Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 250895, 20 pages
doi:10.1155/2008/250895

Research Article

Design and Performance Evaluation of an Adaptive Resource Management Framework for Distributed Real-Time and Embedded Systems

Nishanth Shankaran,[1] Nilabja Roy,[1] Douglas C. Schmidt,[1] Xenofon D. Koutsoukos,[1] Yingming Chen,[2] and Chenyang Lu[2]

[1] *The Electrical Engineering and Computer Science Department, Vanderbilt University, Nashville, TN 37235, USA*
[2] *Department of Computer Science and Engineering, Washington University, St. Louis, MO 63130, USA*

Correspondence should be addressed to Nishanth Shankaran, nshankar@dre.vanderbilt.edu

Received 8 February 2007; Revised 6 November 2007; Accepted 2 January 2008

Recommended by Michael Harbour

Achieving end-to-end quality of service (QoS) in distributed real-time embedded (DRE) systems require QoS support and enforcement from their underlying operating platforms that integrates many real-time capabilities, such as QoS-enabled network protocols, real-time operating system scheduling mechanisms and policies, and real-time middleware services. As standards-based quality of service (QoS) enabled component middleware automates integration and configuration activities, it is increasingly being used as a platform for developing open DRE systems that execute in environments where operational conditions, input workload, and resource availability cannot be characterized accurately a priori. Although QoS-enabled component middleware offers many desirable features, however, it historically lacked the ability to allocate resources efficiently and enable the system to adapt to fluctuations in input workload, resource availability, and operating conditions. This paper presents three contributions to research on adaptive resource management for component-based open DRE systems. First, we describe the structure and functionality of the resource allocation and control engine (RACE), which is an open-source adaptive resource management framework built atop standards-based QoS-enabled component middleware. Second, we demonstrate and evaluate the effectiveness of RACE in the context of a representative open DRE system: NASA's magnetospheric multiscale mission system. Third, we present an empirical evaluation of RACE's scalability as the number of nodes and applications in a DRE system grows. Our results show that RACE is a scalable adaptive resource management framework and yields a predictable and high-performance system, even in the face of changing operational conditions and input workload.

1. INTRODUCTION

Distributed real-time and embedded (DRE) systems form the core of many large scale mission-critical domains. In these systems, achieving end-to-end quality of service (QoS) requires integrating a range of real-time capabilities, such as QoS-enabled network protocols, real-time operating system scheduling mechanisms and policies, and real-time middleware services, across the system domain. Although existing research and solutions [1, 2] focus on improving the performance and QoS of individual capabilities of the system (such as operating system scheduling mechanism and policies), they are not sufficient for DRE systems as these systems require integrating a range of real-time capabilities across the system domain. Conventional QoS-enabled middleware technologies, such as real-time CORBA [3] and the real-time Java [4], have been used extensively as an operating platforms to build DRE systems as they support explicit configuration of QoS aspects (such as priority and threading models), and provide many desirable real-time features (such as priority propagation, scheduling services, and explicit binding of network connections).

QoS-enabled middleware technologies have traditionally focused on DRE systems that operate in *closed* environments where operating conditions, input workloads, and resource availability are known in advance and do not vary significantly at run-time. An example of a closed DRE system is an avionics mission computer [5], where the penalty of

not meeting a QoS requirement (such as deadline) can result in the failure of the entire system or mission. Conventional QoS-enabled middleware technologies are insufficient, however, for DRE systems that execute in *open* environments where operational conditions, input workload, and resource availability cannot be characterized accurately a priori. Examples of open DRE systems include shipboard computing environments [6], multisatellite missions [7]; and intelligence, surveillance, and reconnaissance missions [8].

Specifying and enforcing end-to-end QoS is an important and challenging issue for open systems DRE due to their unique characteristics, including (1) constraints in multiple resources (e.g., limited computing power and network bandwidth) and (2) highly fluctuating resource availability and input workload. At the heart of achieving end-to-end QoS are resource management techniques that enable open DRE systems to *adapt* to dynamic changes in resource availability and demand. In earlier work, we developed adaptive resource management *algorithms* (such as EUCON [9], DEUCON [10], HySUCON [11], and FMUF [12]) and *architectures*, such as HiDRA [13] based on control-theoretic techniques. We then developed FC-ORB [14], which is a QoS-enabled adaptive middleware that implements the EUCON algorithm to handle fluctuations in application workload and system resource availability.

A limitation with our prior work, however, is that it tightly coupled resource management algorithms within particular middleware platforms, which made it hard to enhance the algorithms without redeveloping significant portions of the middleware. For example, since the design and implementation of FC-ORB were closely tied to the EUCON adaptive resource management algorithm, significant modifications to the middleware were needed to support other resource management algorithms, such as DEUCON, HySUCON, or FMUF. Object-oriented frameworks have traditionally been used to factor out many reusable general-purpose and domain-specific services from DRE systems and applications [15]; however, to alleviate the tight coupling between resource management algorithms and middleware platforms and improve flexibility, this paper presents an *adaptive resource management framework* for open DRE systems. Contributions of this paper to the study of adaptive resource management solutions for open DRE systems include the following.

(i) *The design of a resource allocation and control engine (RACE)*, which is a fully customizable and configurable adaptive resource management framework for open DRE systems. RACE decouples adaptive resource management algorithms from the middleware implementation, thereby enabling the usage of various resource management algorithms without the need for redeveloping significant portions of the middleware. RACE can be configured to support a range of algorithms for adaptive resource management without requiring modifications to the underlying middleware. To enable the seamless integration of resource allocation and control algorithms into DRE systems, RACE enables the deployment and configuration of feedback control loops. RACE, therefore, complements theoretical research on adaptive resource management algorithms that provide a model and theoretical analysis of system performance.

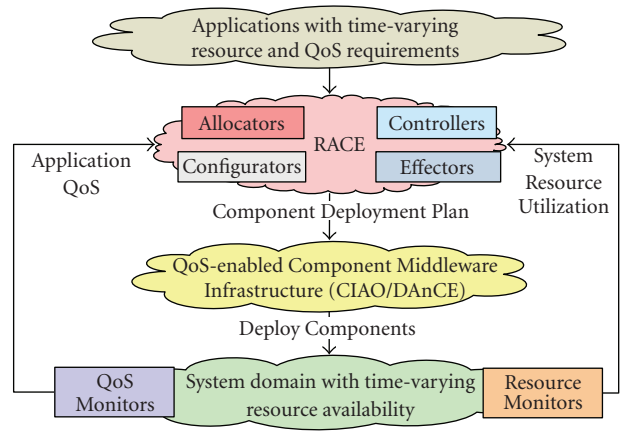


Figure 1: A resource allocation and control engine (RACE) for open DRE systems.

As shown in Figure 1, RACE provides (1) *resource monitors* that track utilization of various system resources, such as CPU, memory, and network bandwidth; (2) *QoS monitors* that track application QoS, such as end-to-end delay; (3) *resource allocators* that allocate resource to components based on their resource requirements and current availability of system resources; (4) *configurators* that configure middleware QoS parameters of application components; (5) *controllers* that compute end-to-end adaptation decisions based on control algorithms to ensure that QoS requirements of applications are met; and (6) *effectors* that perform controller-recommended adaptations.

(ii) *Evaluate the effectiveness of RACE in the context of NASA's magnetospheric multiscale system (MMS) mission*, which is representative open DRE system. The MMS mission system consists of a constellation of spacecrafts that maintain a specific formation while orbiting over a region of scientific interest. In these spacecrafts, availability of resource such as processing power (CPU), storage, network bandwidth, and power (battery) are limited and subjected to run-time variations. Moreover, resource utilization by, and input workload of, applications that execute in this system cannot be accurately characterized a priori. This paper evaluates the adaptive resource management capabilities of RACE in the context of this representative open DRE system. Our results demonstrate that when adaptive resource management algorithms for DRE systems are implemented using RACE, they yield a predictable and high-performance system, even in the face of changing operational conditions and workloads.

(iii) *The empirical evaluation of RACE's scalability* as the number of nodes and applications in a DRE system grows. Scalability is an integral property of a framework as it determines the framework's applicability. Since open DRE systems comprise large number of nodes and applications, to determine whether RACE can be applied to such systems, we empirically evaluate RACE's scalability as the number of applications and nodes in the system increases. Our results

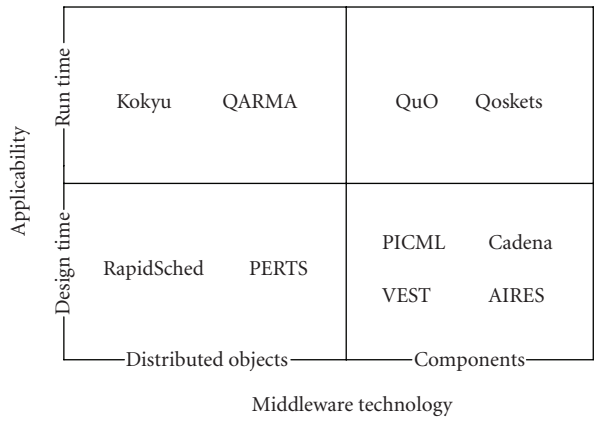


Figure 2: Taxonomy of related research.

demonstrate that RACE scales as well as the number of applications and nodes in the system increases, and therefore can be applied to a wide range of open DRE systems.

The remainder of the paper is organized as follows: Section 2 compares our research on RACE with related work; Section 3 motivates the use of RACE in the context of a representative DRE system case study; Section 4 describes the architecture of RACE and shows how it aids in the development of the case study described in Section 3; Section 5 empirically evaluates the performance of the DRE system when control algorithms are used in conjunction with RACE and also presents an empirical measure of RACE's scalability as the number of applications and nodes in the system grows; and Section 6 presents concluding remarks.

2. RESEARCH BACKGROUND AND RELATED WORK COMPARISON

This section presents an overview of existing middleware technologies that have been used to develop open DRE system and also compares our work on RACE with related research on building open DRE systems. As in Figure 2 and described below, we classify this research along two orthogonal dimensions: (1) QoS-enabled DOC middleware versus QoS-enabled component middleware, and (2) design-time versus run-time QoS configuration, optimization, analysis, and evaluation of constraints, such as timing, memory, and CPU.

2.1. Overview of conventional and QoS-enabled DOC middleware

Conventional middleware technologies for distributed object computing (DOC), such as the object management group (OMG)'s CORBA [16] and Sun's Java RMI [17], encapsulates and enhances native OS mechanisms to create reusable network programming components. These technologies provide a layer of abstraction that shields application developers from the low-level platform-specific details and define higher-level distributed programming models whose reusable API's and components automate and extend native OS capabilities.

Conventional DOC middleware technologies, however, address only *functional* aspects of system/application development such as how to define and integrate object interfaces and implementations. They do not address QoS aspects of system/-application development such as how to (1) define and enforce application timing requirements, (2) allocate resources to applications, and (3) configure OS and network QoS policies such as priorities for application processes and/or threads. As a result, the code that configures and manages QoS aspects often become entangled with the application code. These limitations with conventional DOC middleware have been addressed by the following run-time platforms and design-time tools.

(i) *Run-time:* early work on resource management middleware for shipboard DRE systems presented in [18, 19] motivated the need for adaptive resource management middleware. This work was further extended by QARMA [20], which provides resource management as a *service* for existing QoS-enabled DOC middleware, such as RT-CORBA. Kokyu [21] also enhances RT-CORBA QoS-enabled DOC middleware by providing a portable middleware scheduling framework that offers flexible scheduling and dispatching services. Kokyu performs feasibility analysis based on estimated worst case execution times of applications to determine if a set of applications is *schedulable*. Resource requirements of applications, such as memory and network bandwidth, are not captured and taken into consideration by Kokyu. Moreover, Kokyu lacks the capability to track utilization of various system resources as well as QoS of applications. To address these limitations, research presented in [22] enhances QoS-enabled DOC middleware by combining Kokyu and QARMA.

(ii) *Design-time:* RapidSched [23] enhances QoS-enabled DOC middleware, such as RT-CORBA, by computing and enforcing distributed priorities. RapidSched uses PERTS [24] to specify real-time information, such as deadline, estimated execution times, and resource requirements. Static schedulability analysis (such as rate monotonic analysis) is then performed and priorities are computed for each CORBA object in the system. After the priorities are computed, RapidSched uses RT-CORBA features to enforce these computed priorities.

2.2. Overview of conventional and QoS-enabled component middleware

Conventional component middleware technologies, such as the CORBA component model (CCM) [25] and enterprise Java beans [26, 27], provide capabilities that addresses the limitation of DOC middleware technologies in the context of system design and development. Examples of additional capabilities offered by conventional component middleware compared to conventional DOC middleware technology include (1) standardized interfaces for application component interaction, (2) model-based tools for deploying and interconnecting components, and (3) standards-based mechanisms for installing, initializing, and configuring application

components, thus separating concerns of application development, configuration, and deployment.

Although conventional component middleware support the design and development of large scale distributed systems, they do not address the QoS limitations of DOC middleware. Therefore, conventional component middleware can support large scale enterprise distributed systems, but not DRE systems that have the stringent QoS requirements. These limitations with conventional component-based middleware have been addressed by the following run-time platforms and design-time tools.

(i) *Run-time:* QoS provisioning frameworks, such as QuO [28] and Qoskets [8, 29, 30], help ensure desired performance of DRE systems built atop QoS-enabled DOC middleware and QoS-enabled component middleware, respectively. When applications are designed using Qoskets (1) resources are dynamically (re)allocated to applications in response to changing operational conditions and/or input workload and (2) application parameters are fine-tuned to ensure that allocated resources are used effectively. With this approach, however, applications are augmented explicitly at design-time with Qosket components, such as monitors, controllers, and effectors. This approach thus requires redesign and reassembly of existing applications built without Qoskets. When applications are generated at run-time (e.g., by intelligent mission planners [31]), this approach would require planners to augment the applications with Qosket components, which may be infeasible since planners are designed and built to solve mission goals and not perform such platform-/middleware-specific operations.

(ii) *Design-time:* Cadena [32] is an integrated environment for developing and verifying component-based DRE systems by applying static analysis, model-checking, and lightweight formal methods. Cadena also provides a component assembly framework for visualizing and developing components and their connections. VEST [33] is a design assistant tool based on the *generic modeling environment* [34] that enables embedded system composition from component libraries and checks whether timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. AIRES [35] is a similar tool that provides the means to map design-time models of component composition with real-time requirements to run-time models that weave together timing and scheduling attributes. The research presented in [36] describes a design assistant tool, based on MAST [37], that comprises a DSML and a suite of analysis and system QoS configuration tools and enables composition, schedulability analysis, and assignment of operating system priority for application components.

Some design-time tools, such as AIRES, VEST, and those presented in [36], use *estimates*, such as estimated worst case execution time, estimated CPU, memory, and/or network bandwidth requirements. These tools are targeted for systems that execute in *closed* environments, where operational conditions, input workload, and resource availability can be characterized accurately a priori. Since RACE tracks and manages utilization of various system resources, as well as application QoS, it can be used in conjunction with these tools to build open DRE systems.

2.3. Comparing RACE with related work

Our work on RACE extends earlier work on QoS-enabled DOC middleware by providing an adaptive resource management framework for open DRE systems built atop QoS-enabled component middleware. DRE systems built using RACE benefit from (1) adaptive resource management capabilities of RACE and (2) additional capabilities offered by QoS-enabled component middleware compared to QoS-enabled DOC middleware, as discussed in Section 2.2.

Compared to related research presented in [18–20], RACE is an adaptive resource management framework that can be customized and configured using model-driven deployment and configuration tools such as the *platform-independent component modeling language* (PICML) [38]. Moreover, RACE provides adaptive resource and QoS management capabilities more transparently and nonintrusively than Kokyu, QuO, and Qoskets. In particular, it allocates CPU, memory, and networking resources to application components and tracks and manages utilization of various system resources, as well as application QoS. In contrast to our own earlier work on QoS-enabled DOC middleware, such as FC-ORB [14] and HiDRA [13], RACE is a QoS-enabled component middleware framework that enables the deployment and configuration of feedback control loops in DRE systems.

In summary, RACE's novelty stems from its combination of (1) design-time model-driven tools that can both design applications and customize and configure RACE itself, (2) QoS-enabled component middleware run-time platforms, and (3) research on control-theoretic adaptive resource management. RACE can be used to deploy and manage component-based applications that are composed at design-time via model-driven tools, as well as at run-time by *intelligent mission planners* [39], such as SA-POP [31].

3. CASE STUDY: MAGNETOSPHERIC MULTISCALE (MMS) MISSION DRE SYSTEM

This section presents an overview of NASA's magnetospheric multiscale (MMS) mission [40] as a case study to motivate the need for RACE in the context of open DRE systems. We also describe the resource and QoS management challenges involved in developing the MMS mission using QoS-enabled component middleware.

3.1. MMS mission system overview

NASA's MMS mission system is a representative open DRE system consisting of several interacting subsystems (both in-flight and stationary) with a variety of complex QoS requirements. As shown in Figure 3, the MMS mission consists of a constellation of five spacecrafts that maintain a specific formation while orbiting over a region of scientific interest. This constellation collects science data pertaining to the earth's plasma and magnetic activities while in orbit and send it to a ground station for further processing. In the MMS mission spacecrafts, availability of resource such as processing power (CPU), storage, network bandwidth, and power (battery) are

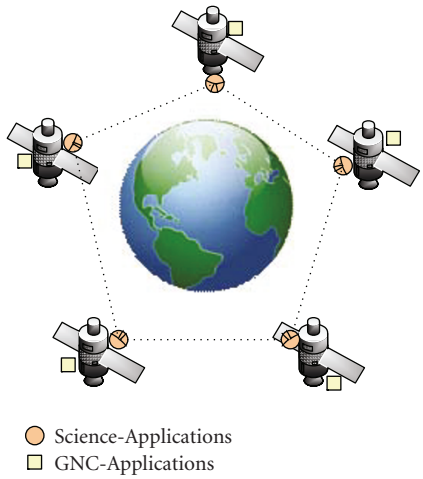


Figure 3: MMS mission system.

limited and subjected to run-time variations. Moreover, resource utilization by, and input workload of, applications that execute in this system cannot be accurately characterized a priori. These properties make the MMS mission system an open DRE system.

Applications executing in this system can be classified as guidance, navigation, and control (GNC) applications and science applications. The GNC applications are responsible for maintaining the spacecraft within the specified orbit. The science applications are responsible for collecting science data, compressing and storing the data, and transmitting the stored data to the ground station for further processing.

As shown in Figure 3, GNC applications are localized to a single spacecraft. Science applications tend to span the entire spacecraft constellation, *that is*, all spacecrafts in the constellation have to coordinate with each other to achieve the goals of the science mission. GNC applications are considered *hard real-time* applications (i.e., the penalty of not meeting QoS requirement(s) of these applications is very high, often fatal to the mission), whereas science applications are considered *soft real-time* applications (i.e., the penalty of not meeting QoS requirement(s) of these applications is high, but not fatal to the mission).

Science applications operate in three modes: *slow survey*, *fast survey*, and *burst* mode. Science applications switch from one mode to another in reaction to one or more *events of interest*. For example, for a science application that monitors the earth's plasma activity, the *slow* survey mode is entered outside the regions of scientific interests and enables only a minimal set of data acquisition (primarily for health monitoring). The *fast* survey mode is entered when the spacecrafts are within one or more regions of interest, which enables data acquisition for all payload sensors at a moderate rate. If plasma activity is detected while in fast survey mode, the application enters *burst* mode, which results in data collection at the highest data rates. Resource utilization by, and importance of, a science application is determined by its mode of operation, which is summarized by Table 1.

Table 1: Characteristics of science application.

Mode	Relative importance	Resource consumption
Slow survey	Low	Low
Fast survey	Medium	Medium
Burst	High	High

Each spacecraft consists of an onboard intelligent mission planner, such as the *spreading activation partial-order planner* (SA-POP) [31] that decomposes overall mission goal(s) into GNC and science applications that can be executed concurrently. SA-POP employs decision-theoretic methods and other AI schemes (such as hierarchical task decomposition) to decompose mission goals into navigation, control, data gathering, and data processing applications. In addition to initial generation of GNC and science applications, SA-POP incrementally generates new applications in response to changing mission goals and/or degraded performance reported by onboard mission monitors.

We have developed a prototype implementation of the MMS mission systems in conjunction with our colleagues at Lockheed Martin Advanced Technology Center, Palo Alto, California. In our prototype implementation, we used the *component-integrated ACE ORB* (CIAO) [41] and *deployment and configuration engine* (DAnCE) [42] as the QoS-enabled component middleware platform. Each spacecraft uses SA-POP as its onboard intelligent mission planner.

3.2. Adaptive resource management requirements of the MMS mission system

As discussed in Section 2.2, the use of QoS-enabled component middleware to develop open DRE systems, such as the NASA MMS mission, can significantly improve the design, development, evolution, and maintenance of these systems. In the absence of an adaptive resource management framework, however, several key requirements remain unresolved when such systems are built in the absence of an adaptive resource management framework. To motivate the need for RACE, the remainder of this section presents the key resource and QoS management requirements that we addressed while building our prototype of the MMS mission DRE system.

3.2.1. Requirement 1: resource allocation to applications

Applications generated by SA-POP are *resource sensitive*, *that is*, QoS is affected significantly if an application does not receive the required CPU time and network bandwidth within bounded delay. Moreover, in open DRE systems like the MMS mission, input workload affects utilization of system resources and QoS of applications. Utilization of system resources and QoS of applications may therefore vary significantly from their estimated values. Due to the operating conditions for open DRE systems, system resource availability, such as available network bandwidth, may also be time variant.

A resource management framework therefore needs to (1) monitor the current utilization of system resources,

(2) allocate resources in a timely fashion to applications such that their resource requirements are met using resource allocation algorithms such as PBFD [43], and (3) support multiple resource allocation strategies since CPU and memory utilization overhead might be associated with implementations of resource allocation algorithms themselves and select the appropriate one(s) depending on properties of the application and the overheads associated with various implementations. Section 4.2.1 describes how RACE performs online resource allocation to application components to address this requirement.

3.2.2. Requirement 2: configuring platform-specific QoS parameters

The QoS experienced by applications depend on various platform-specific real-time QoS configurations including (1) *QoS configuration of the QoS-enabled component middleware*, such as priority model, threading model, and request processing policy; (2) *operating system QoS configuration*, such as real-time priorities of the process(es) and thread(s) that host and execute within the components, respectively; and (3) *networks QoS configurations*, such as *diffserv* code points of the component interconnections. Since these configurations are platform-specific, it is tedious and error-prone for system developers or SA-POP to specify them in isolation.

An adaptive resource management framework therefore needs to provide abstractions that shield developers and/or SA-POP from low-level platform-specific details and define higher-level QoS specification models. System developers and/or intelligent mission planners should be able to specify QoS characteristics of the application such as QoS requirements and relative importance, and the adaptive resource management framework should then configure the platform-specific parameters accordingly. Section 4.2.2 describes how RACE provides a higher level of abstractions and shield system developers and SA-POP from low-level platform-specific details to address this requirement.

3.2.3. Requirement 3: enabling dynamic system adaptation and ensuring QoS requirements are met

When applications are deployed and initialized, resources are allocated to application components based on the *estimated* resource utilization and estimated/current availability of system resources. In open DRE systems, however, *actual* resource utilization of applications might be significantly different than their estimated values, as well as availability of system resources vary dynamically. Moreover, for applications executing in these systems, the relation between input workload, resource utilization, and QoS cannot be characterized a priori.

An adaptive resource management framework therefore needs to provide monitors that track system resource utilization, as well as QoS of applications, at run-time. Although some QoS properties (such as accuracy, precision, and fidelity of the produced output) are application-specific, certain QoS (such as *end-to-end latency* and throughput) can be tracked by the framework transparently to the application.

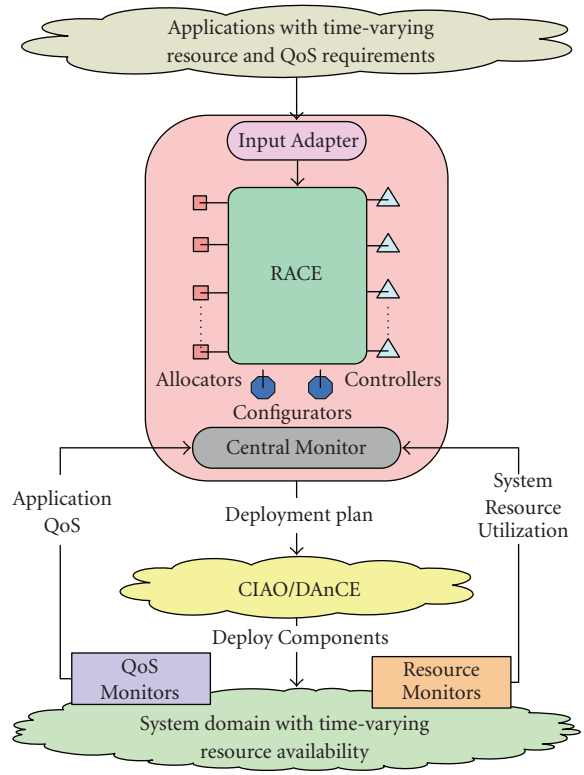


Figure 4: Detailed design of RACE.

However, customization and configuration of the framework with domain-specific monitors (both platform-specific resource monitors and application-specific QoS monitors) should be possible. In addition, the framework needs to enable the system to *adapt* to dynamic changes, such as variations in operational conditions, input workload, and/or resource availability. Section 4.2.3 demonstrates how RACE performs system adaptation and ensures QoS requirements of applications are met to address this requirement.

4. STRUCTURE AND FUNCTIONALITY OF RACE

This section describes the structure and functionality of RACE. RACE supports open DRE systems built atop CIAO, which is an open-source implementation of lightweight CCM. All entities of RACE themselves are designed and implemented as CCM components, so RACE's *Allocators* and *Controllers* can be configured to support a range of resource allocation and control algorithms using model-driven tools, such as PICML.

4.1. Design of RACE

Figure 4 elaborates the earlier architectural overview of RACE in Figure 1 and shows how the detailed design of RACE is composed of the following components: (1) *InputAdapter*, (2) *CentralMonitor*, (3) *Allocators*, (4) *Configurators*, (5) *Controllers*, and (6) *Effectors*. RACE monitors application QoS and system resource usage via its *Resource*

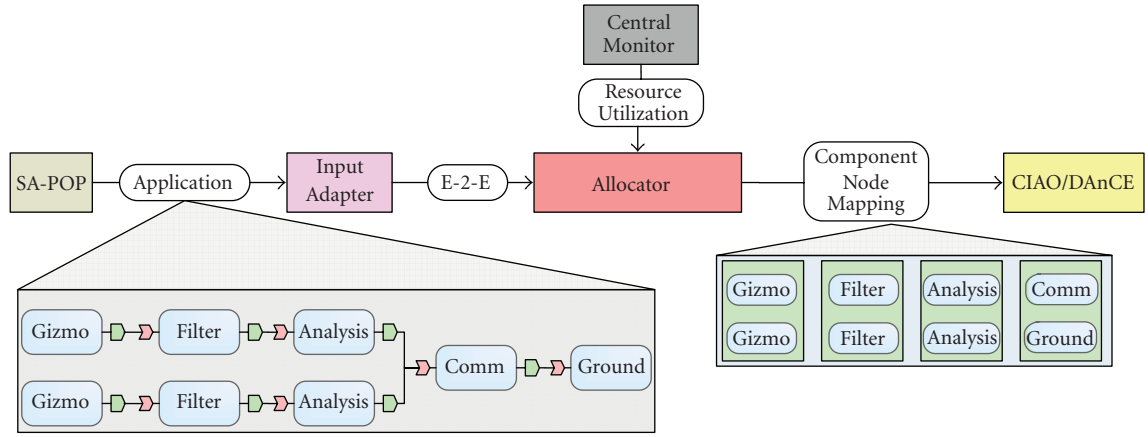


Figure 5: Resource allocation to application components using RACE.

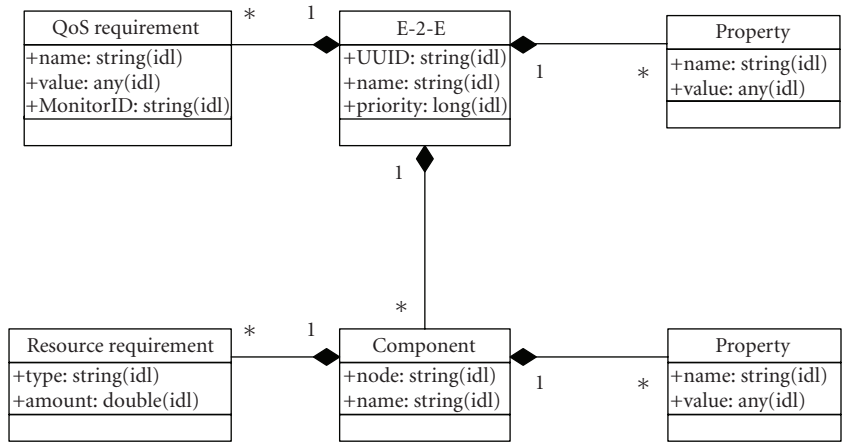


Figure 6: Main entities of RACE's E-2-E IDL structure.

Monitor, *QoS-Monitors*, *Node Monitors*, and *Central Monitor*. Each component in RACE is described below in the context of the overall adaptive resource management challenge it addresses.

4.1.1. *Challenge 1: domain-specific representation of application metadata*

Problem

End-to-end applications can be composed either at design-time or at run-time. At design-time, CCM-based end-to-end applications are composed using model-driven tools, such as PICML; and at run-time, they can be composed by intelligent mission planners like SA-POP. When an application is composed using PICML, metadata describing the application is captured in XML files based on the *PackageConfiguration* schema defined by the object management group's deployment and configuration specification [44]. When applications are generated during run-time by SA-POP, metadata is captured in an in-memory structure defined by the planner.

Solution: domain-specific customization and configuration of RACE's adapters

During design-time, RACE can be configured using PICML and an *InputAdapter* appropriate for the domain/system can be selected. For example, to manage a system in which applications are constructed at design-time using PICML, RACE can be configured with the *PICMLInputAdapter*; and to manage a system in which applications are constructed at run-time using SA-POP, RACE can be configured with the *SAPOPInputAdapter*. As shown in Figure 5, the *InputAdapter* parses the metadata that describes the application into an in-memory end-to-end (*E-2-E*) IDL structure that is internal to RACE. Key entities of the *E-2-E* IDL structure are shown in Figure 6.

The *E-2-E* IDL structure populated by the *InputAdapter* contains information regarding the application, including (1) components that make up the application and their resource requirement(s), (2) interconnections between the components, (3) application QoS properties (such relative priority) and QoS requirement(s) (such as end-to-end delay), and (4) mapping components onto domain nodes. The

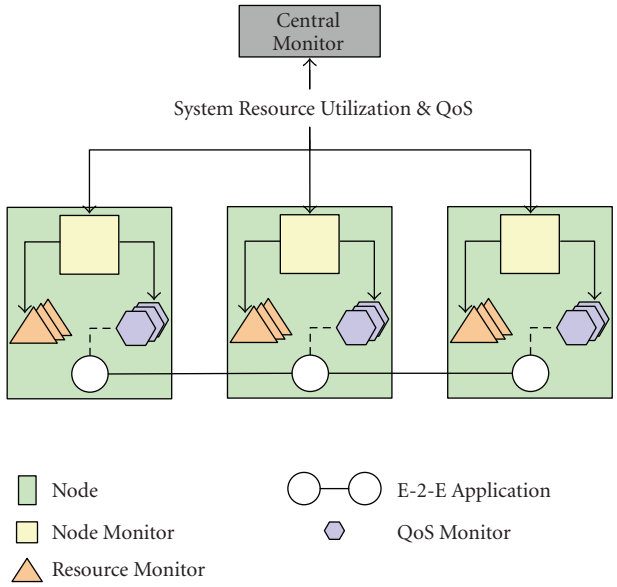


Figure 7: Architecture of monitoring framework.

mapping of components onto nodes need not be specified in the metadata that describes the application which is given to RACE. If a mapping is specified, it is honored by RACE; if not, a mapping is determined at run-time by RACE's *Allocators*.

4.1.2. Challenge 2: efficient monitoring of system resource utilization and application QoS

Problem

In open DRE systems, input workload, application QoS, and utilization and availability of system resource are subject to dynamic variations. In order to ensure application QoS requirements are met, as well as utilization of system resources are within specified bounds, application QoS and utilization/availability of system resources are to be monitored periodically. The key challenge lies in designing and implementing a resource and QoS monitoring architecture that scales as well as the number of applications and nodes in the system increase.

Solution: hierarchical QoS and resource monitoring architecture

RACE's monitoring framework is composed of the *Central Monitor*, *Node Monitors*, *Resource Monitors*, and *QoS Monitors*. These components track resource utilization by, and QoS of, application components. As shown in Figure 7, RACE's *Monitors* are structured in the following hierarchical fashion. A *Resource Monitor* collects resource utilization metrics of a specific resource, such as CPU or memory. A *QoS Monitor* collects specific QoS metrics of an application, such as end-to-end latency or throughput. A *Node Monitor* tracks the QoS of all the applications running on a node as well as the resource utilization of that node. Finally, a *Central Monitor* tracks the QoS of all the applications running the entire system, which captures the system QoS, as well as the resource utilization of the entire system, which captures the system resource utilization.

Resource Monitors use the operating system facilities, such as */proc* file system in *Linux/Unix* operating systems and the *system registry* in *Windows* operating systems, to collect resource utilization metrics of that node. As the resource monitors are implemented as shared libraries that can be loaded at run-time, RACE can be configured with new-/domain-specific resource monitors without making any modifications to other entities of RACE. *QoS-Monitors* are implemented as software modules that collect end-to-end latency and throughput metrics of an application and are dynamically installed into a running system using DyInst [45]. This approach ensure rebuilding, reimplementation, or restarting of already running application components are not required. Moreover, with this approach, *QoS-Monitors* can be turned on or off on demand at run-time.

The primary metric that we use to measure the performance of our monitoring framework is *monitoring delay*, which is defined as the time taken to obtain a snapshot of the entire system in terms of resource utilization and QoS. To minimize the monitoring delay and ensure that RACE's monitoring architecture scales as the number of applications and nodes in the system increase, the RACE's monitoring architecture is structured in a hierarchical fashion. We validate this claim in Section 5.

4.1.3. Challenge 3: resource allocation

Problem

Applications executing in open DRE systems are resource sensitive and require multiple resources such as memory, CPU, and network bandwidth. In open DRE systems, resources allocation cannot be performed during design-time as system resource availability may be time variant. Moreover, input workload affects the utilization of system resources by already executing applications. Therefore, the key challenge lies in allocating various systems resources to application components in a timely fashion.

Solution:online resource allocation

RACE's *Allocators* implement resource allocation algorithms and allocate various domain resources (such as CPU, memory, and network bandwidth) to application components by determining the mapping of components onto nodes in the system domain. For certain applications, *static* mapping between components and nodes may be specified at design-time by system developers. To honor these static mappings, RACE therefore provides a *static allocator* that ensures components are allocated to nodes in accordance with the static mapping specified in the application's metadata. If no static mapping is specified, however, *dynamic allocators* determine the component to node mapping at run-time based on resource requirements of the components and current resource availability on the various nodes in the domain. As shown in

Figure 5, input to *Allocators* include the *E-2-E* IDL structure corresponding to the application and the current utilization of system resources.

The current version of RACE provides the following *Allocators*: (1) a single dimension binpacker [46] that makes allocation decisions based on either CPU, memory, or network bandwidth requirements and availability, (2) a multidimensional binpacker—partitioned breadth first decreasing allocator [43]—that makes allocation decisions based on CPU, memory, and network bandwidth requirements and availability, and (3) a static allocator. Metadata is associated with each allocator and captures its type (i.e., static, single dimension binpacking, or multidimensional binpacker) and associated resource overhead (such as CPU and memory utilization). Since *Allocators* themselves are CCM components, RACE can be configured with new *Allocators* by using PICML.

4.1.4. Challenge 4: accidental complexities in configuring platform-specific QoS parameters

Problem

As described in Section 3.2.2, real-time QoS *configuration* of the underlying component middleware, operating system, and network affects the QoS of applications executing in open DRE systems. Since these configurations are platform-specific, it is tedious and error-prone for system developers or SA-POP to specify them in isolation.

Solution: automate configuration of platform-specific parameters

As shown in Figure 8, RACE's *Configurators* determine values for various low-level platform-specific QoS parameters, such as middleware, operating system, and network settings for an application based on its QoS characteristics and requirements such as relative importance and end-to-end delay. For example, the *MiddleWareConfigurator* configures component lightweight CCM policies, such as threading policy, priority model, and request processing policy based on the class of the application (*important* and *best effort*). The *OperatingSystemConfigurator* configures operating system parameters, such as the priorities of the *component servers* that host the components based on rate monotonic scheduling (RMS) [46] or based on criticality (relative importance) of the application. Likewise, the *NetworkConfigurator* configures network parameters, such as *diffserv* code points of the component interconnections. Like other entities of RACE, *Configurators* are implemented as CCM components, so new configurators can be plugged into RACE by configuring RACE at design-time using PICML.

4.1.5. Challenge 5: computation of system adaptation decisions

Problem

In open DRE systems, resource utilization of applications might be significantly different than their estimated values

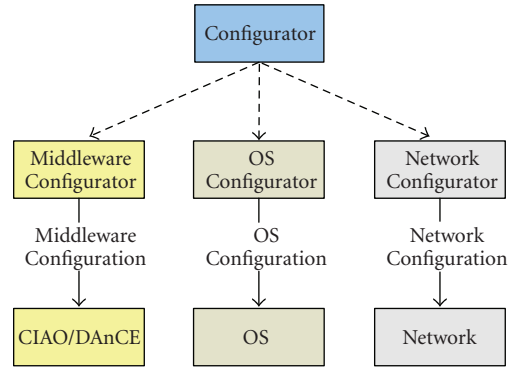


Figure 8: QoS parameter configuration with RACE.

and availability of system resources may be time variant. Moreover, for applications executing in these systems, the relation between input workload, resource utilization, and QoS cannot be characterized a priori. Therefore, in order to ensure that QoS requirements of applications are met, and utilization system resources are within the specified bounds, the system must be able to *adapt* to dynamic changes, such as variations in operational conditions, input workload, and/or resource availability.

Solution: control-theoretic adaptive resource management algorithms

RACE's *Controllers* implement various Control-theoretic adaptive resource management algorithms such as EUCON [9], DEUCON [10], HySUCON [11], and FMUF [12], thereby enabling open DRE systems to adapt to changing operational context and variations in resource availability and/or demand. Based on the control algorithm they implement, *Controllers* modify configurable system parameters, such as execution rates and mode of operation of the application, real-time configuration settings—operating system priorities of *component servers* that host the components—and network *diffserv* code points of the component interconnections. As shown in Figure 9, input to the controllers include current resource utilization and current QoS. Since *Controllers* are implemented as CCM components, RACE can be configured with new *Controllers* by using PICML.

4.1.6. Challenge 6: efficient execution of system adaptation decisions

Problem

Although control theoretic adaptive resource management algorithms compute system adaptation decisions, one of the challenges we faced in building RACE is the design and implementation of *effectors*—entities that modify system parameters in order to achieve the controller recommended system adaptation. The key challenge lies in designing and implementing the effector architecture that scales as well as the number of applications and nodes in the system increases.

Solution: hierarchical effector architecture

Effectors modify system parameters, including resources allocated to components, execution rates of applications, and OS/middleware/network QoS setting for components, to achieve the controller recommended adaptation. As shown in Figure 9, *Effectors* are designed hierarchically. The *Central Effector* first computes the values of various system parameters for all the nodes in the domain to achieve the *Controller*-recommended adaptation. The computed values of system parameters for each node are then propagated to *Effectors* located on each node, which then modify system parameters of its node accordingly.

The primary metric that is used to measure the performance of a monitoring effectors is *actuation delay*, which is defined as the time taken to execute controller-recommended adaptation throughout the system. To minimize the actuation delay and ensure that RACE scales as the number of applications and nodes in the system increases, the RACE's effectors are structured in a hierarchical fashion. We validate this claim in Section 5.

Since the elements of RACE are developed as CCM components, RACE itself can be configured using model-driven tools, such as PICML. Moreover, new- and/or domain-specific entities, such as *InputAdapters*, *Allocators, Controllers, Effectors, Configurators, QoS-Monitors*, and *Resource Monitors*, can be plugged directly into RACE without modifying RACE's existing architecture.

4.2. Addressing MMS mission requirements using RACE

Section 4.1 provides a detailed overview of various adaptive resource management challenges of open DRE systems and how RACE addresses these challenges. We now describe how RACE was applied to our MMS mission case study from Section 3 and show how it addressed key resource allocation, QoS-configuration, and adaptive resource management requirements that we identified in Section 3.

4.2.1. Addressing requirement 1: resource allocation to applications

RACE's *InputAdapter* parses the metadata that describes the application to obtain the resource requirement(s) of components that make up the application and populates the *E-2-E* IDL structure. The *Central Monitor* obtains system resource utilization/availability information for RACE's *Resource Monitors*, and using this information along with the *estimated* resource requirement of application components captured in the *E-2-E* structure, the *Allocators* map components onto nodes in the system domain based on run-time resource availability.

RACE's *InputAdapter, Central Monitor*, and *Allocators* coordinate with one another to allocate resources to applications executing in open DRE systems, thereby addressing the resource allocation requirement for open DRE systems identified in Section 3.2.1.

4.2.2. Addressing requirement 2: configuring platform-specific QoS parameters

RACE shields application developers and SA-POP from low-level platform-specific details and defines a higher-level QoS specification model. System developers and SA-POP specify only QoS characteristics of the application, such as QoS requirements and relative importance, and RACE's *Configurators* automatically configures platform-specific parameters appropriately.

For example, consider two science applications—one executing in fast survey mode and one executing in slow survey mode. For these applications, middleware parameters configured by the *Middleware Configurator* includes (1) CORBA end-to-end priority, which is configured based on execution mode (fast/slow survey) and application period/deadline; (2) CORBA priority propagation model (CLIENT_PROPAGATED/SERVER_DECLARED), which is configured based on the application structure and interconnection; and (3) threading model (single threaded/thread-pool/thread-pool with lanes), which is configured based on number of concurrent peer components connected to a component. The *Middleware Configurator* derives configuration for such low-level platform-specific parameters from application end-to-end structure and QoS requirements.

RACE's *Configurators* provides higher-level abstractions and shield system developers and SA-POP from low-level platform-specific details, thus addressing the requirements associated with configuring platform-specific QoS parameters identified in Section 3.2.2.

4.2.3. Addressing requirement 3: monitoring end-to-end QoS and ensuring QoS requirements are met

When resources are allocated to components at design-time by system designers using PICML, *that is,* mapping of application components to nodes in the domain are specified, these operations are performed based on estimated resource utilization of applications and estimated availability of system resources. Allocation algorithms supported by RACE's *Allocators* allocate resources to components based on current system resource utilization and component's estimated resource requirements. In open DRE systems, however, there is often no accurate a priori knowledge of input workload, the relationship between input workload and resource requirements of an application, and system resource availability.

To address this requirement, RACE's control architecture employs a feedback loop to manage system resource and application QoS and ensures (1) QoS requirements of applications are met at all times and (2) system stability by maintaining utilization of system resources below their specified utilization set-points. RACE's control architecture features a feedback loop that consists of three main components: *Monitors, Controllers*, and *Effectors*, as shown in Figure 9.

Monitors are associated with system resources and QoS of the applications and periodically update the *Controller* with the current resource utilization and QoS of applications currently running in the system. The *Controller* implements a particular control algorithm such as EUCON [9], DEUCON

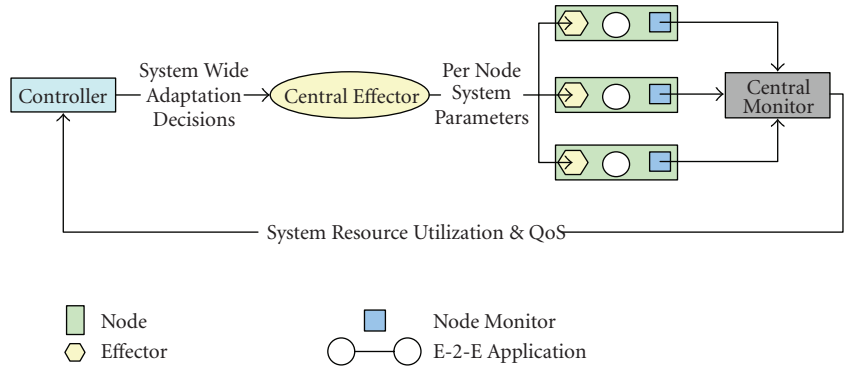


Figure 9: RACE's feedback control loop.

Table 2: Lines of source code for various system elements.

Entity	Total lines of source code
MMS DRE system	19,875
RACE	157,253
CIAO/DAnCE	511,378

[10], HySUCON [11], and FMUF [12], and computes the adaptations decisions for each (or a set of) application(s) to achieve the desired system resource utilization and QoS. *Effectors* modify system parameters, which include resource allocation to components, execution rates of applications, and OS/middleware/network QoS setting of components, to achieve the controller-recommended adaptation.

As shown in Figure 9, RACE's monitoring framework, *Controllers*, and *Effectors* coordinate with one another and the aforementioned entities of RACE to ensure (1) QoS requirements of applications are met and (2) utilization of system resources are maintained within the specified utilization set-point set-point(s), thereby addressing the requirements associated with run-time end-to-end QoS management identified in Section 3.2.3. We empirically validate this in Section 5.

5. EMPIRICAL RESULTS AND ANALYSIS

This section presents the design and results of experiments that evaluate the performance and scalability of RACE in our prototype of the NASA MMS mission system case study described in Section 3. These experiments validate our claims in Sections 4 and 4.2 that RACE is an scalable adaptive resource management framework and can perform effective end-to-end adaptation and yield a predictable and scalable DRE system under varying operating conditions and input workload.

5.1. Hardware and software test-bed

Our experiments were performed on the ISISLab test-bed at Vanderbilt University (www.dre.vanderbilt.edu/ISISlab). The hardware configuration consists of six nodes, five of which acted as spacecrafts and one acted as a ground station. The hardware configuration of all the nodes was a 2.8 GHz Intel Xeon dual processor, 1 GB physical memory, 1 GHz ethernet network interface, and 40 GB hard drive. The Redhat Fedora core release 4 OS with real-time preemption patches [47] was used for all the nodes.

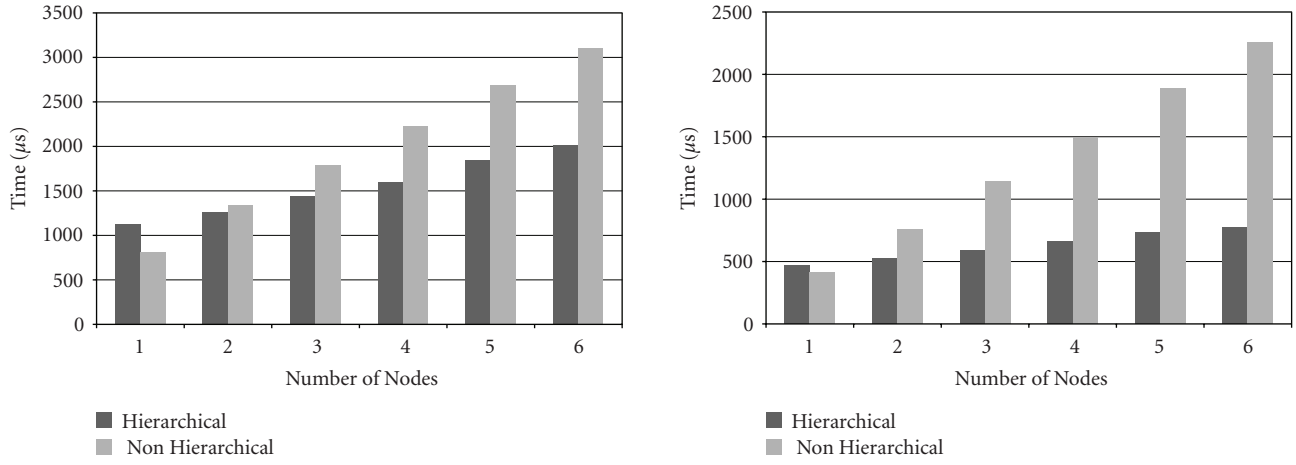
Our experiments also used CIAO/DAnCE 0.5.10, which is our open source QoS-enabled component middleware that implements the OMG lightweight CCM [48] and deployment and configuration [44] specifications. RACE and our DRE system case study are built upon CIAO/DAnCE.

5.2. MMS DRE system implementation

Science applications executing atop our MMS DRE system are composed of the following components:

(i) *plasma sensor component*, which manages and controls the plasma sensor on the spacecraft, collects metrics corresponding to the earth's plasma activity;
(ii) *camera sensor component*, which manages and controls the high-fidelity camera on the spacecraft and captures images of one or more star constellations;
(iii) *filter component*, which processes the data from the sensor components to remove any extraneous noise in the collected data/image;
(iv) *analysis component*, which processes the collected data to determine if the data is of interest or not. If the data is of interest, the data is compressed and transmitted to the ground station;
(v) *compression component*, which uses loss-less compression algorithms to compresses the collected data;
(vi) *communication component*, which transmits the compressed data to the ground station periodically;
(vii) *ground component*, which receives the compressed data from the spacecrafts and stores it for further processing.

All these components—except for the ground component—execute on the spacecrafts. Our experiments used component emulations that have the same resource utilization characteristics as the original components. Table 2 summarizes the number of lines of C++ code of various entities in our middleware, RACE, and our prototype implementation of



(a) Monitoring Delay versus Number of Nodes

(b) Actuation Delay versus Number of Nodes

Figure 10: Impact of increase in number of nodes on monitoring and actuation delay.

the MMS DRE system case study, which were measured using SLOCCount (www.dwheeler.com/sloccount/).

5.3. Evaluation of RACE's scalability

Sections 4.1.2 and 4.1.6 claimed that the hierarchical design of RACE's monitors and effectors enables RACE to scale as the number of applications and nodes in the system grows. We validated this claim by studying the impact of increasing number of nodes and applications on RACE's monitoring delay and actuation delay when RACE's monitors and effectors are configured hierarchically and nonhierarchically. As described in Sections 4.1.2 and 4.1.6, *monitoring delay* is defined as the time taken to obtain a snapshot of the entire system in terms of resource utilization and QoS and *actuation delay* is defined as the time taken to execute controller-recommended adaptation throughout the system.

To measure the monitoring and actuation delays, we instrumented RACE's *Central Monitor* and *Central Effector*, respectively, with high resolution timers—*ACE_High_Res_Timer* [15]. The timer in the *Central Monitor* measured the time duration from when requests were sent to individual *Node Monitors* to the time instant when replies from all *Node Monitors* were received and the data (resource utilization and application QoS) were assembled to obtain a snapshot of the entire system. Similarly, the timer in the *Central Effector* measured the time duration from when system adaptation decisions were received from the *Controller* to the time instant when acknowledgment indicating successful execution of node level adaption from individual *Effectors* (located on each node) were received.

5.3.1. Experiment 1: constant number of application and varying number of nodes

This experiment studied the impact of varying number of nodes in the system domain on RACE's monitoring and actuation delay. We present the results obtained from running the experiment with a constant of five applications, each composed of six components (plasma-sensor/camera-sensor, analysis, filter analysis, compression, communication, and ground), and a varying number of nodes.

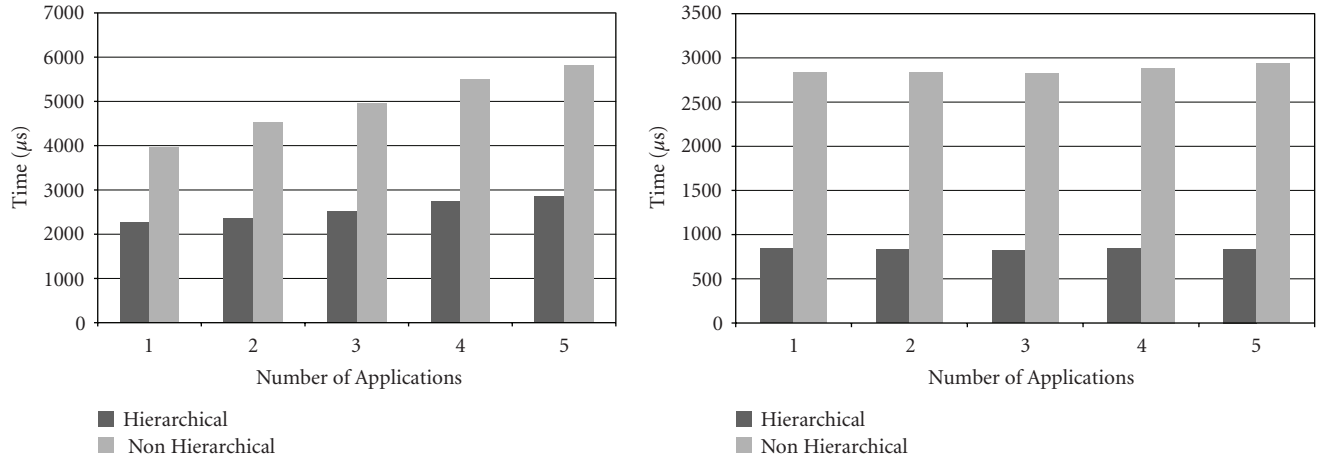
Experiment configuration

We varied the number of nodes in the system from one to six. A total of 30 application components were evenly distributed among the nodes in the system. The experiment was composed of two scenarios: (1) hierarchical and (2) nonhierarchical configuration of RACE's monitors and effectors. Each scenario was comprised of seven runs, and the number of nodes in the system during each run was. During each run, monitoring delay and actuation delay were collected over 50,000 iterations.

Analysis of results

Figures 10(a) and 10(b) compare the impact of increasing the number of nodes in the system on RACE's monitoring and actuation delay, respectively, under the two scenarios. Figures 10(a) and 10(b) show that monitoring and actuation delays are significantly lower in the hierarchical configuration of RACE's monitors and effectors compared to the nonhierarchical configuration. Moreover, as the number of nodes in the system increases, the increases in monitoring and actuation delays are significantly (i.e., 18% and 29%, resp.) lower in the hierarchical configuration compared to the nonhierarchical configuration. This result occurs because individual node monitors and effectors execute in parallel when monitors and effectors are structured hierarchically, thereby significantly reducing monitoring and actuation delay, respectively.

Figures 10(a) and 10(b) show the impact on monitoring and actuation delay when the monitors and effectors are structured hierarchically and the number of nodes in the system increases. Although individual monitors and effectors



(a) Monitoring Delay versus Number of Applications

(b) Actuation Delay versus Number of Applications

Figure 11: Impact of increase in number of application on monitoring and actuation delays.

execute in parallel, resource data aggregation and computation of per-node adaptation decisions are centralized by the *Central Monitor* and *Central Effector*, respectively. The results show that this configuration yields a marginal increase in the monitoring and actuation delay (i.e., 6% and 9%, resp.) as the number of nodes in the system increases.

Figures 10(a) and 10(b) show that when there is only one node in the system, the performance of the hierarchical configuration of RACE's monitors and effectors is worse than the nonhierarchical configuration. This result measures the overhead associated with the hierarchical configuration. As shown in Figures 10(a) and 10(b), however, as the number of nodes in the system increase, the benefit of the hierarchical configuration outweighs this overhead.

5.3.2. Experiment 2: constant number of nodes and varying number of applications

This experiment studied the impact of varying the number of applications on RACE's monitoring and actuation delay. We now present the results obtained from running the experiment with six nodes in the system and varying number of applications (from one to five), each composed of six components (plasma-sensor/camera-sensor, analysis, filter analysis, compression, communication, and ground).

Experiment configuration

We varied the number of applications in the system from one to five. Once again, the application components were evenly distributed among the six nodes in the system. This experiment was composed of two scenarios: (1) hierarchical and (2) nonhierarchical configuration of RACE's monitors and effectors. Each scenario was comprised of five runs, with the number of applications used in each run held constant. As we varied the number of applications from one to five, for each scenario we had a total of five runs. During each run, monitoring delay and actuation delay were collected over 50,000 iterations.

Analysis of results

Figures 11(a) and 11(b) compare the impact on increase in number of applications on RACE's monitoring and actuation delay, respectively, under the two scenarios. Figures 11(a) and 11(b) show that monitoring and actuation delays are significantly lower under the hierarchical configuration of RACE's monitors and effectors compared with the nonhierarchical configuration. These figures also show that under the hierarchical configuration, there is a marginal increase in the monitoring delay and negligible increase in the actuation delay as the number of applications in the system increase.

These results show that RACE scales as well as the number of nodes and applications in the system increase. The results also show that RACE's scalability is primarily due to the hierarchical design of RACE's monitors and effectors, there by validating our claims in Sections 4.1.2 and 4.1.6.

5.4. Evaluation of RACE's adaptive resource management capabilities

We now evaluate the adaptive resource management capabilities of RACE under two scenarios: (1) moderate workload, and (2) heavy workload. Applications executing on our prototype MMS mission DRE system were periodic, with deadline equal to their periods. In both the scenarios, we use the deadline miss ratio of applications as the metric to evaluate system performance. For every sampling period of RACE's *Controller*, deadline miss ratio for each application was computed as the ratio of number of times the application's end-to-end latency was greater than its deadline to the number of times the application was invoked. The end-to-end latency of an application was obtained from RACE's QoS *Monitors*.

5.4.1. Summary of evaluated scheduling algorithms

We studied the performance of the prototype MMS system under various configurations: (1) a baseline configuration without RACE and static priority assigned to application

Table 3: Application configuration under moderate workload.

Application	Component allocation		Ground station	Period (msec)	Mode
	Spacecraft				
	1	2			
1	Communication plasma-sensor	Analysis compression	Ground	1000	Fast survey
2	Analysis camera-sensor Filter	Communication compression	Ground	900	Slow survey
3	Plasma-sensor camera-sensor	Communication compression Filter	Ground	500	Slow survey

components based on rate monotonic scheduling (RMS) [46], (2) a configuration with RACE's maximum urgency first (MUF) *Configurator*, and (3) a configuration with RACE's MUF *Configurator* and flexible MUF (FMUF) [12] *Controller*. The goal of these experiments is not to compare the performance of various adaptive resource management algorithms, such as EUCON [9], DEUCON [10], HySUCON [11], or FMUF. Instead, the goal is to demonstrate how RACE can be used to implement these algorithms.

A disadvantage of RMS scheduling is that it cannot provide performance isolation for higher importance applications [49]. During system overload caused by dynamic increase in the workload, applications of higher importance with a low rate may miss deadlines. Likewise, applications with medium/lower importance but high rates may experience no missed deadlines.

In contrast, MUF provides performance isolation to applications of higher importance by dividing operating system and/or middleware priorities into two classes [49]. All components belonging to applications of higher importance are assigned to the high-priority class, while all components belonging to applications of medium/lower importance are assigned to the low-priority class. Components within a same priority class are assigned operating system and/or middleware priorities based on the RMS policy. Relative to RMS, however, MUF may cause priority inversion when a higher importance application has a lower rate than medium/lower importance applications. As a result, MUF may unnecessarily cause an application of medium/lower importance to miss its deadline, even when all tasks are schedulable under RMS.

To address limitations with MUF, RACE's FMUF *Controller* provides performance isolation for applications of higher importance while reducing the deadline misses of applications of medium/lower importance. While both RMS and MUF assign priorities statically at deployment time, the FMUF *Controller* adjusts the priorities of applications of medium/lower importance dynamically based on performance feedback. The FMUF *Controller* can reassign applications of medium/lower importance to the high-priority class when (1) all the applications currently in the high-priority class meet their deadlines while (2) some applications in the low-priority class miss their deadlines. Since the FMUF *Controller* moves applications of medium/lower importance back to the low-priority class when the high-priority class experiences deadline misses it can effectively deal with workload variations caused by application arrivals and changes in application execution times and invocation rates.

5.4.2. Experiment 1: moderate workload

Experiment configuration

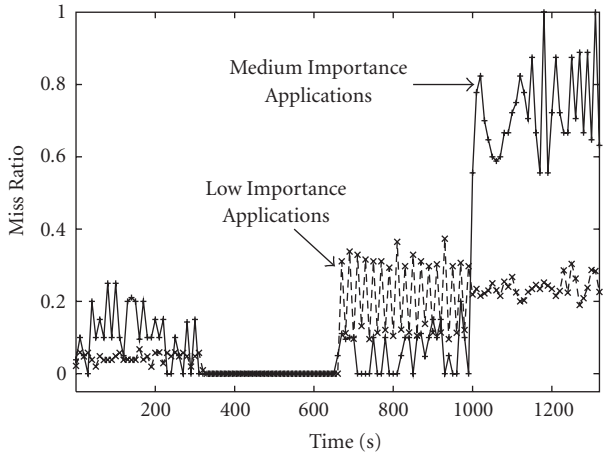
The goal of this experiment configuration was to evaluate RACE's system adaptation capabilities under a moderate workload. This scenario therefore employed two of the five emulated spacecrafts, one emulated ground station, and three periodic applications. One application was initialized to execute in fast survey mode and the remaining two were initialized to execute in slow survey mode. As described in Section 3.1, applications executing in fast survey mode have higher relative importance and resource consumption than applications executing in slow survey mode. Each application is subjected to an end-to-end deadline equal to its period. Table 3 summarizes application periods and the mapping of components/applications onto nodes.

The experiment was conducted over 1,400 seconds, and we emulated variation in operating condition, input workload, and a mode change by performing the following steps. At time $T = 0$ second, we deployed applications one and two. At time $T = 300$ seconds, the input workload for all the application was reduced by ten percent, and at time $T = 700$ seconds we deployed application three. At $T = 1000$ seconds, application three switched mode from slow survey to fast survey. To emulate this mode change, we increased the rate (i.e., reduced the period) of application three by twenty percent. Since each application was subjected to an end-to-end deadline equal to its period, to evaluate the performance of RACE, we monitored the *deadline miss ratio* of all applications that were deployed.

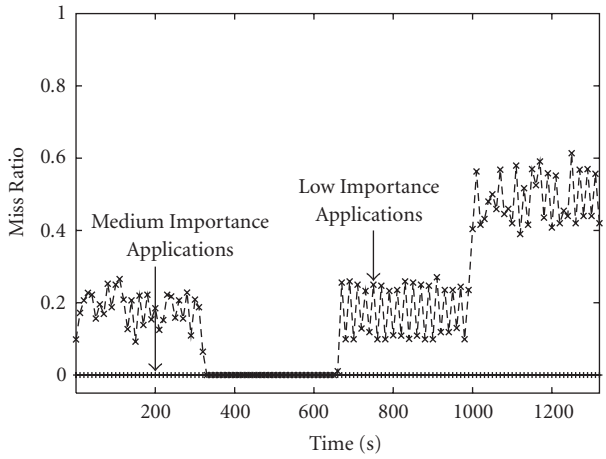
RACE's FMUF *Controller* was used for this experiment since the MMS mission applications described above do not support rate adaptation. RACE is a framework, however, so other adaptation strategies/algorithms, such as HySUCON [11], can be implemented and employed in a similar way. Below, we evaluate the use of FMUF for end-to-end adaptation. Since this paper focuses on RACE—and not the design or evaluation of individual control algorithms—we use FMUF as an example to demonstrate RACE's ability to support the integration of feedback control algorithms for end-to-end adaptation in DRE systems. RACE's FMUF controller was configured with the following parameters: sampling period = 10 seconds, $N = 5$, and *threshold* = 5%.

Analysis of results

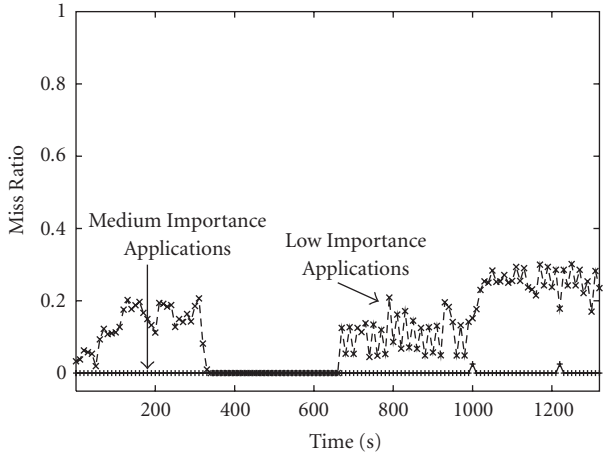
Figures 12(a), 12(b), and 12(c) show the deadline miss ratio of applications when the system was operated under



(a) Baseline (RMS)



(b) MUF Configurator



(c) MUF Configurator + FMUF Controller

Figure 12: Deadline miss ratio under moderate workload.

baseline configuration, with RACE's MUF *Configurator*, and with RACE's MUF *Configurator* along with FMUF *Controller*, respectively. These figures show that under all the three configurations, deadline miss ratio of applications (1) reduced at $T = 300$ seconds due to the decrease in the input work load, (2) increased at $T = 700$ seconds due to the introduction of new application, and (3) further increased at $T = 1,000$ seconds due to the mode change from slow survey mode to fast survey mode. These results demonstrate the impact of fluctuation in input workload and operating conditions on system performance.

Figure 12(a) shows that when the system was operated under the baseline configuration, deadline miss ratio of medium-importance applications (applications executing in fast survey mode) were higher than that of low-importance applications (applications executing in slow survey mode) due to reasons explained in Section 5.4.1. Figures 12(b) and 12(c) show that when RACE's MUF *Configurator* is used (both individually and along with FMUF *Controller*), deadline miss ratio of medium importance applications were nearly zero throughout the course of the experiment. Figures 12(a) and 12(b) demonstrate that the RACE improves QoS of our DRE system significantly by configuring platform-specific parameters appropriately.

As described in [12], the FMUF *Controller* responds to variations in input workload and operating conditions (indicated by deadline misses) by dynamically adjusting the priorities of the low-importance applications (i.e., moving low-importance applications into or out of the high-priority class). Figures 12(a) and 12(c) demonstrate the impact of the RACE's *Controller* on system performance.

5.4.3. Experiment 2: heavy workload

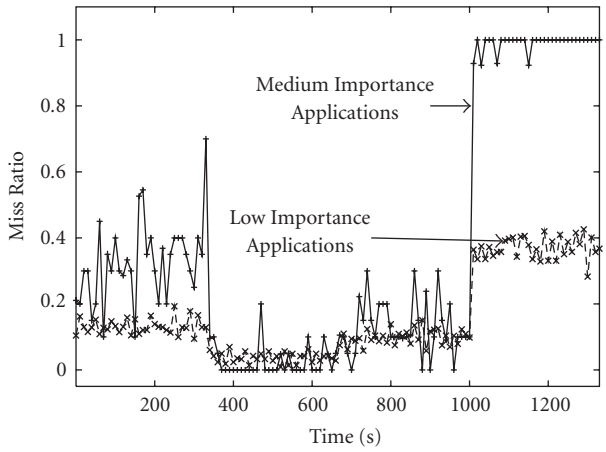
Experiment configuration

The goal of this experiment configuration was to evaluate RACE's system adaptation capabilities under a heavy workload. This scenario, therefore, employed all five emulated spacecrafts, one emulated ground station, and ten periodic applications. Four of these applications were initialized to execute in fast survey mode and the remaining six were initialized to execute in slow survey mode. Table 4 summarizes the application periods and the mapping of components/applications onto nodes.

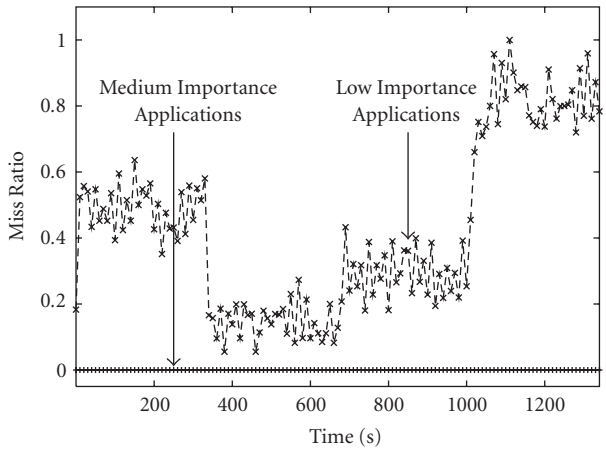
The experiment was conducted over 1,400 seconds, and we emulated the variation in operating condition, input workload, and a mode change by performing the following steps. At time $T = 0$ second, we deployed applications one through six. At time $T = 300$ seconds, the input workload for all the application was reduced by ten percent, and at time $T = 700$ seconds, we deployed applications seven through ten. At $T = 1,000$ seconds, applications two through five switched modes from slow survey to fast survey. To emulate this mode change, we increased the rate of applications two through five by twenty percent. RACE's FMUF controller was configured with the following parameters: sampling period = 10 seconds, $N = 5$, and *threshold* = 5%.

Analysis of results

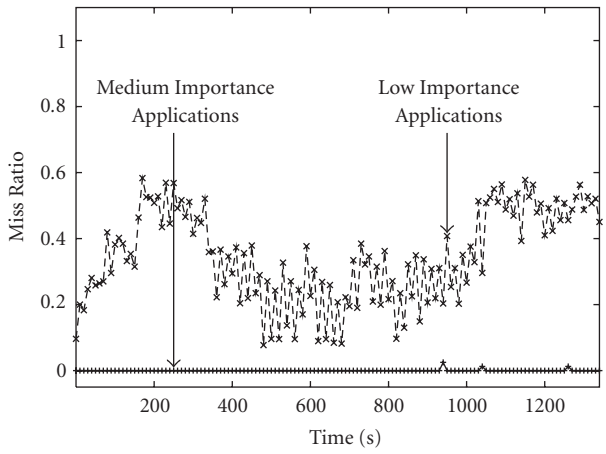
Figure 13(a) shows that when the system was operated under the baseline configuration, the deadline miss ratio of the medium importance applications were again higher than that of the low-importance applications. Figures 13(b) and



(a) Baseline (RMS)



(b) MUF Configurator



(c) MUF Configurator + FMUF Controller

Figure 13: Deadline Miss Ratio under Heavy Workload.

13(c) show that when RACE's MUF *Configurator* is used (both individually and along with FMUF *Controller*), deadline miss ratio of medium importance applications were nearly zero throughout the course of the experiment. Figures 13(a) and 13(b) demonstrate how RACE improves the QoS of our DRE system significantly by configuring platform-specific parameters appropriately. Figures 12(a) and 12(c) demonstrate that RACE improves system performance (deadline miss ratio) even under heavy workload.

These results show that RACE improves system performance by performing adaptive management of system resources there by validating our claim in Section 4.2.3.

5.5. Summary of experimental analysis

This section evaluated the performance and scalability of the RACE framework by studying the impact of increase in number of nodes and applications in the system on RACE's monitoring delay and actuation delay. We also studied the performance of our prototype MMS DRE system with and without RACE under varying operating condition and input workload. Our results show that RACE is a scalable adaptive resource management framework and performs effective end-to-end adaptation and yields a predictable and high-performance DRE system.

From analyzing the results in Section 5.3, we observe that RACE scales as well as the number of nodes and applications in the system increases. This scalability stems from RACE's the hierarchical design of monitors and effectors, which validates our claims in Sections 4.1.2 and 4.1.6. From analyzing the results presented in Section 5.4, we observe that RACE significantly improves the performance of our prototype MMS DRE system even under varying input workload and operating conditions, thereby meeting the requirements of building component-based DRE systems identified in Section 3.2. These benefits result from configuring platform-specific QoS parameters appropriately and performing effective end-to-end adaptation, which were performed by RACE's *Configurators* and *Controllers*, respectively.

6. CONCLUDING REMARKS

Open DRE systems require end-to-end QoS enforcement from their underlying operating platforms to operate correctly. These systems often run in environments where resource availability is subject to dynamic changes. To meet end-to-end QoS in these dynamic environments, open DRE systems can benefit from adaptive resource management frameworks that monitors system resources, performs efficient application workload management, and enables efficient resource provisioning for executing applications. Resource management algorithms based on control-theoretic techniques are emerging as a promising solution to handle the challenges of applications with stringent end-to-end QoS executing in open DRE systems. These algorithms enable adaptive resource management capabilities in open DRE systems and adapt gracefully to fluctuation in resource availability and application resource requirement at run-time.

This paper described the *resource allocation and control engine* (RACE), which is our adaptive resource management framework that provides end-to-end adaptation and resource management for open DRE systems built atop QoS-enabled component middleware. Open DRE systems built using RACE benefit from the advantages of component-based middleware, as well as QoS assurances provided

Table 4: Application configuration under heavy workload.

Application	Component allocation						Period (msec)	Mode
	Spacecraft					Ground station		
	1	2	3	4	5			
1	Communication		Analysis Plasma-sensor	Filter	Compression	Ground	1000	Fast Survey
2	Camera-sensor Compression	Filter Analysis			Communication	Ground	900	Slow Survey
3	Camera-sensor	Plasma-sensor	Communication Compression	Analysis	Filter	Ground	500	Slow Survey
4		Communication	Filter Analysis	Plasma-sensor	Compression	Ground	800	Slow Survey
5	Communication Filter		Camera-sensor	Analysis	Compression	Ground	1200	Slow Survey
6	Analysis	Filter	Communication	Compression	Plasma-sensor	Ground	700	Slow Survey
7	Plasma-sensor	Plasma-sensor	Communication Compression	Analysis	Filter	Ground	600	Fast Survey
8		Communication Filter	Analysis	Plasma-sensor	Compression	Ground	700	Slow Survey
9	Communication Filter		Camera-sensor Plasma-sensor	Analysis	Compression	Ground	400	Fast Survey
10	Compression Filter		Communication Analysis		Plasma-sensor	Ground	700	Fast Survey

by adaptive resource management algorithms. We demonstrated how RACE helped resolve key resource and QoS management challenges associated with a prototype of the NASA MMS mission system. We also analyzed results from performance in the context of our MMS mission system prototype.

Since the elements of the RACE framework are CCM components, RACE itself can be configured using model-driven tools, such as PICML [38]. Moreover, new *InputAdapters, Allocators, Configurators*, and *Controllers* can be plugged into RACE using PICML without modifying its architecture. RACE can also be used to deploy, allocate resources to, and manage performance of, applications that are composed at design-time and run-time.

The lessons learned in building RACE and applying to our MMS mission system prototype thus far include the following.

(i) *Challenges involved in developing open DRE systems.* Achieving end-to-end QoS in open DRE systems requires adaptive resource management of system resources, as well as integration of a range of real-time capabilities. QoS-enabled middleware, such as CIAO/DAnCE, along with the support of DSMLs and tools, such as PICML, provide an integrated platform for building such systems and are emerging as an operating platform for these systems. Although CIAO/DAnCE and PICML alleviate many challenges in building DRE systems, they do not address the adaptive resource management challenges and requirements of open DRE systems. Adaptive resource management solutions are therefore needed to ensure QoS requirements of applications executing atop these systems are met.

(ii) *Decoupling middleware and resource management algorithms.* Implementing adaptive resource management algorithms within the middleware tightly couples the resource management algorithms within particular middleware platforms. This coupling makes it hard to enhance the algorithms without redeveloping significant portions of the middleware. Adaptive resource management frameworks, such as RACE, alleviate the tight coupling between resource management algorithms and middleware platforms and improve flexibility.

(iii) *Design of a framework determines its performance and applicability.* The design of key modules and entities of the resource management framework determines the scalability, and therefore the applicability, of the framework. To apply a framework like RACE to a wide range of open DRE system, it must scale as the number of nodes and application in the system grows. Our empirical studies on the scalability of RACE showed that structuring and designing key modules of RACE (e.g., monitors and effectors) in a hierarchical fashion not only significantly improves the performance of RACE, but also improves its scalability.

(iv) *Need for configuring/customizing the adaptive resource management framework with domain specific monitors.* Utilization of system resources, such as CPU, memory, and network bandwidth, and system performance, such as latency and throughput, can be measured in a generic fashion across various system domains. In open DRE systems, however, the need to measure utilization of domain-specific resources, such as battery utilization, and application-specific QoS metrics, such as the fidelity of the collected plasma data,

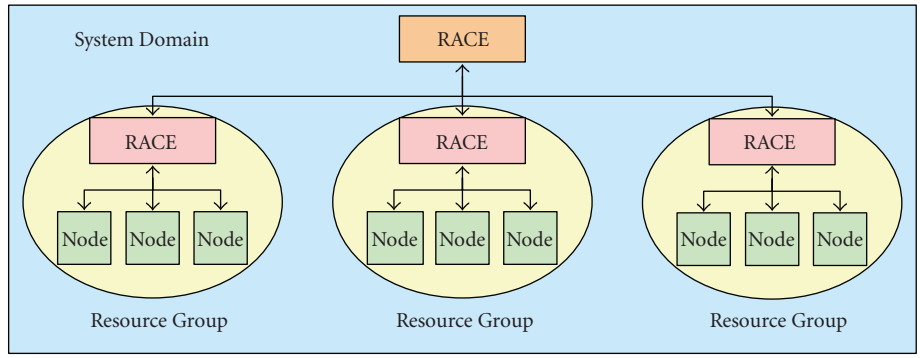


Figure 14: Hierarchical Composition of RACE.

might occur. Domain-specific customization and configuration of an adaptive resource management framework, such as RACE, should therefore be possible. RACE supports domain-specific customization of its *Monitors*. In future work, we will empirically evaluate the ease of integration of these domain-specific resource entities.

(v) *Need for selecting an appropriate control algorithm to manage system performance.* The control algorithm that a *Controller* implements relies on certain system parameters that can be fine-tuned/modified at run-time to achieve effective system adaptation. For example, FMUF relies on fine-tuning operating system priorities of processes hosting application components to achieve desired system adaptation; EUCON relies on fine-tuning execution rates of end-to-end applications to achieve the same. The applicability of a control algorithm to a specific domain/scenario is therefore determined by the availability of these run-time configurable system parameters. Moreover, the responsiveness of a control algorithm and the *Controller* in restoring the system performance metrics to their desired values determines the applicability of a *Controller* to a specific domain/scenario. During system design-time, a *Controller* should be selected that is appropriate for the system domain/scenario.

(vi) *Need for distributed/decentralized adaptive resource management.* It is easier to design, analyze, and implement *centralized* adaptive resource management algorithms that manage an entire system than it is to design, analyze, and implement *decentralized* adaptive resource management algorithms. As a the size of a system grows, however, centralized algorithms can become bottlenecks since the computation time of these algorithms can scale exponentially as the number of end-to-end applications increases. One way to alleviate these bottlenecks is to partition system resources into *resource groups* and employ hierarchical adaptive resource management, as shown in Figure 14. In our future work, we plan to enhance RACE so that a *local* instance of the framework can manage resource allocation, QoS configuration, and run-time adaption within a resource group, whereas a *global* instance can be used to manage the resources and performance of the entire system.

RACE, CIAO, DAnCE, and PICML are available in open source form for download at http://deuce.doc.wustl.edu/.

REFERENCES

[1] S. A. Brandt, S. Banachowski, C. Lin, and T. Bissom, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS '03)*, pp. 396–407, Cancun, Mexico, December 2003.

[2] P. Martí, C. Lin, S. A. Brandt, M. Velasco, and J. M. Fuertes, "Optimal state feedback based resource allocation for resource-constrained control tasks," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, pp. 161–172, Lisbon, Portugal, December 2004.

[3] Object Management Group, "Real-time CORBA Specification," OMG Document formal/05-01-04 ed, August 2002.

[4] G. Bollella, J. Gosling, B. Brosgol, et al., *The Real-Time Specification for Java*, Addison-Wesley, Reading, Mass, USA, 2000.

[5] D C. Sharp and W. C. Roll, "Model-based integration of reusable component-based avionics systems," in *Proceedings of the IEEE Real-time and Embedded Technology and Applications Symposium*, Cancun, Mexico, December 2003.

[6] D. C. Sharp and W. C. Roll, "Model-Based integration of reusable component-based avionics system," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, Washington, DC, USA, May 2003.

[7] D. Suri, A. Howell, N. Shankaran, et al., "Onboard processing using the adaptive network architecture," in *Proceedings of the 6th Annual NASA Earth Science Technology Conference*, College Park, Md, USA, June 2006.

[8] P. K. Sharma, J. P. Loyall, G. T. Heineman, R. E. Schantz, R. Shapiro, and G. Duzan, "Component-based dynamic QoS adaptations in distributed real-time and embedded systems," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA '04)*, vol. 3291, pp. 1208–1224, Agia Napa, Cyprus, October 2004.

[9] C. Lu, X. Wang, and X. D. Koutsoukos, "Feedback utilization control in distributed real-time systems with end-to-end tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 550–561, 2005.

[10] X. Wang, D. Jia, C. Lu, and X. D. Koutsoukos, "Decentralized utilization control in distributed real-time systems," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, pp. 133–142, Miami, Fla, USA, December 2005.

[11] X. D. Koutsoukos, R. Tekumalla, B. Natarajan, and C. Lu, "Hybrid supervisory control of real-time systems," in *Proceedings*

of the 11th IEEE Real-time and Embedded Technology and Applications Symposium, San Francisco, Calif, USA, March 2005.

[12] Y. Chen and C. Lu, "Flexible maximum urgency first scheduling for distributed real-time systems," *Tech. Rep. WUCSE-2006-55*, Washington University, St. Louis, Mo, USA, October 2006.

[13] N. Shankaran, X. D. Koutsoukos, D. C. Schmidt, Y. Xue, and C. Lu, "Hierarchical control of multiple resources in distributed real-time and embedded systems," in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS '06)*, pp. 151–160, Dresden, Germany, July 2006.

[14] X. Wang, C. Lu, and X. D. Koutsoukos, "Enhancing the robustness of distributed real-time middleware via end-to-end utilization control," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, pp. 189–199, Miami, Fla, USA, December 2005.

[15] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, Reading, Mass, USA, 2002.

[16] Object Management Group, "Common Object Request Broker Architecture Version 1.3," OMG Document formal/2004-03-12 ed, March 2004.

[17] E. Pitt and K. McNiff, *Java RMI : The Remote Method Invocation Guide*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[18] B. Ravindran, L. Welch, and B. Shirazi, "Resource management middleware for dynamic, dependable real-time systems," *Real-Time Systems*, vol. 20, no. 2, pp. 183–196, 2001.

[19] L. R. Welch, B. A. Shirazi, B. Ravindran, and C. Bruggeman, "DeSiDeRaTa: QoS management technology for dynamic, scalable, dependable real-time systems," in *Proceedings of the IFACs 15th Workshop on Distributed Computer Control Systems (DCCS '98)*, IFAC, Como, Italy, September 1998.

[20] D. Fleeman, M. Gillen, A. Lenharth, et al., "Quality-based adaptive resource management architecture (QARMA): a CORBA resource management service," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, vol. 18, pp. 1623–1630, Santa Fe, NM, USA, April 2004.

[21] C. D. Gill, *Flexible scheduling in middleware for distributed rate-based real-time applications, Ph.D. dissertation*, St. Louis, Mo, USA, 2002.

[22] K. Bryan, L. C. DiPippo, V. Fay-Wolfe, et al., "Integrated CORBA scheduling and resource management for distributed real-time embedded systems," in *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS '05)*, pp. 375–384, San Francisco, Calif, USA, March 2005.

[23] V. F. Wolfe, L. C. DiPippo, R. Bethmagalkar, et al., "RapidSched: static scheduling and analysis for real-time CORBA," in *Proceedings of the 4th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '99)*, pp. 34–39, Santa Barbara, Calif, USA, January 1999.

[24] J. W. Liu, J. Redondo, Z. Deng, et al., "PERTS: a prototyping environment for real-time systems," *Tech. Rep. UIUCDCS-R-93-1802*, University of Illinois at Urbana-Champaign, Champaign, Ill, USA, 1993.

[25] Object Management Group, "CORBA Components," OMG Document formal/2002-06-65 ed, June 2002.

[26] Sun Microsystems, "Enterprise JavaBeans specification," August 2001, http://java.sun.com/products/ejb/docs.html.

[27] A. Thomas, "Enterprise JavaBeans technology," prepared for Sun Microsystems, December 1998, http://java.sun.com/products/ejb/white_paper.html.

[28] J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 55–73, 1997.

[29] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging quality of service control behaviors for reuse," in *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC '02)*, pp. 375–385, Washington, DC, USA, April-May 2002.

[30] P. Manghwani, J. Loyall, P. Sharma, M. Gillen, and J. Ye, "End-to-end quality of service management for distributed real-time embedded applications," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, p. 138a, Denver, Colo, USA, April 2005.

[31] J. S. Kinnebrew, N. Shankaran, G. Biswas, and D. C. Schmidt, "A decision-theoretic planner with dynamic component reconfiguration for distributed real-time applications," in *Proceedings of the 21th National Conference on Artificial Intelligence*, Boston, Mass, USA, July 2006.

[32] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath, "Cadena: an integrated development, analysis, and verification environment for component-based systems," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pp. 160–172, Portland, Ore, USA, May 2003.

[33] J. A. Stankovic, R. Zhu, R. Poornalingam, et al., "VEST: an aspect-based composition tool for real-time systems," in *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '03)*, pp. 58–69, Toronto, Canada, May 2003.

[34] A. Lédeczi, A. Bakay, M. Maróti, et al., "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[35] S. Kodase, S. Wang, Z. Gu, and K. G. Shin, "Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers," in *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS '03)*, pp. 181–188, San Francisco, Calif, USA, May 2003.

[36] P. López, J. L. Medina, and J. M. Drake, "Real-time modelling of distributed component-based applications," in *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications (Euromicro-SEAA '06)*, pp. 92–99, Dubrovnik, Croatia, August 2006.

[37] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and J. M. D. Moyano, "MAST: modeling and analysis suite for real time applications," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*, pp. 125–134, Delft, The Netherlands, June 2001.

[38] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A platform-independent component modeling language for distributed real-time and embedded systems," in *Proceedings of the11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '05)*, pp. 190–199, San Francisco, Calif, USA, March 2005.

[39] S. Bagchi, G. Biswas, and K. Kawamura, "Task planning under uncertainty using a spreading activation network," *IEEE Transactions on Systems, Man, and Cybernetics Part A*, vol. 30, no. 6, pp. 639–650, 2000.

[40] S. Curtis, "The magnetospheric multiscale mission—resolving fundamental processes in space plasmas," *Tech. Rep. NASA/TM-2000-209883*, NASA STI/Recon, Greenbelt, Md, USA, December 1999.

[41] N. Wang, D. C. Schmidt, A. Gokhale, et al., "QoS-enabled middleware," in *Middleware for Communications*,

Q. Mahmoud, Ed., pp. 131–162, John Wiley & Sons, New York, NY, USA, 2004.

[42] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale, "DAnCE: a QoS-enabled component deployment and configuration engine," in *Proceedings of the 3rd International Working Conference on Component Deployment (CD '05)*, vol. 3798 of *Lecture Notes in Computer Science*, pp. 67–82, Grenoble, France, November 2005.

[43] D. de Niz and R. Rajkumar, "Partitioning bin-packing algorithms for distributed real-time systems," *International Journal of Embedded Systems*, vol. 2, no. 3-4, pp. 196–208, 2006.

[44] Object Management Group, "Deployment and Configuration Adopted Submission," OMG Document mars/03-05-08 ed, July 2003.

[45] B. Buck and J. K. Hollingsworth, "API for runtime code patching," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[46] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proceedings of the Real-Time Systems Symposium (RTSS '89)*, pp. 166–171, Santa Monica, Calif, USA, December 1989.

[47] I. Molnar, "Linux with real-time pre-emption patches," September 2006, http://www.kernel.org/pub/linux/kernel/projects/rt/.

[48] Object Management Group, "Light Weight CORBA Component Model Revised Submission," OMG Document realtime/03-05-05 ed, May 2003.

[49] D. B. Stewart and P. K. Khosla, "Real-time scheduling of sensor-based control systems," in *Real-time Programming*, W. Halang and K. Ramamritham, Eds., pp. 139–144, Pergamon Press, Tarrytown, NY, USA, 1992.