>
<tr>
<td>utility(P_i.R)</td>
<td>The utility value associated with path P_i given a set of resources, R.</td>
</tr>
<tr>
<td>utility(SS_i, R)</td>
<td>The maximum utility of software system i given a set of available resources, R.</td>
</tr>
</tbody>
</table>

### 3.6 Specification Language and Grammar

The specification language is used to specify both the static and dynamic attributes of the system. This specification language is a programming-language-independent meta-language that describes the real-time QoS in terms of end-to-end paths, multi-level timing constraints, deterministic, stochastic and dynamic characteristics of environment-dependent paths, scalability and fault tolerance features of the paths and the application programs. A detailed description of the specification language can be found in [2]. In order to accommodate the concept of service levels and utility, the DeSiDeRaTa specification language has been enhanced with constructs to handle the new features.

Figure 3.10 shows an example of a service level block. The service levels are defined by the name of the service level and the utility value associated with it. A
complete example of the specification file can be found in Appendix A. The optional keyword **“Service Level”** defines the different levels of fidelity at which the IPA can run.

```
Application ipa {
  Service Level Algorithm1 {
    Utility 1.0;
  }
  Service Level Algorithm2 {
    Utility 0.15;
  }
  Service Level Algorithm3 {
    Utility 0.2;
  }
  Service Level Algorithm4 {
    Utility 0.25;
  }
  Service Level Algorithm5 {
    Utility 0.35;
  }
  Service Level Algorithm7 {
    Utility 0.5;
  }
  Service Level Algorithm10{
    Utility 0.7;
  }
  Service Level Algorithm20{
    Utility 0.8;
  }
}
```

**Figure 3.10. Software specification file**

This keyword is followed by an identifier, which is a unique name within an application block. Each service level is associated with a utility, which is a unique value (between 0 and 1) within an application block. **“Utility”** defines the benefit that the user will achieve by running the application at that service level. The applications have different resource requirements to achieve the expected QoS performance at a particular level. If the utility
decreases it is assumed that the resource usage decreases. For the example application (IPA), which detects the cloud cover percentage present in an image taken by a camera on a satellite, the service levels are based on the number of pixels that are skipped while processing an image. Each algorithm uses a different skip pixel rate and requires different amounts of CPU time to process an image. The utility is the measure of how accurate the result of cloud cover percentage detection is when a certain algorithm is used to process an image. The hardware configuration is specified in the hardware specification file. This file provides information about all the hardware resources available. An example of a hardware specification file can be found in Appendix A.

```xml
<application> → APPLICATION <ID>
    LEFT CURLY
    [<appQoSInfo>]*
    [<startupEnvironment>]*
    [<appServiceLevelInfo>]*
    [<appStartupInfo>]+
    [<appDependencyInfo>]? 
    RIGHT CURLY
```

**Figure 3.11. Grammar for Application**

The grammar for application is shown in Figure 3.11. The application block begins with a keyword APPLICATION followed by an identifier name (unique across all software systems) and a pair of curly brackets containing all the properties of the application. The <appQoSInfo> block describes the various quality of service parameters associated with the application. The service level threshold for an application is specified
within this block as shown in Figure 3.12. The keyword SERVICELEVEL THRESHOLD is followed by an identifier name (Name of the threshold service level). The <startupEnvironment> block contains the information about the environment variables (if any) to be set for the application. The <appServiceLevelInfo> block describes the different service levels of an application. The <appStartupInfo> block contains all the parameters needed to start an application. The <appDependencyInfo> block contains the startup and shutdown dependencies of the application.

The grammar for the service levels is shown in Figure 3.12. The service level block begins with a keyword SERVICE LEVEL followed by a unique identifier name and a pair of curly braces containing the utility value associated with that particular service level. The grammar has been developed using the BNF format.

![Figure 3.12. Grammar for Service Levels](image)

### Figure 3.12. Grammar for Service Levels

#### 3.7 Algorithm to Determine the Service Levels

This section describes the algorithm used in making the reallocation decisions based on service level information. The resource manager (RM) component responsible for making all the allocation decisions is the Allocation Manager (AM). These allocation
decisions include choosing the host and service level for each application. At start-up, the allocation manager tries to run the application at the highest service level so as to achieve maximum benefit.

The two assumptions on which this algorithm is based are:

1. The service levels are monotonically related to the CPU utilization. If the service level increases, then the application is using more CPU resource.
2. The utility associated with the service levels is mapped in the range [0,1].

The resource manager selects the service level for an application based on the available resources and benefit. There are various ways to determine the resource availability. We use VMSTAT to get the resource usage statistics. These statistics are collected every second by the Resource Monitor component of RM. CPU overload is determined by the incidence of the missed deadlines in the running application. CPU underutilization is determined by the CPU idle time. In the current RM, computing the moving average of CPU utilization over 10 cycles and then subtracting that value from 100 gives the CPU idle time. This determines the underutilization of the CPU. The algorithm to determine the service level is divided into two parts:

A. Initial Allocation Algorithm:

1. for all $A_i$ in SS.A
2. if(application has service levels)
3. assign highest service level;
Initially the allocation manager starts all applications at their highest service level.

During run-time if there is a latency violation, then it uses the run-time allocation algorithm.

B. Run-Time Allocation Algorithm:

_Diagnose unhealthy application and decrement service level_

1. if(latency violation)
2. determine unhealthy applications;
3. for all unhealthy applications
4. {
5.  getAppCurrentServiceLevel();
6.  decrement(service level of the application);
7. }

_When application requests its service level_

1. getAppCurrentServiceLevel();
2. getResourceUsage();
3. if(Current_CPU_idletime >= CPU_IDLETIME)
4.  increment(service level of the application)
5.  sendServiceLevel();

If there is a latency violation for a real-time system then the RM determines the unhealthy applications. For each unhealthy application, RM then decreases its service level by 1. When the application requests for its service level (at each processing cycle), the allocation manager gets the service level value set previously and gets the current resource usage information from the Resource Monitor. If the CPU idle time is greater than or equal to CPU_IDLETIME, RM increases the service level of the application by 1, and sends the service level to the application. The application will use this value for processing in the next cycle. The variable CPU_IDLETIME is set to 30%. This number
was determined after experimenting with various different values. Chapter 4 describes how we arrived at this number.

The above run-time algorithm was further modified to handle the service level threshold value. This approach provides the best of both worlds for the applications that have service levels. If there is a latency violation, then RM tries to reduce the service level of the application. If the application is running at the threshold service level, then RM will try to find a new host for the application and restart the application on the new host at the highest service level. This action can be taken only if the application is capable of running on multiple hosts. If it cannot find a new host, then it will lower the service level below the threshold value. The application requests its service level at each processing cycle.

C. Run-Time Allocation Algorithm with service level threshold:

_Diagnose unhealthy application and decrement service level_

1. if(latency violation)
2. determine unhealthy applications;
3. for all unhealthy applications
4. if(application has service level && (current service level < threshold service level))
5. {
6.   getAppCurrentServiceLevel();
7.   decrement(service level of the application);
8. }
9. else
10. {
11.   choosHost();
12.   if(newHost == oldHost)
13.     decrement(service level of the application);
14.   else
15.   {
16. kill the application;
17. restart application on new host;
18. update allocation state;
19. }
20. }

*When application requests its service level*

1. getAppCurrentServiceLevel();
2. getResourceUsage();
3. if(CPU_idletime >= CPU_IDLETIME)
4. increment(service level of the application)
5. sendServiceLevel();
CHAPTER 4

EXPERIMENTS AND RESULTS

This Chapter presents the experiments that were conducted to validate the approach presented in this thesis and the results derived from it. Section 4.1 gives the overview of the system that was used as test bed to validate the approach. Architecture of the test bed is given in section 4.1.1. Section 4.2 describes the experiments that were performed to validate the approach and the results derived from these experiments are discussed in section 4.3.

4.1 HART System Test Bed

The HART project is designed to unify the agent based computing paradigm with the theory of adaptive resource management for dynamic real-time systems. To demonstrate this capability, this project is prototyping a system by applying adaptive resource management and the adaptive real-time agent concept to perform onboard prioritizing of downlink data. It is targeted to meet the needs of future autonomous satellite to perform onboard event detection and response processing automatically. The
HART system prototype is being developed at NASA. This system served as a test bed to validate the concept of service levels and utility added to the adaptive resource management approach. Only the key components of the system were implemented for this test bed. The prototype system discussed in Chapter 3 is a more evolved version of this prototype.

The HART prototype consists of a Resources Manager (RM), an adaptive real-time agent (the Image Processing Agent – IPA), a Camera Simulator (CS) and a real-time Cloud Display (CD) system. Figure 4.1 shows the system diagram.

![Figure 4.1. HART high level conceptual diagram](image-url)
The resource manager (RM) monitors the performance of the image-processing agent (IPA) and ensures its quality of service requirements. The camera simulator simulates an onboard camera and sends calibrated images to the IPA at a rate specified by the RM. The IPA performs the cloud cover analysis of the image received from the camera simulator and makes real-time downlink decisions. The cloud display demonstrator allows the user to observe the performance of the IPA and view the downlink decisions made by the IPA. The data flow amongst the different components of the system is shown in Figure 4.2.
4.1.1 Architecture of the Test Bed prototype

Image Processing Agent: The Image Processing Agent (IPA) software uses the adaptive real-time agent concept to perform the decision-making function of prioritizing data for downlink. Currently the IPA uses a simplified NOAA’s cloud detection
algorithm to perform cloud coverage detection for onboard images. The IPA has the ability to switch to a different cloud detection algorithm to satisfy the onboard sciences preprocessing performance requirements. It also makes real-time decisions based on the image’s cloud percentage and sends back the data. Low-quality (high cloud percentage) science images could be discarded or compressed to free up space for newer science images and reduce bandwidth requirements. This will eliminate the need for scientists to preview thousands of low-priority images. Figure 4.3 shows the program flow for the IPA.

![Flow Chart](image_url)

**Figure 4.3. Program flow of the IPA**
Cloud Display: The cloud display (CD) system is designed to display real-time cloud mask and its related statistics. Figure 4.4 shows a sample output from the system. The image on the left is the original image and the one on the right is the processed image. The box at the bottom displays various statistics of the algorithm.

Camera Simulator: The Camera Simulator (CS) is designed to simulate the onboard camera. It generates calibrated image data and periodically sends it to the IPA based on the data rate specified. Figure 4.5 shows the program flow for camera simulator.
4.2 Experiments

The adaptive resource management approach presented in this thesis was validated on the prototype system, HART, developed by NASA. Two experiments were conducted to show that having service levels for applications provides an alternate way to restore QoS for the applications when other options like moving the application to another host or starting a replica to share the workload are more computationally expensive or even impossible. Sometimes moving an application from one host to another can result in the loss of small amounts of data. Applications with service levels provide an elegant solution for systems that may result in malfunctioning when small amounts of data are
lost. The service level of the application can be adjusted in order to function within the available resource constraints.

The experiments designed to validate this approach bring out the two points that are mentioned above. In the first experiment, we show that when the application is moved to another host, it loses one frame of data (an image for our example) but operates at the highest service level. Consider an example when the system is monitoring an important terrestrial event like a forest fire or an approaching tornado. In such cases, where the image data is being used for analysis, losing a single frame might give inaccurate results. In the second experiment, the capability to adjust service levels based on the available resources has been highlighted. Artificial load was created on the hosts to cause a latency violation. The resource manager then detects the violation and lowers the service level of the application to restore the QoS. When there are additional resources available, the resource manager increases the service level.

The experiments were conducted in the Laboratory for Intelligent Real-Time Secure Systems (LIRTSS) at Ohio University. Two Sun Ultra 10 workstations were used: Host ‘Water’ and Host ‘Patriot’. Dynamic scenarios challenging enough to force resource reallocation were run and the QoS performance in each case was recorded. A load generator program was used to create dynamic load on the hosts. The performance of the IPA was observed.

The experimental set-up is shown in Figure 4.6. The RM, camera simulator and IPA were started on host ‘Water’. The IPA and camera simulator were started via RM. The
Figure 4.6. Experimental Set-up: H/W: 2 Sun workstations N/W 100 Mbps

cloud display program was started on host ‘Patriot’. The load generator program was
started on host ‘Water’ to generate artificial load on the host.

4.2.1 Experiment 1:

The first experiment shows that moving the application to another host in order to
restore QoS causes system overhead for applications that are capable of running at
multiple levels of fidelity. There is a reasonable degree of trade off between missed
deadlines and a small degradation in the quality of the output.

In this experiment, the RM is started on host ‘Water’. IPA is made movable, i.e. it
is capable of running on multiple hosts. For this experiment, IPA can run on hosts
‘Water’ and ‘Patriot’. The camera simulator is not movable; it can run only on ‘Water’,
hence cannot be moved. The user sends a request to RM via the Human Computer Interface HCI to start the ImageProcAgent system. The RM gathers the real-time constraints for the system from the software specification file and selects an allocation plan for the real-time system. Initially, both IPA and camera simulator are started on host ‘Water’ since host ‘Water’ has the maximum resources. The cloud cover demonstrator is started on host ‘Patriot’. Figure 4.7 shows the output window of Application Control component of RM. We can see that the camera simulator and IPA were started successfully on host ‘Water’. The camera simulator is started with PID 17385 and IPA is started with PID 17386. RM starts IPA at the highest service level since there are enough resources available on the chosen host. Figure 4.8 shows the observed latency and service levels for IPA. It is evident that IPA is getting all the resources it needs and can satisfy its real-time constraints.

![Image](image.png)

**Figure 4.7.** IPA and camera simulator successfully started on Water.
In Figure 4.8, the left hand side y-axis shows the latency values. The required latency for IPA is 30 seconds. The right hand side y-axis shows the service levels for IPA. The service level is represented by the utility value associated with it. In order to increase the workload on the host water, a load generator program was started. This caused the IPA to miss a deadline. Figure 4.8 shows that IPA violates its real-time constraints for 12th processing cycle. This latency violation is detected by RM. This triggers RM’s diagnostic algorithm. RM needs to re-allocate resources in order to restore the QoS. RM stops IPA on host ‘Water’ (refer Figure 4.8 and 4.9). It chooses host ‘Patriot’ since that host is less loaded and moves IPA on to that host. This can be seen from Figure 4.10. We can see
that stopping and restarting IPA resulted in missing deadlines. In this experiment, a missed deadline would result in loss of an image. We can also see that IPA starts processing the image A1424286.L1900456. Thus the image A1424286.L1900442 sent by the camera simulator is lost when IPA is stopped and restarted.

![Figure 4.9. Application IPA is stopped.](image)

![Figure 4.10. IPA misses an image.](image)
Figure 4.8 shows that IPA was started successfully on host ‘Patriot’ at the highest service level and that it is satisfying all its real-time constraints.

4.2.2 Experiment 2:

The workload experienced by dynamic real-time systems is unpredictable and cannot be characterized a priori. Such conditions were simulated in our second experiment.

The two hosts modeled the two satellite constellations presented in the example scenario in section 3.1. RM was started on Host ‘Water’. In order to start a real-time system, the user needs to load the specification files first. Thus the user loads the hardware and software specification files for the real-time system IPA through the HCI. Once the specification files are loaded successfully, the user requests RM to start a real-time system ImageProcAgent. RM starts the IPA and camera simulator on host ‘Water’. The user starts the cloud display program on host ‘Patriot’. Initially, when the system is not overloaded, RM starts IPA at the highest service level. The graph shown in Figure 4.11 depicts the service level information and real-time path latency of IPA. From the utility plot, we can see that the application IPA is running at the highest service level Algorithm1, which provides the highest utility to the system.
Figure 4.11 Observed latency and service levels for IPA (with 40% load)

Figure 4.12 shows the cloud cover detection results. The left window shows the original image and the right window shows the processed image. The percentage cloud cover for the image is 56.52%. The IPA computes the utility of this image to the scientist and makes a downlink decision. The downlink decision for this image is to compress the data.
Figure 4.12. Cloud cover percentage with Algorithm 1.

A load generator program is started on the same host on which IPA is running to create artificial load on the CPU. It increases the load on CPU by 40 %. As the load on the host increases, IPA needs to compete for resources in order to complete its tasks. As a result the latency of IPA starts increasing (refer Figure 4.11). Due to the increased load, IPA fails to get the required resources. The performance of IPA degrades and it fails to meet its QoS requirements and causes a latency violation as seen in Figure 4.11. As a result, RM starts lowering the service level of IPA until the QoS is restored. From Figure
4.8, we can see that the service level of the application drops to Algorithm 4 and the QoS is restored.

The utility provided to the system at this level is much less than its maximum capability, but still it does not render the results useless, as the application can derive some useful information from it. The performance of IPA is degraded as seen in the Figure 4.13. The cloud cover algorithm uses significantly sparse image data to make the downlink decision. It provides a utility of 1.75 to the scientist and the action is to compress the data before downlink.

Figure 4.13. Cloud cover display with Algorithm 4.
Next we conducted the experiment again increasing the load generated to 60%. A load generator was started to produce 60% load on the CPU. We can see that it forces a violation and RM takes corrective action to restore the QoS. RM lowers the service level to the lowest, Algorithm 2. Figure 4.14 shows that the QoS is restored. Figure 4.15 shows the performance at the lowest service level.

![Observed Latency and Service Levels for IPA (with 60% Load)](image)

**Figure 4.14 Observed latency and service levels of IPA (with 60% load)**

Besides handling overload conditions, RM also monitors if any of the hosts are underloaded. If it finds that there are surplus resources available, it tries to increase the service level of applications that are not performing at their maximum capacity. If the load on the machine is reduced, then the RM increases the service level of the application.
since there are enough resources available. The RM increments the service levels in steps and keeps increasing the service level as long as the latency requirements are met.

Figure 4.11 and Figure 4.14 show that when the system is not overloaded, RM increases the service level of the application. Eventually when there are enough resources available, the service level returns to its maximum.

4.2.3 Experiment 3:
The third experiment shows the coordinated management of resource reallocation and application QoS level adaptation. The service level threshold value specified by the user determines the tradeoff between moving an application to another host and reducing its

![Figure 4.15: Performance at the lowest service level.](image)
service level. In this experiment, IPA was made capable of running on both host ‘Patriot’ and host ‘Water’. The camera simulator is started on host ‘Water’. The cloud display program is started on host ‘Patriot’. Initially RM starts IPA on host ‘Water’ at the highest service level. This can be seen from Figure 4.16. In order to simulate sudden workload, we start a load generator program on host ‘Water’. Due to this, IPA needs to compete for resources and when it does not get the required resources for it’s processing, it misses a deadline. This latency violation is detected by RM. As a corrective action, RM lowers the service level of IPA and IPA stabilizes at service level 5 (Algorithm 4) as can be seen from the Figure 4.16. Now we increase the load on host ‘Water’ by starting another load generator. We see that this causes IPA to miss a deadline. RM needs to take corrective action so as to restore the QoS of IPA. IPA is already operating at the threshold service level. Thus RM finds a less loaded host (Patriot) for IPA and moves it to that host. It starts IPA at the highest service level on host ‘Patriot’ (refer Figure 4.16).
Figure 4.16. Performance of IPA with modified run-time allocation algorithm

4.2.4 Experiment 4:

This experiment was conducted to determine the approximate value for CPU idle time. This value is used to determine the CPU underload and is used to increase the service level when additional CPU is available.

Figure 4.17 shows the performance of IPA with various CPU idle time values. The x-axis shows the processing cycle and the y-axis shows the different service levels of IPA based on the utility value. From the graph, we can see that at 20% CPU idle time, we can never achieve a stable service level. At 25% CPU idle time, there is instability when RM tries to increase the service level. For the 30% CPU idle time line, there is a smooth
degradation of service levels when there is a latency violation as well as increase in service level when there are enough resources available. For the 35% line, even though it behaves the same way as 30%, the time taken to increase the service levels is much more.

![Service Levels for IPA with varying CPU idle time](image)

**Figure 4.17. Performance of IPA service levels with varying CPU idle time**

### 4.3 Results

Based on the above experiments, we can say that the concept of having service levels is useful for applications that have multiple QoS levels. The real-time system can still derive useful information from the application by compromising on the quality of the output. In experiment 1, IPA was made movable onto multiple hosts. While moving the application to another host, there is a very short period of time loss when the
application is killed and restarted. This results in missing the deadline. For real-time systems, every second is critical. Thus it is quite possible to miss some data when the application misses a deadline when it is killed and restarted. This could cause a potential overhead for the system. Thus there is a trade off between missing some of the data and performing at the highest quality of service level. For the particular example in our experiment, the application IPA gives fairly accurate results even at a lower service level. From experiment 2, we can see that even though RM lowered the service level of IPA when the host became unhealthy or overloaded, IPA could process the image with its deadline and provide useful results to the user. Thus, it is more beneficial to lower the service level of the application than to move it to another host. From experiments 1 and 2 we can see that the user has to compromise either on quality or on missed deadlines in order to meet its real-time constraints. The third experiment highlights the trade off between missing deadlines and degraded quality. By using a user defined threshold to determine the trade off RM provides a more fine grain control over managing real-time applications.
CHAPTER 5

CONCLUSIONS AND FUTURE WORK

This thesis presents a novel approach for coordinated management of resource allocations and application QoS level adaptation for hard and soft real-time systems. It provides a method to accommodate soft real-time systems in the DeSiDeRaTa resource management system, which is primarily for hard real-time systems. This thesis introduced the concepts of service levels and utility in order to provide fine-grain control in managing the real-time systems. The service levels act as knobs that are controlled by DeSiDeRaTa in order to effectively adapt the applications to the amount of resources available.

Two experiments were designed to test the validity of this approach. The design of the experiments was based on the two salient points projected in this thesis:

1. Having service levels for an application provides an alternate way to restore the QoS for the applications when other options like moving the application to
another host or starting a replica to share the workload are more computationally expensive or even impossible.

2. Sometimes moving an application may result in missing deadlines that can cause loss of small amount of data. Adjusting the service levels to adapt the application to the available resources provides an elegant solution for applications that may result in malfunctions when small amounts of data are lost.

Experimental results validate the effectiveness of the service level approach. In the case of applications with multiple levels of fidelity, it is better to lower the service level of an application when there is a QoS violation. Moreover, for applications that are not movable, it provides an elegant way to handle the performance degradation since the degradation is only in terms of quality of the output. The real-time constraints are satisfied and the results are delivered in a timely manner.

Dynamic, distributed real-time systems are widely used in mission-critical environments. Therefore, it is imperative to pursue research in the area of quality of service management of such systems. The work presented in this thesis provides an insight into how the service levels can be used to characterize soft-real time systems and applications that are capable of running at multiple levels of fidelity. In this thesis we assume that the service levels are discrete. But in most cases, it is easier to specify the service levels as a continuous function and then use interpolation to determine the value at a particular point. This opens up a whole new world to be explored in the area of characterizing soft real-time systems. Further research can be done to find out how to use
continuous service levels and the interpolation techniques that can be employed to accurately capture the quality of service achieved by running the applications at different levels of fidelity. This thesis only explored the service levels in one QoS dimension, i.e. CPU resources. However, this can be extended to work for multiple QoS dimensions by considering other types of resources like network bandwidth, memory and storage. The trade-offs between all these dimensions need to be determined and defined.

Another interesting research area is in the field of resource reservation. There are certain applications that are capable of monitoring themselves. Such applications are capable of adapting themselves to changes in the availability of resources. These applications only have a local perspective; hence they do not have knowledge on how the decisions made impact the other applications in the system. Thus, it is beneficial to have these applications make a resource reservation request to a higher-level resource manager. In turn it would grant or deny the request based on real-time constraints. Determining the policies to grant or deny the request would form a potential avenue of research in this area.
BIBLIOGRAPHY


APPENDIX A

1. Example Hardware Configuration Specification File
// Sample Hardware Spec File

Hardware System A {
    Host patriot {
        Type "sun";
        Memory 384;
        CPUspeed 433;
    }
    Host water {
        Type "sun";
        Memory 384;
        CPUspeed 433;
    }
    Host secure-rm {
        Type "sun";
        Memory 384;
        CPUspeed 433;
    }
    Host america {
        Type "pentium";
        Memory 256;
        CPUspeed 300;
    }
}

2. Example Real-Time Software System Specification File
// Sample Software Spec File

Software System ImageProcAgent {
    Path ipa {
        Connectivity {
            (camera:ipa)
        }
        RealTimeQoS {
            Deadline 30.00;
        }
    }
    Application camera {
        Environment DISPLAY="water.ece.ohiou.edu:0.0",
        PATH="/home/hart/ipa_may02/ipa/"
        Startup solaris {
            ...
Working Directory "/home/hart/ipa_may02/ipa/"
Executable Name "camSim"
Arguments "camServer", "35"
Host water;
}
}

Application ipa {
  Environment DISPLAY="water.ece.ohiou.edu:0.0",
  PATH="/home/hart/ipa_may02/ipa/"

  Service Level Algorithm1 {
    Utility 1.0;
  }
  Service Level Algorithm2 {
    Utility 0.15;
  }
  Service Level Algorithm3 {
    Utility 0.2;
  }
  Service Level Algorithm4 {
    Utility 0.25;
  }
  Service Level Algorithm5 {
    Utility 0.35;
  }
  Service Level Algorithm7 {
    Utility 0.5;
  }
  Service Level Algorithm10 {
    Utility 0.7;
  }
  Service Level Algorithm20 {
    Utility 0.8;
  }

  Startup solaris {
    Working Directory "/home/hart/ipa_may02/ipa/"
    Executable Name "ipa"
    Arguments "ImageProcAgent:ipa", "-ORBInitRef",
    "NameService=corbaloc:iiop:water:9878/NameService"
    Host water;
  }
  Dependencies {
    Startup camera;
    Shutdown camera;
  }
}
}

// Sample Software Spec File

Software System ImageProcAgent {

    Path ipa {
        Connectivity {
            (camera:ipa)
        }

        RealTimeQoS {
            Deadline 30.00;
        }
    }

    Application camera {
        Environment DISPLAY="water.ece.ohiou.edu:0.0",
        PATH="/home/hart/ipa_may02/ipa/";
        Startup solaris {
            Working Directory "/home/hart/ipa_may02/ipa/";
            Executable Name "camSim";
            Arguments "camServer", "35";
            Host water;
        }
    }

    Application ipa {
        ServiceLevel Threshold Algorithm4;
        Environment DISPLAY="water.ece.ohiou.edu:0.0",
        PATH="/home/hart/ipa_may02/ipa/";

        Service Level Algorithm1 {
            Utility 1.0;
        }

        Service Level Algorithm2 {
            Utility 0.15;
        }

        Service Level Algorithm3 {
            Utility 0.2;
        }

        Service Level Algorithm4 {
            Utility 0.25;
        }

        Service Level Algorithm5 {
            Utility 0.35;
        }

        Service Level Algorithm7 {
            Utility 0.5;
        }

        Service Level Algorithm10 {

            }
Utility 0.7;
}

Service Level Algorithm20 {
    Utility 0.8;
}

Startup solaris {
    Working Directory "/home/hart/ipa_may02/ipa/";
    Executable Name "ipa";
    Arguments "ImageProcAgent:ipa", "-ORBInitRef", "NameService=corbaloc:iiop:water:9878/NameService";
    Host water;
}

Dependencies {
    Startup camera;
    Shutdown camera;
}

}

Software Specification file for load generator (lg.spec)

// Sample Software Spec File

Software System lg1 {

    Path myPath1 {
        Connectivity{
            (lg1)
        }

        RealTimeQoS {
            Deadline 1.0;
        }
    }

    Application lg1 {
        Autostart 1;
        Start Delay 0.0;
        Multiplicity 0;

        Environment DISPLAY="water.ece.ohiou.edu", PATH="/home/hart/";

        Startup solaris {
            Working Directory "/home/hart/";
            Executable Name "lg";
            Arguments "20"
            Host water;
        }
    }

}
Software System lg2 {

Path myPath2 {
    Connectivity{
        (lg2)
    }

    RealTimeQoS {
        Deadline 1.0;
    }
}

Application lg2 {
    Autostart 1;
    Start Delay 0.0;
    Multiplicity 0;

    Environment DISPLAY="water.ece.ohiou.edu", PATH="/home/hart/";

    Startup solaris {
        Working Directory "/home/hart/";
        Executable Name "lg";
        Arguments "20";
        Host water;
    }
}
}

Software System lg3 {

Path myPath3 {
    Connectivity{
        (lg3)
    }

    RealTimeQoS {
        Deadline 1.0;
    }
}

Application lg3 {
    Autostart 1;
    Start Delay 0.0;
    Multiplicity 0;

    Environment DISPLAY="water.ece.ohiou.edu", PATH="/home/hart/";
Startup solaris {
    Working Directory "/home/hart/";
    Executable Name "lg";
    Arguments "20";
    Host water;
}
}

Software System lg4 {
    Path myPath4{
        Connectivity{
            (lg4)
        }
        RealTimeQoS {
            Deadline 1.0;
        }
    }
    Application lg4 {
        Autostart 1;
        Start Delay 0.0;
        Multiplicity 0;
        Environment DISPLAY="water.ece.ohiou.edu", PATH="/home/hart/";
        Startup solaris {
            Working Directory "/home/hart/";
            Executable Name "lg";
            Arguments "20";
            Host water;
        }
    }
}
APPENDIX B

Allocation Manager Code

/*****************************************************************************/
// Function: startApp
// Purpose: Start the given application on the given host and returns the process ID of the application
// Parameters: string rtsName - name of the RTS
// string hostName - name of the host to start the application on
// Application *app - the application to start
// Time Analysis: O(e + v) where e = the number of environment variables needed to be put into a string
// and v = the size of the argument vector to be put into a string
/*****************************************************************************/

bool AllocationManager::startApp(string rtsName, Application *app, AllocationStateApp *appState) {
    if (debug)
        cout << "DEBUG: Entering AllocationManager::startApp\n";

    //Code added by Shikha 06/27/02
    //Assign the service level for the application before starting it
    // If application has service levels, then set service level
    if(!(app->GetServiceLevelInfoList().empty)){
        assignServiceLevel(app);
    }

    // Set up CORBA communication with Application Control
    // (hostnameApplicationControl)
    string hostName = appState->getHostName();
    string serverName = hostName + "ApplicationControl";
    DSDRT::AppData appData; // CORBA Application Information

    if (debug)
        cout << "\tAttempting to connect to " << serverName << endl;

    try {
        // Become a client of the Application Control server
        DSDRT::ClientINIT clientORB(orb_argc, orb_argv);
        DSDRT::ServiceINIT<DSDRT::AppControlInterface>
            acServer(serverName.c_str(), clientORB.getNamingContext());
    }
if (debug)
    cout << "\tConnected to " << serverName << endl;

// Get the Host data
appData.rtsName = rtsName.c_str();
appData.appName = (app->GetName()).c_str();
appData.startupDelay = app->GetStartupDelay();

// Create the environment string in format:
// var1=value1 var2=value2

string environment = "";
map<string,string> envPairs = app->GetEnvironmentPairs();
map<string,string>::iterator envIterator;
envIterator = envPairs.begin();
while (envIterator != envPairs.end()) {
    environment = environment + envIterator->first + "="
    + envIterator->second + " ";
    envIterator++;
}
appData.environment = strdup(environment.c_str());

// Gather the Startup Information
vector<StartupInformation> startupInfo = app->GetStartupInfoList();
string argumentVector = "";
if (startupInfo.size() > 0) {
    appData.workingDirectory = startupInfo[0].executablePath.c_str();
    appData.executableName = startupInfo[0].executableName.c_str();

    // Create space-delimited argument vector with executable name
    first
    argumentVector = "";
    if (startupInfo[0].argumentList.size() > 0) {
        unsigned int i;
        for (i=0; i<startupInfo[0].argumentList.size()-1; i++) {
            argumentVector = argumentVector +
            startupInfo[0].argumentList[i]
            + " ";
        }
        argumentVector = argumentVector + startupInfo[0].argumentList[i];
    }
    appData.argumentVector = argumentVector.c_str();

    if (debug)
        cout << "\tCalling startApp at Application Control\n";

    // Start the application with Application Control
    acServer->startApp(appData);
if (debug) {
    cout << "\t** Gathering App Data from Application Control\n"
    << "\t\t-- rtsName = " << rtsName << endl
    << "\t\t-- appName = " << appData.appName << endl
    << "\t\t-- hostName = " << hostName << endl
    << "\t\t-- startupDelay = " << appData.startupDelay << endl
    << "\t\t-- workingDirectory = " << appData.workingDirectory
    << endl
    << "\t\t-- argumentVector = " << appData.argumentVector
    << endl
    << "\t\t-- environment = " << appData.environment << endl
    << "\t\t-- processID = " << appData.processID << endl;
}
else {
    cout << "ERROR: AllocationManager::startApp\n"
    << "Could not find startup information for "
    << appData.appName << endl;
    return (false);
}

} catch (CORBA::Exception &e) {
    ACE_PRINT_EXCEPTION(e, ");
    return (false);
}
} catch (...) {
    cerr << "Error: Internal CORBA error in function "
    << "AllocationManager::startApp\n"
    << "Application was not started!\n";
    return (false);
}

if (appData.processID <= 0) {
    cout << "ERROR: Application " << appData.appName << " has processID "
    << appData.processID << endl
    << "Application was not started!\n";
    return (false);
}

cout << "Application " << appData.appName
    << " was started on host " << hostName
    << " with PID " << appData.processID << endl;

appState->setProcessID(appData.processID);
return true;
}

//=====================================================================
//=
Function: stopApp

Purpose: Stop the given application and update its state

Parameters: string rtsName - name of the RTS
            string hostName - name of the application to stop
            AllocationStateApp *allocation - the allocation of
            the application

Time Analysis: O(1)

bool AllocationManager::stopApp(string rtsName, Application *app,
                                AllocationStateApp *appState) {
    if (debug)
        cout << "DEBUG: Entering AllocationManager::stopApp\n";

    // Set up CORBA communication with Application Control
    // (hostnameApplicationControl)
    string appName = app->GetName();
    string serverName = appState->getHostName() + "ApplicationControl";
    DSDRT::AppData appData; // CORBA Application Information

    if (debug)
        cout << "\tAttempting to connect to " << serverName << endl;

    try {
        // Become a client of the Application Control server
        DSDRT::ClientINIT clientORB(orb_argc, orb_argv);
        DSDRT::ServiceINIT<DSDRT::AppControlInterface,
        DSDRT::AppControlInterface_var> 
        acServer(serverName.c_str(), clientORB.getNamingContext());

        if (debug)
            cout << "\tConnected to " << serverName << endl;

        // Set the Host data
        appData.rtsName = rtsName.c_str();
        appData.appName = appName.c_str();
        appData.processID = appState->getProcessID();

        if (!acServer->stopApp(appData)) {
            cout << "\tApplication " << appName << " could not be
        stopped.\n";
            return (false);
        }

        // If application was stopped successfully, then reset the service
        level.
        app->SetServiceLevelIndex(-1);
    }

}
} catch (CORBA::Exception &e) {
    ACE_PRINT_EXCEPTION(e, "");
    return (false);
} catch (...) {
    cerr << "Error: Internal CORBA error in function "
    << "AllocationManager:startApp\n"
    return (false);
}

return (true);

// Function: maintainRTS
// Purpose: This function is invoked through the Allocation Manager Interface when an RTS Path Manager detects a latency violation. It is responsible for taking corrective action to restore quality of service by reallocating applications that are causing the latency violations.
// Parameters: string rtsName - name of the RTS with poor QoS
//              string pathName - name of the path of RTS
//              string appName - name of the application in the path suspected of the poor QoS
//              string hostName - name of the host the application is running on.
//              long processID - processID of the application
// Time Analysis: O(1) because we do nothing currently!

bool AllocationManager::maintainRTS(string rtsName, string pathName, string appName, string hostName, long processID)
{
    if (debug)
        cout << "DEBUG: Entering AllocationManager::maintainRTS\n";

    // Find the needed information
    SoftwareSystem *rts = rtsInformation->GetSoftwareSystem(rtsName);
    Application *app = rtsInformation->GetApplication(appName);
    AllocationStateRTS *rtsState = allocationState->getRTSAllocation(rtsName);
    vector<ServiceLevelInformation> sLInfo =
        app->getServiceLevelInfoList();
// If application has service levels, then lower service level
if (!(sLInfo.empty()) &
    ((app->GetServiceLevelIndex()) < (app->GetServiceLevelThreshIndex()))) {
    if (debug)
        cout << "\t AllocationManager::assignServiceLevel being called\n" << "\t from maintainRTS \n";
    assignServiceLevel(app);
} else {
    string newHostName = chooseHost(app);

    if (newHostName == "") {
        cout << "Error in choosing host. No allocation action taken.\n";
        return (false);
    }

    if (hostname == newHostName) {
        // Same host chosen so try to lower service level
        cout << "The application " << appName << " is already allocated " << "to the host " << hostname << " .\n";
        if (sLInfo.empty())
            assignServiceLevel(app);
    } else {
        // Stop application and it's dependencies
        if (!stopAppWithDependencies(appName, rts, rtsState)) {
            cout << "Error stopping application " << appName << " .\n";
            cout << "RTS " << rtsName << " may not be running properly.\n";
            return (false);
        }

        // Update allocation with new host and process ID
        rtsState->updateAllocation(appName, newHostName, -1);

        // Restart all stopped applications
        // Loop through and start all applications
        vector<string> appNameList = rts->GetApplicationNameList();
        bool success = true;
        for (unsigned int i = 0; i < appNameList.size(); i++) {
            if (debug)
                cout << "\t" << appNameList[i] << endl;
            success = startAppWithDependencies(appName, rts, rtsState);
            if (!success) {
                stopRTS(rtsName);
                return (false);
            }
        }
    }
}
} // End looping through applications (for)

if (!success) {
    cout << "Error restarting stopped applications.\n";
    cout << "RTS " << rtsName << " may not be running properly.\n";
    return (false);
}

return (true);

short AllocationManager::determineServiceLevel(string rtsName,
                                                  string appName){
    if (debug)
        cout << "DEBUG: Entering AllocationManager::determineServiceLevel\n";

    // Get the application object
    Application *app =
        rtsInformation->GetApplication(appName); // should take rtsName in future

    if (!app) {
        cout << "ERROR: Could not find application " << appName << ".\n";
        return 0;
    }

    // Get the service level index of the object
short serviceLevel = app->GetServiceLevelIndex(); // The index ranges from 0 to 
// (number of service levels - 1)

if(debug)
    cout << "DEBUG: In AllocationManager::determineServiceLevel\n" 
    << " The service level index is " << serviceLevel << endl;

if (serviceLevel < 0) {
    cout << "ERROR: Could not determine service level.\n";
    return 0;
}

if(serviceLevel != -1){
    // Need to find the host on which the application is running
    // and then get the resource usage information for that host.
    // Then we can increase the service level if possible.

    AllocationStateApp *appState = allocationState-
    >getAppAllocation(rtsName, appName);
    string hostName = appState->getHostName();

    // Connect to resource monitor to get the current resource usage on
    // that host.

    string serverName = hostName + "ResourceMonitor";

    try {
        // Become a client of the Resource Monitor server
        DSDRT::ClientINIT clientORB(orb_argc, orb_argv);
        DSDRT::ServiceINIT<DSDRT::ResourceMonitor, 
        DSDRT::ResourceMonitor_var>
        resourceServer(serverName.c_str(), clientORB.getNamingContext());

        // Get the Host data
        DSDRT::HostData hostData; // CORBA Host Information
        hostData.hostName = hostName.c_str();
        resourceServer->getHostResources(hostData);

        if(debug){
            cout << "\t** AllocationManager::determineServiceLevel\n" 
            << "\t** Gathering Host Data from Resource Monitor\n" 
            << "\t\t-- hostName = " << hostData.hostName << endl 
            << "\t\t-- cpu = " << hostData.cpu << endl 
            << "\t\t-- average_cpu = " << hostData.average_cpu << endl 
            << "\t\t-- memory = " << hostData.memory << endl 
            << "\t\t-- average_memory = " << hostData.average_memory << 
            << "\t\t-- contextSwitch = " << hostData.contextSwitch << 
            << "\t\t-- average_contextSwitch " 
            }
if (hostData.average_cpu >= CPU_IDLETIME) {
    if (serviceLevel == 0) {
        // We are running at the highest service level. And so cant go
        // higher.
        if (debug)
            cout << "\tThe service level set by Allocation Manager is:
                 << serviceLevel + 1 << endl;
        return (serviceLevel + 1);
    }
    else { // Increase the service level.
        app->SetServiceLevelIndex(serviceLevel - 1);
        if (debug)
            cout << "\tIncreasing the service level\n"           << "\tThe service level set by Allocation Manager is:
                 << serviceLevel << endl;
        return (serviceLevel);
    }
}
else{
    if (debug)
        cout << "\tThe service level set by Allocation Manager is:
                 << serviceLevel + 1 << endl;
    return (serviceLevel + 1);
}

} catch (CORBA::Exception &e) {
    ACE_PRINT_EXCEPTION(e, "");
    return 0;
} catch (...) {
    cerr << "Error: Internal CORBA error in function "
         << "AllocationManager:determineServiceLevel\n"
         << endl;
    return 0;
}
}

/****************************************************************************
 ///
 /// Function: assignServiceLevel
 ///
 /// Purpose: Find a suitable service level for the application.
 ///
 /// Parameters: Application *app - Application to find service
 /// level for
 ///
 */
bool AllocationManager::assignServiceLevel(Application *app)
{
    vector<ServiceLevelInformation> slList;
    int slIndex = 0;

    if (debug)
        cout << "DEBUG: Entering AllocationManager::assignServiceLevel\n";

    // Get the service levels available for the application
    slList = app->GetServiceLevelInfoList();

    // If there are service levels, choose one
    if (!slList.empty()){
        //This is just the debug information.
        if (debug){
            cout << "The number of service levels for this application is : 
                  " << slList.size() << endl;
            vector<ServiceLevelInformation>::iterator m;
            for (m=slList.begin(); m!=slList.end(); m++) {
                cout << "\tService Level " << m->GetName() << endl;
            }
        } // Check to see if this is the first time to set service level
        if (app->GetServiceLevelIndex() == -1){
            // Use the highest service level for now.
            // The service level list is sorted in descending order of utility
            app->SetServiceLevelIndex(slIndex);
        } else {
            slIndex = app->GetServiceLevelIndex();
            if (debug)
                cout << "\tCurrent service level index is : " << slIndex << endl;

            //Lower the service level if possible
            if (slIndex != int((slList.size() - 1)))
                app->SetServiceLevelIndex(slIndex+1);
        } // If (debug)
cout << "\tNew service level index is :" << app-
>GetServiceLevelIndex() << endl;

} else {
    if (debug)
        cout << "ERROR: In AllocationManager::assignServiceLevel
               Got an empty service level list.\n";

        return 0; //if no service levels found.
    }

return 1;