DESIGN OF A RESOURCE MANAGEMENT SERVICE FOR THE QUALITY-BASED ADAPTIVE RESOURCE MANAGEMENT ARCHITECTURE

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

David T. Fleeman

November 2006
This thesis entitled

DESIGN OF A RESOURCE MANAGEMENT SERVICE FOR THE QUALITY-BASED
ADAPTIVE RESOURCE MANAGEMENT ARCHITECTURE

by

DAVID T. FLEEMAN

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Lonnie R. Welch

Professor, Electrical Engineering and Computer Science

R. Dennis Irwin

Dean, Russ College of Engineering and Technology
The Quality-based Adaptive Resource Management Architecture (QARMA) consists of a framework for describing resource management solutions and a collection of CORBA-based middleware services for the management of distributed, real-time systems. The framework can be used to: (1) characterize existing resource management architectures and tools and (2) assist in integrating existing tools into coherent resource management solutions. The middleware components are an instantiation of a resource management solution based directly on the framework.

The main contributions of this thesis include an analysis of the information life-cycle in a resource management system, a collection of algorithmic models that serve as a basis for resource allocation algorithms in the QARMA middleware, and a description of the greedy algorithm used by the Resource Management Service. Experimental results demonstrate that QARMA can control existing application systems and can be integrated with existing management middleware.
ACKNOWLEDGMENTS

To my wife, Dan, thank you for your constant love, encouragement and sympathy. To my parents, thank you for your love and encouragement. Your willingness to sacrifice so that I could attend college has made a better life possible for my own family. To my child, may your life be filled with blessings and may you grow in spirit and in truth.

To Dr. Welch, thank you for your deep respect and trust in my character. Your sponsorship and trust in my abilities put me in a position to learn many things that most graduate students do not have the opportunity to learn. To Dr. Juedes, Dr. Ostermann and Dr. Chelberg, thank you for your contributions to my education and research at Ohio University.

To the former students of our resource management team, your friendship is always remembered. Special thanks to Matthew Gillen whose technical ability brought QARMA from concept to source code. Good luck to all of you in your future endeavors.

To our collaborators, it was a truly fulfilling experience working with esteemed universities and companies on a variety of projects. Special thanks to Washington University in St. Louis, the University of Rhode Island, Vanderbilt University, Lockheed Martin Advanced Technology Labs, Raytheon, BBN Technologies, the NASA Goddard Space Flight Center, and the Naval Surface Warfare Center Dahlgren Division.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Abstract</strong></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Acknowledgments</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>List of Tables</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td><strong>Introduction</strong></td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td><strong>Related Work</strong></td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td><strong>The QARMA Architecture</strong></td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Generic Resource Management Components</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Quality-based Adaptive Resource Management Architecture (QARMA)</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>A Real Application Example</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Architecture Versus Information</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td><strong>The QARMA Information Layer</strong></td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>The System Model</td>
<td>28</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Resource Configuration</td>
<td>28</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Software Configuration</td>
<td>31</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Dynamic State Information</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Information Representations</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td><strong>Algorithmic Models Representation</strong></td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Algorithmic Model Interface</td>
<td>45</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1</td>
<td>The oven heating cycle</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Six basic types of resource management components</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Quality-based Adaptive Resource Management Architecture</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>QARMA applied to UAV application</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Five stages of information representation</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Example hardware configuration with system model</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Various network links represented by the system model</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>Attribute categories supported by system model</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>QARMA visual specification editor</td>
<td>41</td>
</tr>
<tr>
<td>4.6</td>
<td>Example two-host hardware configuration with system model</td>
<td>42</td>
</tr>
<tr>
<td>4.7</td>
<td>Example of translation between information representations</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Algorithms and algorithmic models within the architecture</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Example mapping of job series to the resource graph</td>
<td>51</td>
</tr>
<tr>
<td>7.1</td>
<td>QARMA controlling the UAV applications</td>
<td>59</td>
</tr>
<tr>
<td>7.2</td>
<td>Comparison of latencies for UAV 1</td>
<td>60</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 Comparison of resource management architectures ..................... 25
5.1 Algorithmic model interface methods ........................................ 46
6.1 Common control mechanisms .................................................... 55
7.1 UAV latency and frame rate statistics ........................................ 61
7.2 QARMA and DSS integration results .......................................... 63
CHAPTER 1

INTRODUCTION

The past generation of system developers relied heavily on static certification of the performance of real-time systems. Allocation of computing resources, failure response mechanisms, and verification that performance requirements would be satisfied were considered an expensive but necessary part of the design process. Additionally, such allocation design and analysis have two limiting facets: (1) the approach relies heavily on worst-case performance analysis, and (2) not all operating environments can be modeled and analyzed during the system design phase. Static allocation approaches often result in the inefficient use of the available resources and the inability to react to unpredictable changes in the operating environment.

A leading research direction to overcome these problems is called dynamic resource management, which can be used to replace these three system developer tasks with software that can dynamically allocate resources, respond to failures and guarantee performance requirements are satisfied at runtime. Such an approach reduces real-time system development costs, allows more efficient use of computing resources, and permits real-time systems to continue to operate when resources become scarce due to resource failure or other system anomalies.

One direct result of the research in dynamic resource management for computing systems is the development of specialized middleware [1, 2, 3, 4, 5, 6, 7] for use with dynamic
real-time systems. In addition, development paradigms such as aspect-oriented programming [8] are now used to separate the functional requirements from the quality requirements. The goal of such approaches is to reduce the complexity of developing distributed systems with requirements such as timeliness and fault-tolerance while using resources both more efficiently and more effectively.

This thesis focuses on the development of a resource management framework and architecture for real-time systems, with special emphasis on (1) the representation and flow of information, and (2) the decision-making process in such a management system. The Quality-based Adaptive Resource Management Architecture (QARMA)\(^1\) [9, 10, 11] is both a framework for resource management and a collection of CORBA-based\(^2\) middleware services for the resource management of distributed, real-time systems.

The QARMA framework is an architecture paradigm that can be used to: (1) characterize existing resource management architectures and tools, and (2) assist in integrating existing tools into coherent resource management solutions [9, 10, 11]. The components and functionality of many existing architectures [1, 2, 3, 4, 6, 7] and middleware components [5, 12, 13] can be described by the characterizations in the QARMA framework. The framework consists of an architectural view of the components of a resource management system.

---

\(^1\)The QARMA middleware was developed by Matthew Gillen, Andrew Lenharth and David Fleeman for the Center for Intelligent, Distributed and Dependable Systems at Ohio University. David implemented the Resource Management Service. Matthew implemented the Enactor Service and various lower-level enactors. Andrew implemented the System Repository Service.

\(^2\)CORBA is an acronym for the Common Object Request Broker Architecture, a standard for distributed computing that is maintained by the Object Management Group (OMG). All QARMA middleware services use The ACE ORB (TAO), an implementation of CORBA, as their communication mechanism.
The QARMA middleware components are an instantiation of a resource management solution based directly on the framework. The key components are the System Repository Service, the Resource Management Service, and the Enactor Service. The Resource Management Service is the component responsible for making most resource allocation decisions within QARMA. Other components that support these services include monitors for gathering information, detectors for determining when resource allocation changes are needed and enactors for carrying out the desired allocation changes.

The main contributions of this thesis include an analysis of the information life-cycle in a resource management system, a collection of algorithmic models that serve as a basis for resource allocation algorithms in the QARMA middleware, and a description of the greedy algorithm used by the Resource Management Service. Experimental results demonstrate that QARMA can control existing application systems and can be integrated with existing management middleware.

The thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 provides an overview of the QARMA framework and architecture. Chapter 4 describes the QARMA information layer. Chapter 5 describes the algorithmic models representation for making allocation decisions. Chapter 6 discusses the use of algorithms in the Resource Management Service. Chapter 7 provides experimental results using the RMBench simulator [14] and other existing systems and management components [9, 10]. The conclusions are summarized in chapter 8.
CHAPTER 2

RELATED WORK

Resource management in the context of dynamic, distributed real-time systems is the process of monitoring and evaluating the performance of software systems while controlling the amount of hardware resources used by the systems. Monitoring and control must be done in such a manner as to ensure an acceptable quality of service. Dynamic resource management indicates that the resource manager can act autonomously to restore or maintain quality of service as resource reallocations are needed.

Traditionally, real-time system developers coded resource management logic directly into the applications. Current research separates application logic from the resource management logic in order to simplify the development of real-time systems, hence reducing the overall cost of development. One method of separating application logic from resource management logic is referred to as aspect-oriented programming. Another method is the use of middleware services that provide mechanisms for adapting to changes in the operating environment.

The Real-Time CORBA 1.1 [15] specification describes the properties of priority-based end-system schedulers such as Kokyu [13, 16] that controls the local scheduling policy on an end-system. The Real-Time CORBA 2.0 [17] specification adds the concept of a distributable thread in which a task can extend over multiple computation nodes but still have real-time performance requirements. New distributable threads are admitted to the system
by a global scheduling service [12, 13]. The scheduling service can maintain scheduability by cancelling existing tasks or rejecting new tasks if it determines that the requested set of distributable threads is not schedulable¹.

The Realize system, the Dynamic Quality Manager (DQM), the DeSiDeRaTa resource manager, Quality Objects (QuO), the CPU Broker, the ARMS Multi-Level Resource Manager, the Resource Allocation and Control Engine (RACE), and the Amaranth Framework are existing resource management components or architectures. The basic premise in many of these systems is that for environments with resource constraints, it would be beneficial to have applications degrade their quality of output in order to meet their real-time deadlines. In general, each application should provide a result within its respective time constraints. The quality of output for a particular application may be degraded or enhanced to provide a higher overall system benefit. Such mechanisms have proven useful in actual real-time systems such as multimedia applications [18], unmanned aerial vehicles [3], onboard satellite systems [19, 20, 21], and naval surface warfare applications [22].

The Realize [2, 23, 24, 25] system, developed at the University of California, Santa Barbara, is a collection of resource management middleware for use with CORBA applications. Realize is able to dynamically choose the degree of replication and allocate replicas to processors. The residual latencies of each task and the load of each resource is monitored to allow Realize to change the replication degree and allocation as needed. Application sys-

¹QARMA and the Distributed Scheduling Service can work in tandem to provide quality-of-service adjustments as opposed to cancelling or rejecting threads as demonstrated in [10].
tems that have deadline requirements that span over multiple objects working together are not supported by the Realize framework.

In the Dynamic Quality Manager (DQM) [18, 26], *operating levels* are defined in such a manner that choosing a lower operating level reduces the resource requirements for an application. The DQM monitors the performance of applications. When the performance is unsatisfactory, the DQM will interact with the operating system in order to change the application to a lower operating level. The DQM does not need actual resource usage estimates for each operating level since it is intended for use with soft real-time multimedia applications. In other words, the adaptation of the multimedia applications may occur over several periods since a few missed deadlines is not critical.

The service level mechanism in DeSiDeRaTa [27, 28] allocates tasks to processors while choosing initial service levels having the most utility. The service level is not decreased unless the quality constraints are violated and the application cannot be migrated to a new machine. The service level is decreased by one level for each violation until the required quality of service constraint is met. Since actual resource needs at each service level are not considered, the method of service level degradation in DeSiDeRaTa is similar to the DQM approach. When resources are again free, resource monitors cause the resource manager to increase the applications service level. This approach lacks the ability to handle multiple QoS dimensions, assumes monotonically non-decreasing resource usage thus limiting its applicability to one resource, and cannot consider the effects of external environmental conditions such as data-dependent characteristics like cloud-cover [28].
Quality Objects (QuO) [3, 29, 30] is a development paradigm and toolkit for embedding resource management logic within an application using aspect-oriented programming. *System conditions* are objects that contain information about the current state of the system. Information is stored in the system condition objects by various application-specific monitoring components. A *delegate* is linked into an application to control the quality-of-service (QoS) for the application. The delegates invoke *contracts* [31] to decide whether to change the QoS settings for the application based on the current state stored in the system conditions. The delegate then enacts the QoS changes determined by the contract. Initial versions of the Unmanned Aerial Vehicle (UAV) [3] relied on QuO to provide local adaptation absent from any central control. The QARMA middleware successfully controlled this initial version of the UAV in [9]. Recent versions of the UAV demonstrate that applications developed in QuO can interact successfully with centralized management components [32].

The CPU Broker [5] is a CORBA-based middleware component that acts as an agent between real-time applications and real-time operating systems. The CPU Broker controls which applications are using the processor or have DiffServ reservations on the network based on dynamically changing parameters to ensure the most important applications have high quality-of-service. QARMA has been used to coordinate multiple CPU Brokers to allow changes to system priorities across multiple hosts [11].

The ARMS Multi-Level Resource Manager (MLRM) [6] is a collection of middleware implemented as CORBA components. The Resource Status Service (RSS) stores dynamic information concerning the state of the resources in the system. The Application String
Manager monitors application metrics and detects quality-of-service violations for a particular set of applications. The Pool Manager acts as the resource management coordinator for a subset of the resources in the system. The Pool Manager invokes the Resource Allocator which executes resource allocation algorithms. The decisions from these allocators are enacted by the Bandwidth Broker for network priority decisions and the Resource Provider for CPU priority decisions. The Infrastructure Allocator is responsible for deciding the resource pool in which each application will execute. The Deployment and Configuration Engine (DAnCE) [33] is used for deploying the CORBA components.

The Resource Allocation and Control Engine (RACE) [7] is a framework for the active deployment of CORBA components onto computing resources. ResourceMonitors and ApplicationQoSMonitors collect state information. The TargetManager acts as an information repository for resource state information. The DeploymentManager is the container for the decision-making engine for controlling the resource allocation. The algorithms used by the DeploymentManager are programmed in separate objects called Allocators. Changes are enacted in RACE by Controllers.

The Amaranth Framework [4] uses the QoS-based Resource Allocation Model, QRAM, [34, 35] as a policy advisor. In QRAM, each application can have several quality dimensions that are assigned based on the resources available to it. The goal is to optimize overall system utility by choosing both the allocation and the quality dimensions wisely. Each application under QRAM control has a minimum QoS requirement on all relevant quality dimensions and requires a resource profile for each set of quality dimensions.
CHAPTER 3

THE QARMA ARCHITECTURE

Consider the heating cycle of an oven as depicted in Figure 3.1. The oven monitors the current oven temperature, detects the difference in temperature between the desired and actual temperatures, decides if the heat source needs to be activated or deactivated, and then enacts that decision by either activating or deactivating the heat source. The information in the system at any given instant in time includes the desired temperature setting, the actual temperature, and whether or not the heat source is currently activated. Both the actual and desired temperatures are typically displayed to the user on a panel on the top of the oven.

The primary functions of the oven system can be summarized as monitoring, detection, decision-making, enactment of changes, and visualization of the information for the system user. Many systems that we use daily, such as vehicle subsystems, refrigerators, elevators and motion-activated lights, can be characterized by these five functions. Traditionally,

![Figure 3.1: The oven heating cycle](image)

The QARMA Architecture
such systems are referred to as control systems. A control system consists of a feedback loop [36] that is based on a timer or is triggered by a system event.

3.1 Generic Resource Management Components

A resource management system in the context of distributed computing can be defined as a control system. Instead of controlling mechanical devices, such as the heating system in the oven, resource management middleware typically controls other software systems, as well as sensors and actuators that may interface with the software systems. The five functions identified in the oven example can also be identified within many existing resource management systems, along with a sixth function, the storage and distribution of information.

Resource management solutions for distributed systems perform six distinct functions: (1) storing and distributing data, (2) monitoring, (3) detecting, (4) decision-making, (5) enacting decisions and (6) visualization. These six functions are depicted as components within a generic resource management framework in Figure 3.2. The six categories of components are: (1) information repository, (2) monitor, (3) detector, (4) decision-maker, (5) enactor and (6) visualizer. These components represent the basic component types used in the implementation of our middleware services. Each component type is discussed in more detail in [9].
The related work section identifies several existing resource management systems. The DeSiDeRaTa resource manager [27], the Multi-Level Resource Manager (MLRM) [6] and the Resource Allocation and Control Engine (RACE) [7] each have components that can be classified into the six categories in Figure 3.2. Some of the components perform more than one of the six functions. For example, the tasks of monitoring latencies and detecting latency violations in DeSiDeRaTa [27] are carried out by a single management component, the PathManager. The various component classifications for these systems are summarized in Table 3.1.
3.2 Quality-based Adaptive Resource Management Architecture (QARMA)

The Quality-based Adaptive Resource Management Architecture (QARMA) is an implementation of resource management middleware for distributed, real-time systems. The components depicted in Figure 3.3 are derived from the six component types discussed in the previous section. QARMA consists of three core application components: the System Repository Service, the Resource Management Service, and the Enactor Service. Support components include generic monitors, detectors, enactors and visualizors that can be replaced by specialized software depending on the operating context.

Figure 3.3 shows an example configuration of the QARMA components at run-time. One instance of the System Repository Service, the Resource Management Service and the Enactor Service is deployed. An operator provides the specification files, which can be generated through the specification tool, to the System Repository Service. The Resource Management Service and Enactor Service retrieves the specification before listening for system events. These components are the central core of the middleware services.

The remaining components that are depicted in Figure 3.3 are monitors, detectors and enactors. A Host Monitor is executed on each host to log utilization metrics into the System Repository Service. QARMA provides a generic threshold detector program that periodically reads a specific value from the System Repository Service. When a threshold violation is detected, the detector program will invoke the Resource Management Service.
Using the threshold detector program, two Host Detectors are executed on each host for detecting processor overload and underload conditions respectively.

Software Performance Monitors are specialized for the software system they are monitoring. For example, RMBench [14] applications report performance metrics and environmental conditions such as latencies and workload directly into the System Repository Service, bypassing the need to have separate performance monitoring components. However, since the UAV applications [9] write performance metrics into an event channel, a specialized Software Performance Monitor must read the data from the event channel and store the information into the System Repository Service in order for QARMA to manage the UAV systems. Again, the threshold detector program can be used to create detectors for latency violations.

The Resource Management Service retrieves a snapshot of the current state of the system from the System Repository Service once it is invoked by a Host Detector or a Software
Performance Detector. Using this snapshot as the latest information, an algorithm will be executed to determine what control actions would improve the performance of the application systems. The control actions will be sent to the Enactor Service, which is responsible for ensuring that the control actions are successfully enacted.

The Enactor Service does not directly enact the changes, but invokes lower-level enactors that are able to interact with the application systems. For example, QARMA provides a generic lower-level enactor for controlling RMBench applications. To control the UAV system mentioned earlier, a specialized enactor takes the control actions and uses them to change the QuO system conditions that are used by the UAV system.

The monitors, detectors and enactors that are depicted in Figure 3.3 are replaceable depending on operating context or on the development paradigm used for the application systems. One major advantage of QARMA is that it can simultaneously manage multiple application systems developed with different paradigms if the proper specialized monitors and enactors are provided.

3.3 A Real Application Example

Figure 3.4 depicts the architecture resulting from integrating the QARMA middleware with the initial UAV system [3, 9, 37]. The top layer shows the core QARMA components that did not need to be modified to work with the UAV. The second layer depicts the QARMA middleware components that are tailored to a particular environment. For our experiments, we use a Host Monitor for Linux systems and a generic Host Detector. A Software Performance Monitor and Detector collects and analyzes data from an Event Channel provided by the UAV system. A specialized Enactor interprets changes to service attributes into changes to system conditions for the QuO-based UAV system.
The third layer is the QuO architecture components used for the monitoring and adaptation of the UAV system. The fourth layer includes the UAV applications themselves. Video is streamed from a source process that simulates an Unmanned Aerial Vehicle (UAV). The video stream is sent to the distributor process which then sends the video on to the display process. The video can be streamed at 30, 10 or 2 frames per second depending on how many resources are available. In our experiments, we place all UAV source processes on a single host, all the UAV distributor processes on another host, and all the UAV display processes on a final host. The core QARMA components ran on a host separate from the UAV processes.

### 3.4 Architecture Versus Information

In any resource management system, the flow of information is tightly coupled with the design of the architectural component interfaces. However, the information content is not required to be tied to the component interfaces. The *information layer* in QARMA is separated from the architectural component layer as described in the next chapter. The main purpose for this separation is to allow a developer to reuse the QARMA component architecture in various execution contexts by simply replacing the *information layer*. This feature is useful for developing resource managers for different execution contexts. It is also useful for researchers that want to experiment with multiple algorithms within a single execution context.
<table>
<thead>
<tr>
<th>QARMA Framework</th>
<th>QARMA Architecture</th>
<th>DeSiDeRaTa</th>
<th>MLRM</th>
<th>RACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitors</td>
<td>SoftwarePerformanceMonitor HostMonitor HostMonitor NetworkMonitor</td>
<td>PathManager HostMonitor NetworkMonitor</td>
<td>ApplicationStringManager PoolManager</td>
<td>ResourceMonitors ApplicationQoSMonitors</td>
</tr>
<tr>
<td>detectors</td>
<td>HostDetector (threshold)</td>
<td>PathManager HostMonitor PoolManager</td>
<td>ApplicationStringManager</td>
<td>DeploymentManager</td>
</tr>
<tr>
<td>decision-makers</td>
<td>ResourceManagementService AllocationManager</td>
<td>PoolManager ResourceAllocator InfrastructureAllocator</td>
<td>DeploymentManager Allocators</td>
<td></td>
</tr>
<tr>
<td>enactors</td>
<td>EnactorService special purpose enactors (e.g., QuO, RMBench) ApplicationControl special purpose application libraries</td>
<td>ResourceProvisioner BandwidthBroker</td>
<td></td>
<td>Controllers</td>
</tr>
<tr>
<td>information repositories</td>
<td>SystemRepositoryService</td>
<td>none</td>
<td>ResourceStatusService</td>
<td>TargetManager Historian</td>
</tr>
<tr>
<td>visualizers</td>
<td>SystemRepositoryDisplay HCI</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of resource management architectures
CHAPTER 4

THE QARMA INFORMATION LAYER

The QARMA information layer is shown in Figure 4.1. At the highest level, a system model defines the abstract resources and software elements that may be managed by the resource manager. The system model is not actually part of the implemented resource management solution, but provides a language for describing (1) the characteristics of each resource and software element and (2) the resource management algorithms to be implemented.

While the system model provides an abstract representation of managed resources and software elements, more concrete representations must be implemented within the resource management middleware. Figure 4.1 depicts the relationships between five common stages of information representations in resource management middleware: (1) specification, (2) run-time storage, (3) algorithmic models, (4) run-time visualization and (5) log files. The transformation of information between these various representations are part of the information life-cycle. All five of these representations are subject to the abstract limitations and assumptions imposed by the system model.

The system model must classify information as either static or dynamic. Static information does not change while the resource manager and application systems are in operation. Information is dynamic if the value can change while the system is in operation. A com-
mon scenario is that the number of processors in a hardware system is static while the CPU utilization of each processor is dynamic.

In a generic resource management approach, it is difficult to draw a line that defines which information is static and which is dynamic. For example, in manufacturing systems, the number of computing resources available may be static as new resources are not added or removed while the system is in operation. However, in complex military environments such as those involving UAV systems, the number of computing resources may be dynamic as new computing resources enter and exit the system. For this reason, our system model assumes all information can be dynamic. However, restrictions are imposed within the current middleware implementation.
The next section describes the system model currently used by the QARMA middleware. Examples demonstrate the transformation of information during the information life-cycle. The next chapter describes the algorithmic models representation in detail.

4.1 The System Model

The system model can be divided into three parts: (1) resource configuration, (2) software configuration, and (3) dynamic state information.

4.1.1 Resource Configuration

A resource is an allocatable unit of time or space. While this definition of a resource is conceptually simple, the system model must provide a representation of the resources that can be tied to real hardware configurations. The QARMA system model currently describes basic properties of hardware systems consisting of single-processor machines, network routers, network switches and network links between the network interfaces of these devices. Figure 4.2 illustrates three single-processor machines connected by full-duplex links through a network switch with a subset of the system model describing the configuration.

Each computer, router and switch is called a device. The set of devices in a particular hardware configuration is represented by the symbol $D$, and the number of devices are denoted as $|D|$. A specific device is referenced as device$_i$ where $1 \leq i \leq |D|$. Each device$_i$
Figure 4.2: Example hardware configuration with system model

has a type written as $\text{type}(device_i)$. If the device is a general purpose computer, then the type is host. If the host is a dedicated router or switch, the type is router or switch. Each $device_i$ has a single processor resource denoted as $proc_i$ and a memory resource written as $mem_i$. The size of the memory on $device_i$ is written as $\text{size}(mem_i)$, an integer in units of Mega-bytes (MB). Each $device_i$ also has a set of network interfaces. The $k^{th}$ network interface of $device_i$ is written as $\text{int}_{i,k}$. Finally, each host has a loopback interface, $\text{loopback}_i$ that is used to represent the communications channel between two applications executing on the same device.

The previous description of the $device$ element is sufficient for describing stand-alone computers. However, to support a network of computers, the connections between each $device$ must be represented. The QARMA system model defines a network link as a con-
A connection between two or more network interfaces. Each network link is denoted as $\text{link}_i$ and has a bandwidth resource written as $\text{bandwidth}_i$. The amount of bandwidth available on $\text{link}_i$ is written as $\text{size}(\text{bandwidth}_i)$, an integer representing the maximum Mega-bits per second (Mb/sec) transfer capacity of the network link.

Figure 4.3: Various network links represented by the system model

$D = \{\text{device}_1, \text{device}_2\}$
$I_1 = \{\text{int}_1\}$
$I_2 = \{\text{int}_2, \text{int}_3, \text{int}_4\}$
$L = \{\text{link}_1, \text{link}_2\}$
$\text{link}_1 = \{\{\text{int}_1\}, \{\text{int}_2\}\}$
$\text{link}_2 = \{\{\text{int}_2\}, \{\text{int}_1\}\}$
$\text{size}(\text{bandwidth}_1) = \text{size}(\text{bandwidth}_2) = 100$

$D = \{\text{device}_1, \text{device}_2\}$
$I_1 = \{\text{int}_1\}$
$I_2 = \{\text{int}_2, \text{int}_3, \text{int}_4\}$
$L = \{\text{link}_1\}$
$\text{link}_1 = \{\{\text{int}_1, \text{int}_2\}\}$
$\text{size}(\text{bandwidth}_1) = 100$

$D = \{\text{device}_1, \text{device}_2, \text{device}_3\}$
$I_1 = \{\text{int}_1\}$
$I_2 = \{\text{int}_2\}$
$I_3 = \{\text{int}_3, \text{int}_4, \text{int}_5\}$
$L = \{\text{link}_1\}$
$\text{link}_1 = \{\{\text{int}_1, \text{int}_2, \text{int}_3\}\}$
$\text{size}(\text{bandwidth}_1) = 100$

$D = \{\text{device}_1, \text{device}_2, \text{device}_3, \text{device}_4\}$
$I_1 = \{\text{int}_1\}$
$I_2 = \{\text{int}_2\}$
$I_3 = \{\text{int}_3\}$
$I_4 = \{\text{int}_4, \text{int}_5\}$
$L = \{\text{link}_1\}$
$\text{link}_1 = \{\{\text{int}_1, \text{int}_2, \text{int}_3, \text{int}_4\}\}$
$\{\text{int}_1, \text{int}_2, \text{int}_3, \text{int}_4\}\}$
$\text{size}(\text{bandwidth}_1) = 56$
A network link must have two non-empty sets of participating interfaces. The source interfaces are the interfaces that can send data over the network link. The sink interfaces are the interfaces that can receive data over the link. These two sets of interfaces are sufficient to model the following network configurations: (1) full-duplex link between two hosts, (2) half-duplex link between two hosts, (3) multiple hosts connected through a network hub, and (4) multiple remote hosts connected to a wireless access point. Figure 4.3 demonstrates how all four of these network links would be represented using the system model notation.

Additional notation for network links is needed to identify the link between two specific interfaces. The notation \( \text{link} \left( int_{i,k}, int_{j,l} \right) \) refers to the network link that connects the source interface \( int_{i,k} \) to the sink interface \( int_{j,l} \).

The loopback interface on each device is treated as a network link that connects a computer to itself. It is assumed that the bandwidth on each loopback interface is unlimited, and that the speed is fast enough that it can be ignored. Future versions of the system model should take into account delay introduced by communicating over the loopback interface, as well as other communication mechanisms such as shared memory.

### 4.1.2 Software Configuration

**Systems, Applications and Paths**

The software system model describes the properties of the application systems to be managed. At the top level, there is a collection of software systems denoted as \( S \). The number of software systems is represented as \(|S|\). The \( i^{th} \) software system, \( \text{system}_i \), contains a
set of applications represented as $A_i$. The number of applications in $A_i$ is $|A_i|$ and the $k^{th}$ application in $A_i$ is referenced as $app_{i,k}$. Each application represents an executable with a primary thread of execution. Executables with multiple primary threads could be modeled as separate applications, however, this technique restricts QARMA from dynamically relocating applications from one processor to another. This issue has been addressed in subsequent versions of the system model by adding tasks to applications, but has not yet been integrated into the middleware.

To describe a collection of connected applications, each software system has a set of path elements, $P_i$, where the number of path elements is $|P_i|$. Each $path_{i,k}$ has a connectivity graph of applications that contains one application denoted as the source application, $source_{i,k}$, and the remaining connectivity denoted as pairs of applications. To denote that $app_{i,s}$ communicates to $app_{i,t}$, we write $comm_{i,k}(app_{i,s}, app_{i,t})$. A path can have zero or more real-time requirements including the $period(path_{i,k})$ and the $deadline(path_{i,k})$.

**Service and Environment Attributes**

Perhaps the most important part of the software system model is the representation of attributes. Much documented work exists for the topics of quality-of-service (QoS) parameters and workload parameters. However, most work focuses on just one of these parameters, ignoring both the modeling and effects of the other influencing parameters. The software system model provides a framework for describing systems that have many competing attributes with varying properties.
The two classes of attributes are service attributes and environment attributes. An attribute is a *service attribute* if the resource management system has the ability to control the value of the attribute. An attribute is an *environment attribute* if the resource management system is not permitted to control the value of the attribute. In other words, the value of an environment attribute is determined by the environment in which the software system is executing. In earlier papers [9, 10], the term extrinsic is used in place of the term environment.

The classification of an attribute is further refined based on two factors: (1) whether or not the attribute affects the resource usage of a software system, and (2) whether or not the attribute affects the perceived benefit of executing the software system. The number of possible combinations of these two factors is four. Thus, an environment attribute can be further classified as one of four attribute categories. However, a service attribute can be further classified as one of only two attribute categories since some combinations of the factors would not occur in a properly designed system. The six categories are depicted in Figure 4.4.
The two categories of service attributes are *quality-of-service* and *service alternative*. The category of a service attribute is *quality-of-service* if the attribute directly affects the perceived benefit of executing the application. The category is *service alternative* if the attribute does not affect the perceived benefit.

For both of these service attribute categories, the resource usage will be affected. If changes in the value of a quality-of-service attribute does not affect resource usage, then the resource management system would always choose the value that has the largest perceived benefit. Thus, in reality, there is really only one meaningful value for the attribute making it useless to model it as such. Along the same line of reasoning, if changes in the value of a service alternative does not affect the resource usage of a software system, then the changing of the attribute value has no perceptible effect on the system making it useless to model it as a service attribute.

The four categories of environment attributes are *load*, *characteristic*, *simple* and *complex*. Most research has focused on environment attributes that are categorized as load. Historically, the term *workload* has been used to represent the amount of work that an application must complete during a given invocation [38]. The category of an environment attribute is *load* if changes in the value of the attribute affects the resource usage of the software system, but does not affect the perceived benefit of executing the applications in the system. This definition is consistent with much of the literature concerning the term *workload*. 
The category of an environment attribute is *characteristic* if changes in the value of the attribute affects the perceived benefit of system execution, but does not affect the resource usage of the software system. The color of a captured image is a common characteristic used in real-time vision systems. For example, in a cloud cover detection algorithm on a low-earth orbiting satellite for hot-spot detection [28], a high percentage of white pixels in the picture may indicate a high cloud-cover meaning that the picture holds little perceived scientific benefit.

The final two categories of environment attributes are less common, and have been termed simple and complex. A *simple* environment attribute does not affect resource usage nor perceived benefit. It is useful to include this category to make use of the monitoring and display tools available in the resource management software. The software may not be able to use the simple attribute in making decisions, but the changing values may still be important to the human operating the visualization tools and analyzing the log files. A *complex* environment attribute affects both the resource usage and perceived benefit of the software system. Our research has not yet identified an environment attribute that must be classified as complex, but since the category can be considered a superset of all the other types, it is useful to include in the system model for use in algorithms as described in the next chapter.

The set of attributes in \( \text{system}_i \) is denoted as \( \alpha_i \). The notation \( \text{attr}_{i,k} \) refers to the \( k^{th} \) attribute in \( \alpha_i \). The class of each attribute is written as \( \text{class}(\text{attr}_{i,k}) \) and evaluates to either *service* or *environment*. The category of each attribute is written as \( \text{category}(\text{attr}_{i,k}) \). The
value assumed by an attribute can be a string, an integer, or a real number. This is called the type of the attribute and is written as $type(\text{attr}_i,k)$. The type method can assume the value string, integer, or real.

All possible values for an attribute must be enumerated in the system model. For the numeric types, having a statistical model such as min and max values would be desirable, but this is left for future work. The $n^{th}$ possible value for the $k^{th}$ attribute of $\text{system}_i$ is written as $value_n(\text{attr}_{i,k})$. The current value of the attribute while the system is in operation is denoted as $value(\text{attr}_{i,k})$.

**Resource Usage Profiles**

A resource usage profile describes how much of each resource is needed for some task or set of tasks to finish. Resource usage profiles are recorded for each application using the methods described in previous work with the DeSiDeRaTa resource manager [39]. The original resource usage profile model allows two parameters to affect the resource usage of an application: the host on which the resource usage measurements were collected, and an optional workload value. The workload value is replaced in the resource usage profiles in QARMA with a set of values for all service attributes, and a set of values for all environment attributes.

The notation for the profile model of $\text{app}_{i,k}$ is $\text{profile}_{i,k}$. A specific resource usage profile is denoted as $\text{profile}_{i,k,l}$. Properties of a resource usage profile include the execution time, $\text{executime}(\text{profile}_{i,k,l})$, measured in milliseconds (ms), the memory usage,
mem(\text{profile}_{i,k,l})$, measured in kilobytes (KB), the received bytes, \( rcbytes(\text{profile}_{i,k,l}) \), measured in bytes, and the transmitted bytes, \( trbytes(\text{profile}_{i,k,l}) \), also measured in bytes.

Each of the resource usage properties can be representative of any statistical measurement. For soft real-time systems, the system designer may use average measurements to reduce the amount of wasted processor time. For hard real-time systems, the designer may use worst-case measurements so that schedulability analysis [40] algorithms can ensure that timing violations will not occur. Future work will develop resource usage profiles with complete statistical models of the resource usage of applications as opposed to the single value approach now implemented.

One limitation in QARMA is that possible values of the attributes must be enumerated. Likewise, all possible resource usage profiles must also be enumerated. However, since attributes are modeled for systems, each attribute is not required to affect the resource usage of every application. If the resource usage of an application is not affected by any attribute in the system, then the application will have only one resource usage profile. If an application is affected by a single attribute with five possible values, then the application will have five distinct resource usage profiles representative of each value of the attribute. If an application is affected by two attributes each with five possible values, the number of resource usage profiles needed is twenty-five. Thus, as the number of influencing attributes increase, the number of profiles needed grows exponentially. For the applications we have dealt with, this growth is not a problem as most applications usually have one service
attribute, one environment attribute, or one of each type of attribute that affects the resource usage of the application.

**Benefit For User Preferences**

A *benefit model* formally describes user preferences for use by the resource management middleware in making decisions that will optimize system performance. Like the resource usage profile model described in the previous section, the current values for both service and environment attributes can affect the benefit that the user experiences from having the software system execute. The *benefit* of executing a system given specific attribute settings is represented as a real number in the range \([0, 1]\), where a value of 0 represents no benefit to the user and a value of 1 indicates that maximum benefit is achieved. A benefit model exists for each software system, so the benefit model for \(system_i\) is denoted as \(benefit_i\).

As described in the section about attributes, not all service and environment attributes are required to affect the benefit of the software system. For example, a video transmission system may contain a *service alternative* attribute that allows the system to either compress a video stream before transmission or to transmit the video stream uncompressed. Both alternatives may be perfectly acceptable to the system user, but one alternative will use more processor resource and less bandwidth, where the other alternative will use less processor resource and more bandwidth. The resource manager is free to decide which alternative to use in order to make best use of the available resources.
The benefit model in our system model requires a complete description of the attributes that affect the benefit. This leads to the same exponential modeling problem as described for the resource usage profile model.

### 4.1.3 Dynamic State Information

A *snapshot* [31] refers to the system state at a particular instance in time. For example, the current utilization of each processor may be included in a *snapshot* of the resource state. A snapshot in the system model contains the current utilization of all the resources, the current value of all service and environment attributes, and the current mapping of applications to hosts.

The utilization of the processor resource on device$_i$ is written as $\text{util}(\text{proc}_i)$. The amount of memory used on device$_i$ is denoted as $\text{used}(\text{mem}_i)$ and the utilization is denoted as $\text{util}(\text{mem}_i)$. The amount of bandwidth used on link$_i$ is written as $\text{used}(\text{bandwidth}_i)$ and the utilization is written as $\text{util}(\text{bandwidth}_i)$. The current value of attr$_{i,k}$ while the system is in operation is denoted as $\text{value}(\text{attr}_{i,k})$. The notation $\text{host}(\text{app}_{i,k})$ refers to the host on which app$_{i,k}$ currently resides.

The system model elements can be used to formulate new system model elements as defined functions. For example, the available memory on device$_i$ can be computed as $\text{avail}(\text{mem}_i) = \text{size}(\text{mem}_i) - \text{used}(\text{mem}_i)$. 
4.2 Information Representations

The five representations depicted in Figure 4.1 are: (1) specification, (2) run-time storage, (3) algorithmic models, (4) run-time visualization, and (5) log files. The transformation of information between these various representations are part of the information life-cycle. All five of these representations are subject to the limitations and assumptions imposed by the system model.

The system model assumes that any particular system model component or property is dynamic. However, the current version of our middleware requires that the resource configuration and software configuration defined in the previous section are modeled as static information. Future versions of the resource management middleware will allow dynamic changes to the configuration.

The specification representation consists of a language for describing the static properties of the resources and software configuration. The language is intended to be both human and computer readable. A visual specification editor\(^1\) [41] is provided with QARMA to aid system developers. The editor, depicted in Figure 4.5, is based on the Generic Modeling Environment (GME) [42, 43] and the VEST project [44, 45].

The properties of a system are defined in the specification language and stored in specification files. The specification parser converts the specification files into the run-time storage representation and stores the information in the System Repository Service.

---

\(^1\)Eric Aber is the primary developer of the QARMA visual specification editor.
The Resource Management Service pulls both the static and dynamic information from the System Repository Service. This information is converted from the run-time storage representation into the algorithmic models representation in order for the Resource Management Service to make allocation decisions. The dynamic run-time storage information in the System Repository Service are eventually updated with new values based on the decisions of the Resource Management Service.

To help tie these information representations together, consider a resource configuration that consists of two hosts connected through a full-duplex network switch. A subset of the system model for such a configuration is provided in Figure 4.6.

The translation between three of the information representations is depicted in Figure 4.7. The resource configuration information is described in the specification language and stored in a file. The specification language representation is converted by a parser into the run-time storage representation and stored as data in the System Repository Service along with the dynamic state information for each resource. The Resource Management Service
$|D| = 3$
$D = \{\text{device}_1, \text{device}_2, \text{device}_3\}$
type(\text{device}_1) = \text{type(}\text{device}_2) = \text{host}$
type(\text{device}_3) = \text{switch}$
size(\text{mem}_1) = size(\text{mem}_2) = 512
$I_1 = |I_2| = 1$
$I_4 = 4$
$I_1 = \{\text{int}_{1,1}\}$
$I_2 = \{\text{int}_{2,1}\}$
$I_3 = \{\text{int}_{3,1}, \text{int}_{3,2}, \text{int}_{3,3}, \text{int}_{3,4}\}$
$L = \{\text{link}_1, \ldots, \text{link}_4\}$
link$_1 = \{\{\text{int}_{1,1}\}; \{\text{int}_{3,1}\}\}$
link$_2 = \{\{\text{int}_{3,1}\}; \{\text{int}_{1,1}\}\}$
link$_3 = \{\{\text{int}_{2,1}\}; \{\text{int}_{3,2}\}\}$
link$_4 = \{\{\text{int}_{1,2}\}; \{\text{int}_{2,1}\}\}$
size(\text{bandwidth}_1) = size(\text{bandwidth}_2) = 100
size(\text{bandwidth}_3) = size(\text{bandwidth}_4) = 100

Figure 4.6: Example two-host hardware configuration with system model

uses the run-time storage representation to generate the algorithmic model of the resource configuration.

The final representations are less important to the dynamic operation of the middleware, but are required for human interaction and off-line analysis. Host monitors collect utilization metrics on each host and update the run-time storage values in the System Repository. Logging tools pull the utilization metrics from the System Repository Service and write the information to files for off-line analysis. Visualization tools pull the resource configuration and the utilization metrics from the System Repository and present the human operator with a visual representation of the information.

All five of these information representations provide interesting research opportunities and have significant design implications on the resource management architecture. However, the specification, run-time storage, run-time visualization and log file representations
Figure 4.7: Example of translation between information representations

will not be described in detail as part of this thesis as the work was primarily implemented by other students at Ohio University. The next chapter describes in detail the algorithmic model representation used by the Resource Management Service.
CHAPTER 5

ALGORITHMIC MODELS

REPRESENTATION

The Resource Management Service uses a special information representation referred to as the algorithmic models representation. This thesis develops a set of algorithmic models that encapsulates the methods that an algorithm designer would use if trying to write algorithms based on the system model presented in the previous chapter. The core features for this model are found in Jane Liu’s Real-Time Systems book [46]. The research presented in this chapter does not claim to produce new scheduling theory results, but provides an architecture in which algorithms can be developed and tested with real resource management middleware and with real application systems.

Figure 5.1 depicts the relationship between resource management algorithms and the algorithmic models developed in QARMA. The algorithmic model interface is a set of methods inherited by all algorithms implemented in the Resource Management Service. Behind the interface, each algorithm also inherits four algorithmic models: (1) the resource model, (2) the task model, (3) the profile model, and (4) the benefit model. Each model encapsulates information needed to make resource management decisions, and provides further interfaces for resource allocation algorithms to invoke.
Figure 5.1: Algorithms and algorithmic models within the architecture

The interface is implemented as an abstract class that is inherited by each algorithm in the Resource Management Service. The next section discusses the main functions available in the abstract class. The rest of this chapter discusses the four algorithmic models.

5.1 Algorithmic Model Interface

Table 5.1 lists the methods available through the top-level algorithmic model interface. These methods are used to create the algorithmic models and to update the current state of the models in a dynamic environment.

The initialize_models method takes as input the run-time storage representations of the complete resource and software configurations, as well as the most recent snapshot of the following dynamic state information: (1) allocation, (2) attributes, (3) system importances, and (4) resource usage. The method uses these inputs to generate instances of the four abstract algorithmic models. The result is that every algorithm has the follow-
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize_models</td>
<td>rconf, sconf, allo, attrs, sysimps, rusage</td>
<td>Creates the resource model, benefit model, profile model and task model based on the resource and software configuration. Then updates the state of the models with the current allocation, attribute settings, system importances and resource usage.</td>
</tr>
<tr>
<td>set_allocation</td>
<td>allo</td>
<td>Updates the resource and task models based on the given allocation.</td>
</tr>
<tr>
<td>set_attributes</td>
<td>attrs</td>
<td>Updates the task model based on the given attribute settings.</td>
</tr>
<tr>
<td>set_service_attributes</td>
<td>sattr</td>
<td>Updates the task model based on the given service attribute settings.</td>
</tr>
<tr>
<td>set_environment_attributes</td>
<td>eattr</td>
<td>Updates the task model based on the given environment attribute settings.</td>
</tr>
<tr>
<td>set_resource_usage</td>
<td>rusage</td>
<td>Updates the resource model based on the given resource usage.</td>
</tr>
<tr>
<td>set_system_importances</td>
<td>sysimps</td>
<td>Updates the importance of each of the systems for the algorithm. This currently does not affect any of the algorithmic models, but is used by our greedy algorithm.</td>
</tr>
</tbody>
</table>

Table 5.1: Algorithmic model interface methods

...ing private member variables which provide further interface methods: resource\_model, profile\_model, benefit\_model, and task\_model. The algorithmic models representation in QARMA consists of these four models. The methods that start with set are used to update the four models with the current state of the system.
5.2 Resource Model

The resource model transforms the resource configuration described in the system model into a directed graph, where each vertex represents a processor\(^1\), and each edge represents the ability to send data from the origin vertex to the terminal vertex. A vertex \(v_t\) is directly accessible from vertex \(v_o\) if an edge exists such that \(v_o\) is the origin vertex and \(v_t\) is the terminal vertex. Algorithm 1 is used to construct the directed graph which we will call the resource graph in the remainder of this thesis.

The resource model does not assume static routes at this time. The route between two hosts is defined on the resource graph as an ordered list of vertices such that the following three properties hold: (1) the first vertex represents the host that is transmitting data, (2) for each remaining vertex, \(v_i\), in the list, \(v_i\) is directly accessible from \(v_{i-1}\), and (3) the final vertex represents the host that is receiving data. The resource model provides the method \(\text{find\_route}(v_i, v_j)\) which will return the route between two processors based on the shortest path algorithm.

5.3 Profile Model

The profile model encapsulates the resource usage information for each application. The \text{get\_profile()} interface returns the resource usage profile for a given application when

\(^1\text{Excerpt from page 35 of reference [46]: “They [processors] carry out machine instructions, move data from one place to another, retrieve files, process queries, and so on. Every job must have one or more processors in order to execute and make progress toward completion.”}
Algorithm 1 Generate the resource graph

\[ n = 0 \]
\[ \text{for (each } host, \ router, \text{ and } switch \text{) do} \]
\[ \quad \text{Add vertex } v_n \text{ to graph for host, router or switch} \]
\[ \quad \text{Add vertex } v_{n+1} \text{ to graph for loopback interface} \]
\[ \quad \text{Add directed edge from } v_n \text{ to } v_{n+1} \]
\[ \quad \text{Add directed edge from } v_{n+1} \text{ to } v_n \]
\[ \quad n = n + 2 \]
\[ \text{end for} \]
\[ \text{for (each } link_i \text{) do} \]
\[ \quad \text{Add vertex } v_n \text{ to represent link} \]
\[ \quad \text{for (each source interface) do} \]
\[ \quad \quad \text{Add directed edge from the source interface host vertex to } v_n \]
\[ \quad \text{end for} \]
\[ \quad \text{for (each sink interface) do} \]
\[ \quad \quad \text{Add directed edge from } v_n \text{ to the sink interface host vertex} \]
\[ \quad \text{end for} \]
\[ \quad n = n + 1 \]
\[ \text{end for} \]

operating with given attributes settings. The profile includes the execution time, the amount of data received and generated by the application and the amount of memory required by the application. This information is extracted from the specification files that were parsed into run-time data structures and stored in the System Repository Service.

The task model described later in this chapter will further discuss how to use the profile model. Specifically, the profile model is used heavily by schedulability analysis algorithms which verify whether or not there are sufficient resources to satisfy all system requirements for a particular system configuration.
5.4 Benefit Model

The benefit model encapsulates the benefit information (e.g., user preferences for attribute settings) for each software system. Given a software system identifier and the settings of all attributes in the system, the `get_benefit()` interface returns a numeric value in the range \([0, 1]\) which represents how much the user values the given attribute settings.

The benefit model also provides the interfaces `get_min_benefit()` and `get_max_benefit()` which return the associated service attribute settings that render the least benefit and the most benefit respectively. These methods are useful when the developer can assume that the resource usage is monotonically non-decreasing with respect to the benefit. The next chapter includes details on how resource management developers can incorporate the benefit model interfaces into their allocation algorithms.

The benefit model currently used is very limited. However, by isolating the concept of benefit into a separate model, future work can easily enhance the benefit model concept by including aggregate utility measures in resource allocation algorithms [47]. Other existing benefit or utility models can also be incorporated into the framework [34, 48, 49].

5.5 Task Model

The task model transforms the software configuration described in the system model to a task/job model similar to the one presented by Jane Liu [46]. We consider a job as the smallest unit of work in the system. QARMA currently supports three types of jobs: (1)
Algorithm 2 Generate the job series for a single path

Require: The path must have at least one application
Require: jobseries is empty
        Add job_0 to jobseries for the first application in path
        n = 1
        for (each remaining app_i in path) do
            route_i = find_route(host(app_{i-1}), host(app_i))
            Discard v_0 {since already added to job series}
            for (each remaining v_j in route_i) do
                Add job_n to jobseries
                n = n + 1
            end for
        end for

computation, (2) data transmission, and (3) routing. Each processor defined in the resource graph is capable of completing one or more of these job types.

The primary purpose of the task model is to expand each path into a complete series of jobs that are mapped onto the vertices of the resource graph. We will use the phrase job series to refer to the generated series of jobs. In order to do the conversion, the task model is given an allocation, or mapping of each application in the path to a host. The function host(app_i) will be used in Algorithm 2 to indicate the processor to which the app_i is assigned. Algorithm 2 generates the job series for a given path.

Figure 5.2 demonstrates how a path with two applications can be expanded into its corresponding job series. The resource configuration consists of two hosts connected through a network switch. The first application is allocated to host_1 and the second application is allocated to host_2. host_1 communicates to host_2 through the network switch. The job series for this path consists of five jobs: (1) computation on v_0, (2) transmission on v_6, (3) routing on v_4, (4) transmission on v_8, and (5) computation on v_2.
Once all paths are mapped into their respective job series, then each vertex in the resource graph will have an associated list of jobs that will execute on the resource. The desired goal is to be able to perform schedulability analysis independently on each processor such that we can guarantee that the path scheduling parameters (e.g., deadlines) are satisfied. Recent work has provided mechanisms for performing such schedulability analysis algorithms [40, 46].

The algorithmic model currently assumes each path has a period and a deadline such that the deadline is less than or equal to the period. Aperiodic paths can be modeled as periodic by dividing 1 by the minimum arrival rate. While this leads to pessimistic assumptions about the utilization of the system, it provides a simple framework for providing guarantees on the performance of the entire system.
Algorithm 3 Schedulability analysis for single resource

**Require:** \( 0 < \text{threshold} \leq 100 \)

\[
\text{if } (\text{util}(\text{processor}_i) > \text{threshold}) \text{ then}
\]
\[
\quad \text{return } false
\]
\[
\text{end if}
\]
\[
\text{return } true
\]

Algorithm 4 provides a high-level description of how QARMA performs schedulability analysis for the entire system. QARMA assumes that each job in the job series has a period equal to the period of the path to which the job belongs. This assumption can be enforced by the scheduler by using the release guard protocol\(^2\) [40, 46]. Deadlines are assigned to each job in a path based on the proportional deadline algorithm\(^3\) [46]. Now that each job has an assigned period and deadline, classical uni-processor schedulability analysis algorithms can be applied to each vertex in the resource graph independently. If every vertex passes the schedulability analysis test, then it can be assumed that the entire distributed system is schedulable.

The assignment of periods and deadlines to jobs can be changed to reflect the synchronization protocols used in the underlying system. Also, the uni-processor schedulability analysis algorithm can be changed to reflect the schedulers used on the underlying system. Our experiments perform schedulability analysis based on rate-monotonic scheduling.

The utilization bound [50] is computed and if the utilization of the processor is less than the bound, then the task set on the processor is schedulable. We have also implemented Lehoczky’s [51] exact algorithm for rate-monotonic scheduling.

---

\(^2\)Pages 361-368 of [46] describe several task synchronization protocols.

\(^3\)Pages 369-371 of [46] describe four common deadline-assignment algorithms.
**Algorithm 4** Perform schedulability analysis for the entire system

**Require:** Resource graph and allocation are given

for (each path $p_i$ in the system) do

  Generate the job series for $p_i$
  Assign period to each job (period of $p_i$)
  Assign deadline to each job (proportional deadline)

end for

for (each vertex $v_i$ in the resource graph) do

  Perform algorithm 3 for uni-processor schedulability analysis
  if (schedulability analysis fails) then
    return false
  end if

end for

return true
CHAPTER 6

RESOURCE ALLOCATION

ALGORITHMS

Resource allocation algorithms in QARMA are algorithms that take as input the specification and system state and produce as output changes to the system state such that performance requirements are satisfied and performance is optimized based on defined optimization criteria. The definition of any resource allocation algorithm is greatly dependent on four factors: (1) what information is available, (2) what performance requirements must be satisfied, (3) what performance measure must be optimized, and (4) what control mechanisms are available for changing the state of the system.

The information discussed in the previous two chapters is available for use by the allocation algorithms in QARMA. One requirement for all allocation algorithms is that they produce system state changes such that all real-time tasks are schedulable, meaning each task is able to complete before its respective deadline. The task model has a schedulability_check() method that is used to determine if the state of a particular system is schedulable. Schedulability analysis is performed as described in the previous chapter, e.g., classic single-processor analysis is applied to each set of tasks that are mapped onto each resource.
Control Mechanisms | Description
---|---
Service Level Adaptation | The term *service level* is used to specify the application-specific, non-empty set of variables that can be used to control the performance characteristics of an application. In QARMA, the *service level* is affected by setting new values for *service attributes*.
Process Initial Allocation | The ability to dynamically choose a processor on which to execute a new application that has been introduced into the system.
Process Migration | The ability to move an application that is already operational in the system to a new processor. This process includes both stateless and stateful migration of processes.
Process Replication | The ability to support multiple instances of a single application such that the instances reside on separate processors. Replication can be used for *hot spares* or for *load sharing*.
Dynamic Network Routing | The ability to re-route the network channels used between two communicating applications in the case of link overload or link failure.
Network Reservation and Prioritization | The ability to dynamically control the IntServ priorities and DiffServ network reservations for the system. Cite Bandwidth Broker and CPU Broker.
CPU Reservation and Prioritization | The ability to dynamically control the amount of CPU reserved for an application or the operating system priorities for each application. Cite CPU Broker and Scheduling Service and Kokyu.

Table 6.1: Common control mechanisms

Table 6.1 lists several common control mechanisms used by various resource management solutions. However, the only control mechanism available in QARMA at the time of this thesis is the ability to change service attribute settings. Therefore the goal of the algorithms in this thesis is to choose a set of service attributes such that the system is schedulable and the overall benefit is maximized. The *benefit model* described in the previous chapter reduces a particular system state to a single real value that represents the combined overall benefit.
Algorithm 5 Greedy algorithm for determining service attribute settings

Require: Algorithmic models initialized to current state of system
1: benefit_model → get_min_benefit(service_attrs)
2: task_model → set_service_attrs(service_attrs)
3: if (task_model → schedulability_check()) then
4:   for (each system \(i\) from most important to least important) do
5:     schedulable = false
6:     benefit_model → analyzers[system\(i\)] → sort_rows_to_benefit()
7:     rowcount = benefit_model → analyzers[system\(i\)] → rowcount
8:     row = rowcount − 1
9:     while ((not schedulable) and (row ≥ 0)) do
10:    t_attrs = benefit_model → analyzers[system\(i\)][row] → sa_settings
11:    task_model → set_service_attrs(system\(i\), t_attrs)
12:    schedulable = task_model → schedulability_check()
13:    if (not schedulable) then
14:      row = row − 1
15:    else
16:      service_attrs[system\(i\)] = t_attrs
17:    end if
18: end while
19: end for
20: end if
21: generate_change_list(service_attrs, changes)
22: return changes

Algorithm 5 is a greedy heuristic for assigning service attribute settings for a system operating under the control of the QARMA resource manager. This algorithm assumes that less resources are required by service attribute settings when benefit values are lower. Lines 1 and 2 set the service attributes to be the settings that accrue the least benefit to the user. If this setting of attributes is not schedulable in line 3, then the algorithm skips to line 21. Otherwise, the algorithm tries to optimize the service attribute settings in lines 4 through 19.
Line 4 loops through each software system in order of their importance as defined by a QARMA-defined environmental attribute. Since our benefit models are discrete, every combination of service attribute settings can be identified by a row number. Line 6 orders the benefit rows for a particular software system from least benefit to greatest benefit. Lines 9 through 18 loops through these benefit rows from greatest to least until a schedulable setting of service attributes is found. Therefore, the most important software system will have the first opportunity to increase its resource consumption. This process continues from the most important software system to the least important software system.

Line 21 now uses the optimized service attribute settings to generate a list of changes to the current state of the system. These changes are returned in line 22. The Resource Management Service would then invoke the Enactor Service to carry out the changes determined by the algorithm.

The experiments in the next chapter use Algorithm 5 as the decision-making process in the Resource Management Service. Previous work with the maximizing allowable workload problem [52, 53] indicates that such a greedy algorithm is a good solution for resource allocation algorithms at run-time.
CHAPTER 7

EXPERIMENTAL RESULTS

Two separate experiments\(^1\) demonstrate the effectiveness of the QARMA decision-making process. The first experiment \([9]\) places the BBN UAV platform \([3, 37]\) under the control of QARMA. At the time of the experiment, the UAV applications had been designed to use QuO with local adaptation strategies absent from any central control. The experiment demonstrates that QARMA can use the notion of system importance to ensure that the more important systems have access to the computing resources before less important systems.

The second experiment \([10]\) integrates QARMA with the Dynamic Scheduling Service (DSS) \([13]\) provided by the University of Rhode Island. QARMA alone cannot enforce that the operating system executes applications based on their priorities. The DSS ensures that the applications are scheduled on the processor resources in a predictable manner. Experiments were conducted using simulated UAV systems designed in our RMBench tool \([14]\).
7.1 QARMA-Controlled UAV Experiment

The first experiment [9] places the BBN UAV platform [3, 37] platform under the control of QARMA as depicted in Figure 7.1. There are two UAV systems in the experiment each consisting of a sender, distributor, receiver, and video display. Both sender applications reside on one host. Both distributor applications reside on another host, and the receiver and video displays are allocated to a third host. All major QARMA components were allocated to a fourth host separate from the UAV applications. Other components, such as host monitors, were allocated to each host.

---

1The results, graphs and tables in this section are published in [9] and in [10].
During the experiment, the first UAV system is given a higher importance than the second UAV system. Load is introduced at 90 second intervals as described in our original publication [9]. The baseline experiment is conducted without QARMA management. In the qarma experiment, the UAV systems are under the control of QARMA. The data collected from the experiment is analyzed by two metrics: latency measured in microseconds and frame rate for the receivers measured in frames-per-second.

Figure 7.2 contains graphs of the latency of the first UAV system during both the baseline and qarma experiments. The increased number of spikes in the baseline graph is indicative of the local adaptation policies of the UAV without central control. Each UAV in the system performs a test-and-check routine every 30 seconds to see if the UAV can sustain a higher frame rate causing the spikes in the graphs.

Instead of relying on test-and-check mechanisms, QARMA only changes the frame rate of the UAV when there is evidence that the UAV system can successfully execute at a higher frame rate. The spikes that are present in the right-hand graph in Figure 7.2 occur...
<table>
<thead>
<tr>
<th>Experiments</th>
<th>Latency (ms)</th>
<th>Frame Rate (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Min</td>
</tr>
<tr>
<td>qarma</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UAV1</td>
<td>106.87</td>
<td>35.00</td>
</tr>
<tr>
<td>UAV2</td>
<td>88.92</td>
<td>33.00</td>
</tr>
<tr>
<td>baseline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UAV1</td>
<td>252.48</td>
<td>27.00</td>
</tr>
<tr>
<td>UAV2</td>
<td>244.63</td>
<td>30.00</td>
</tr>
</tbody>
</table>

Table 7.1: UAV latency and frame rate statistics

only when load is increasing in the system. QARMA improves the average latency for the delivery of frames and provides better stability than the local adaptation policy. The latency statistics from the experiment are summarized in Table 7.1.

The collected frame rate metrics are also summarized in Table 7.1. The local adaptation policies of the baseline experiment results in both UAV systems maintaining the same frame rate during most of the experiment regardless of the importance of each UAV system. Using the notion of system importance, QARMA is able to ensure that the more important UAV system achieves a higher frame rate than the less important UAV system when resources are available.

The more important UAV stream averaged 3.27 frames-per-second more in the qarma experiment than in the baseline experiment The increased performance of UAV1 comes at expense of the less important UAV2 system which averaged 3.32 frames-pers-second less in the qarma experiment than in the baseline experiment.
7.2 QARMA / Dynamic Scheduling Service Experiment

In the second experiment [10], QARMA is integrated with existing middleware services. The Dynamic Scheduling Service (DSS) is a middleware service that coordinates local schedulers such that the distributable threads admitted to the system remain schedulable. Kokyu is an end-system scheduler that can be configured to enforce scheduling policies such as EDF on an end-system. The architectural description of the integration of QARMA with the DSS and with the end-system scheduler Kokyu is described in [10], a joint paper from Ohio University, University of Rhode Island and University of Washington in St. Louis.

One shortcoming of the DSS at the time of this experiment is that when overload is detected in the system, the DSS is only able to cancel existing tasks or reject new tasks. However, in this experiment, the DSS is able to invoke the Resource Management Service when overload is detected in order to adjust the service attributes of applications. The Resource Management Service chooses new service attribute settings so that the schedulability of the system is maintained. Likewise, the QARMA system alone is not able to enforce that each end-system schedules tasks such that the real-time performance is guaranteed. The DSS enforces the real-time scheduling behavior leading to improved predictability of the system and a decrease in the number of latency violations as summarized in Table 7.2 from [10].

The scenario for this experiment consists of six UAV systems each having 10 possible frame rates, 1 through 10. The UAV systems are simulated by the RM Bench tool. Task 1 refers to the distributed task of sending the video for UAV1 through the distributor to the
Table 7.2: QARMA and DSS integration results

receiver. The tasks are ordered by importance such that Task 1 is the least important task and Task 6 is the most important task.

The same scenario is evaluated in three distinct experiments. The baseline experiment allows the six tasks to execute without any resource management middleware. The qarma-only experiment allows QARMA to manage the tasks by adjusting the frame rates in
response to processor overload and underload detection, but without the help of the DSS. The qarma-dss experiment allows the DSS to trigger the Resource Management Service when tasks are admitted or released from the system. The DSS coordinates the end-system schedulers to enforce EDF scheduling on each end-system.

The baseline experiment indicates that without resource management the systems would continue to run at the highest frame rates and miss a lot of the deadlines. The qarma-only experiment improves the situation by adjusting the frame rates of the less important tasks when there is competition for the resources. This results in lower frame rates for tasks 1 through 4, but reduces the percentage of missed deadlines from 26% to 10%. The qarma-dss experiment further reduces the percentage of missed deadlines to 1%, where the missed deadlines occur only at critical moments when new tasks are admitted to the system causing transient overload on the resources. Extensive details of the integrated experiments with the DSS are provided in [10].
CHAPTER 8

CONCLUSIONS

The Quality-based Adaptive Resource Management Architecture, QARMA, provides both a generic framework and a collection of middleware. The framework summarizes the resource management process as six functions: (1) monitoring, (2) detection, (3) decision-making, (4) enactment, (5) information storage and sharing, and (6) visualization. The middleware components are an instantiation of a resource management solution based directly on the framework.

The notion of service attributes and environment attributes are introduced in the system model. These attributes provide a generic mechanism for defining both parameters for controlling application quality requirements and for modeling the operating environment of the system. The ability to model user-perceived benefit functions allows QARMA to control service attributes causing applications to adapt to changing environmental conditions such that real-time requirements are satisfied and user benefit is maximized.

The Resource Management Service is the core decision-making component. A greedy resource allocation algorithm is presented that is both efficient and provides promising experimental results. The algorithm makes use of algorithmic models based on the system model and that encapsulate existing distributed scheduling theory and algorithms.

Experimental results using QARMA to control an initial version of the UAV system [9] demonstrate that QARMA can in fact provide enhanced performance for dynamic real-time
systems. Integration with existing middleware services [10, 11] further show the flexibility of the QARMA architecture in cooperating with other management services.
BIBLIOGRAPHY


[8] M. Wand, “A semantics for advice and dynamic join points in aspect-oriented pro-


gration of ohio university resource management with university of utah cpu broker,”
tech. rep., Center for Intelligent, Distributed and Dependable Systems, Ohio University, March 2004.


dynamic systems,” in *In Proceedings of the 9th International Workshop on Parallel

[36] L. Welch, D. Lawrence, and D. Alexander, “Feedback control resource management
using a posteriori workload characterizations,” in *Proc. of the 39th IEEE Conference
on Decision and Control*, December 2000.

service in a distributed video application by an adaptive middleware framework,” in

[38] L. Welch, P. Werme, L. Fontenot, M. Masters, B. Shirazi, B. Ravindran, and D. Mills,
“Adaptive qos and resource management using a posteriori workload characteriza-
tions,” in *Proc. of the 5th IEEE Real-Time Technology and Applications Symposium*,

Ohio University, 2002.

[40] J. Sun, *Fixed-Priority End-to-End Scheduling in Distributed Real-Time Systems*. PhD
thesis, Department of Computer Science, University of Illinois at Urbana-Champaign,
1997.


