Design and Performance Evaluation of Resource-Management Framework for End-to-End Adaptation of Distributed Real-time Embedded Systems

Nishanth Shankaran[†], Douglas C. Schmidt[†], Xenofon D. Koutsoukos[†],
Yingming Chen[‡], and Chenyang Lu[‡]

[†]Dept. of EECS, Vanderbilt University, Nashville, TN

[‡]Dept. of Computer Science and Engineering, Washington University, St. Louis

Abstract

Standards-based quality of service (QoS)-enabled component middleware is increasingly being used as a platform for developing distributed real-time embedded (DRE) systems. Although QoS-enabled component middleware offers many desirable features, until recently it lacked the ability to monitor utilization of system resources, efficiently allocate resources to application components, and ensure application QoS requirements are met. Moreover, it has also lacked the ability to handle fluctuations in availability of resource resources and application workload.

This paper presents two contributions to research on adaptive resource management for component-based 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 DRE system: NASA's Magnetospheric Multi-scale Mission system. Our empirical results demonstrate that when adaptive resource management algorithms for DRE systems are implemented using RACE, they yield in a predictable and high performance system, even in the face of changing operational conditions, workloads, and resource availability.

1 Introduction

Distributed real-time and embedded (DRE) systems form the core of many mission-critical domains, such as shipboard computing environments [1], avionics mission computing [2], multi-satellite missions [3], and intelligence, surveillance and reconnaissance missions [4]. *Quality of service (QoS)-enabled distributed object computing (DOC) middleware* based on standards like Real-time Common Object Request Broker Architecture (RT-CORBA) [5] and the Real-Time Specification for Java (RTSJ) [6] have been used to develop such DRE systems. More recently, *QoS-enabled component middleware*, such as the Lightweight CORBA Component Model (CCM) [7] and PRiSm [8], have been used as the middleware for DRE

systems [2].

1.1 Evolution of Middleware Technology

This section summarizes the evolution of various middleware technologies primarily focusing on their contributions and limitations.

1. Distributed Object Computing (DOC) Middleware: Commercial-off-the-shelf (COTS) middleware technologies for DOC based on standards such as The Object Management Group (OMG)'s CORBA [9] and Sun's Java RMI [10], 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 APIs and components automate and extend native OS capabilities. Conventional DOC middleware technologies, however, have the following limitations:

- **They address only functional aspects of application development:** DOC middleware address *functional* aspects of application development such as how to define and integrate object interfaces and implementations. They do not address QoS aspects 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.

- **Lacks support for system design and development:** DOC middleware support only the design and development of individual application objects. They lack generic standards for (1) distributing object implementations within the system, (2) installing, initializing, and configuring objects, and (3) interconnection between independent objects, all of which are crucial in system development. Therefore, when large-scale distributed systems are built using DOC middleware technologies, system design and development is tedious, error prone, hard to maintain and/or evolve, and results in a brittle system.

2. QoS-enabled DOC Middleware: QoS limitations of conventional DOC middleware have been addressed by new middleware standards such as RT-CORBA [5] and RTSJ [6]. Middleware based on these technologies support explicit configuration of QoS middleware aspects such as priority and threading models. However, it lacks the higher level abstraction that separates real-time policy configuration from application functionality. Moreover, these enhancements do not address the limitation of conventional DOC middleware in the context of system design and development as described above.

3. Conventional Component Middleware: Component middleware technologies, such as the CORBA Component Model (CCM) [11] and Enterprise Java Beans [12, 13] provide capabilities that addresses the limitation of conventional DOC middleware in the context of system design and development. Examples of capabilities offered 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 design and development of large scale distributed systems, they do not address the 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.

4. QoS-enabled Component Middleware: To address the limitations with various middleware technologies listed above, QoS-enabled component middleware such as Component Integrated ACE ORB (CIAO) [14] have evolved that combines the capabilities of conventional component middleware and real-time DOC middleware. CIAO is our open source implementation of the OMG's Lightweight CORBA Model (LCCM) specification [7] and is built atop The ACE ORB (TAO) [15]. QoS-enabled component middlewares offer explicit configuration of QoS middleware parameters as well as provide capabilities that aid in design and development of large scale distributed systems. QoS-enabled component middleware capabilities enhance the design, development, evolution, and maintenance of DRE systems [16].

1.2 Limitations of Existing Middleware Technologies and Solution Approach

These middleware technologies are suitable for 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 runtime. However, these technologies are insufficient for DRE systems that execute in *open* environments where system operational conditions, input workload, and resource availability cannot be characterized accurately *a priori*. Therefore, there is an increasing need to introduce resource management mechanisms within the middleware that can *adapt* to dynamic changes in resource availability and requirements. As a first step towards this, to provide end-to-end QoS assurances for applications executing in open DRE systems, in our earlier work we developed adaptive resource management algorithms, such as EUCON [17], DEUCON [18], HySUCON [19], and FMUF [20], based on control theoretic techniques. When then developed FC-ORB [21], which is a QoS enabled *adaptive* DOC middleware that implements the EUCON algorithm and can handle fluctuation in application workload and system resource availability.

Since the design and implementation of FC-ORB is based on the EUCON adaptive resource management algorithm, enhancing FC-ORB or any other DOC middleware to use other algorithms such as DEUCON, HySUCON, FMUF, and others that may be developed in the future would involve re-designing and/or re-implementing significant portions of the middleware. To address this issue, we have developed a component-based adaptive resource management framework – the *Resource Allocation and Control Engine* (RACE). Application built using RACE benefit from the aforementioned advantages of component based middleware as well as QoS and/or performance assurances provided by adaptive resource management algorithms. RACE therefore complements the theoretical work on resource allocation and control algorithms that provide a model and theoretical analysis of system performance.

RACE is built atop our CIAO QoS-enabled component middleware. 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.

RACE supports multiple applications running in various DRE system environments and allows applications with diverse QoS requirements to share resources simultaneously. As various entities of RACE themselves are designed and implemented as CCM components, RACE's allocators and controllers can be configured with diverse resource allocation and control algorithms using tools based on domain-specific modeling languages (DSML) such as the *Platform-Independent Component Modeling Language* (PICML) [22].

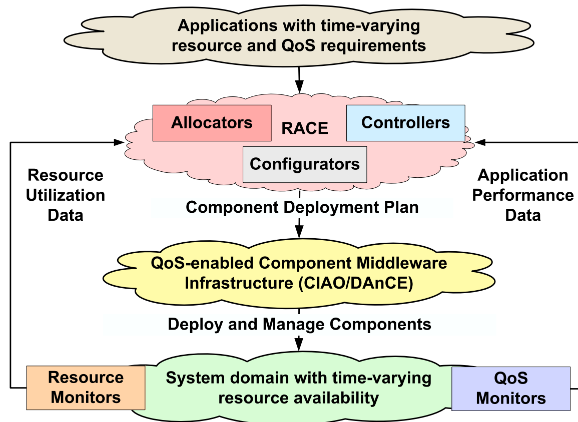


Figure 1: Resource Allocation and Control Engine (RACE) for DRE Systems

This paper provides two contributions to research on adaptive resource management for component-based DRE systems. First, it describes the design and implementation of the RACE framework. Second, we qualitatively and quantitatively evaluate the effectiveness of RACE in resolving key adaptive resource management challenges of a representative DRE system. 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; Section 5 describes the architecture of RACE and shows how it aids in the development of the DRE system described in Section 3; Section 6 empirically evaluates the performance of the DRE system when various control algorithms are used in conjunction with RACE and also presents an empirical measure of the overhead associated with the RACE framework; and Section 7 presents concluding remarks.

2 Related Work

This section compares our work on RACE with related research on building large-scale DRE systems. As shown below, we classify this research along two orthogonal dimensions: (1) QoS-enabled DOC middleware vs. QoS-enabled component middleware and (2) design-time vs. run-time QoS configuration, optimization, analysis, and evaluation of constraints, such as timing, memory, and CPU.

2.1 QoS-enabled DOC Middleware

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.

Run-time. Early work on resource management middleware for shipboard DRE systems presented in [25, 26] motivated the need for adaptive resource management middleware. This work was further extended by QARMA [27], which provides resource management as a *service* for existing QoS-enabled DOC middleware, such as RT-CORBA. Kokyu [28] 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 [29] enhances QoS-enabled DOC middleware by combining Kokyu and QARMA.

Our work on RACE extends this earlier work on QoS-enabled DOC middleware by providing an adaptive resource management framework for DRE systems built atop QoS-enabled component middleware. DRE systems built using RACE benefit from the additional capabilities offered by QoS-enabled component middleware compared to QoS-enabled DOC middleware, as described in Section 1. Moreover, the elements of RACE are designed as CCM components, so RACE itself can be configured using DSML tools, such as PICML.

2.2 QoS-enabled Component Middleware

Design-time. Cadena [30] is an integrated environment for developing and verifying component-based DRE systems by applying static analysis, model-checking, and lightweight formal methods. Like PICML, Cadena also provides a component assembly framework for visualizing and developing components and their connections. VEST [31] is a design assistant tool based on the *Generic Modeling Environment* [32] 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 [33] 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.

These tools are similar to PICML and 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 DRE systems that execute in open environments.

Run-time. QoS provisioning frameworks, such as QuO [34] and Qoskets [35, 4, 36] 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 resource 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 [37]), 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 to work atop any component middleware, not just CCM.

Compared with related work, RACE provides adaptive resource and QoS management capabilities in a more transparent and non-intrusive way. 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 [21] and HiDRA [38], 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 design-time DSML tools, which can be used design the application as well as configure RACE itself, QoS-enabled component middleware run-time platforms, and research on adaptive resource management algorithm. RACE can be used to deploy and manage component-based applications that are composed at design-time via PICML, as well as at run-time by *intelligent mission planners* [39] such as SA-POP [37].

3 Motivating Application Scenario

We use the NASA's upcoming Magnetospheric Multi-scale (MMS) mission (`stp.gsfc.nasa.gov/missions/mms/mms.htm`) as a motivating DRE system example to evaluate the effectiveness and performance of RACE. First, we present an overview of the MMS mission system, followed by the resource and QoS management challenges involved in developing the MMS mission using QoS-enabled component middleware.

3.1 DRE System Case Study

NASA's MMS mission is a representative DRE system consisting of several interacting subsystems (both in-flight and stationary) with a variety of complex QoS requirements. The MMS mission consists of four identical instrumented spacecrafts that maintain a specific formation while orbiting over a region of scientific interest. The primary function of the spacecraft(s) is to collect data while in orbit and send it to a ground station for further processing when appropriate.

Applications in the MMS mission have the following QoS characteristics and requirements: (1) they operate in multiple modes, (2) relative importance between applications varies at runtime according to the mode in which they are operating, (3) end-to-end response time shoud be within the specified end-to-end deadline, and (4) resource utilization is dependent on the data collected. Applications in the MMS mission execute in three modes of operation: *slow*, *fast*, and *burst* survey modes. Applications executing in fast and burst mode are considered more important than applications executing in slow mode, *i.e.*, applications executing in fast and burst mode belong to the *important* class, whereas application executing in slow mode belong to the *best-effort* class. Resource utilization by these applications are also dependent on their mode of operation, *i.e.*, resource utilization of an application in slow, fast, and burst modes are minimal, medium, and maximum respectively.

For example, for an application that monitors the plasma activity, *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.

Each spacecraft consists of an on-board intelligent mission planner such as the *spreading activation partial order planner* (SA-POP) [37] which decomposes overall mission goal(s) into navigation, control, data gathering-processing 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 applications, SA-POP incrementally generates new applications in response to changing mission goals and/or degraded performance reported by the mission and system monitors.

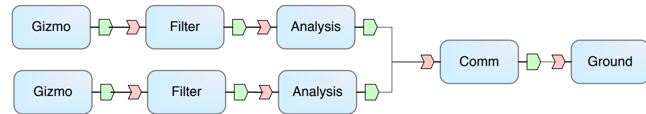


Figure 2: Example MMS Data Gather-Process Application

Figure 2 shows the instances of—and connections between—software components of an example data gather-process application executing within a single spacecraft. As shown in Figure 2, data gathering-processing applications typically consists of the following components: *gizmo*, *filter*, *analysis*, *comm*, and *ground* components. Each *gizmo* component collects data from the sensors, which have varying data rate, data size, and compression requirements. The data collected from the different sensors have varying importance, depending on the mode and on the mission. The collected data is passed through *filter* components, which remove noise from the data. The *filter* components pass the data onto *analysis* components, which compute a quality value indicating the likelihood of a transient plasma event. This quality value is then communicated to the other spacecraft and used to determine entry into burst mode while in fast mode. Finally, the analyzed data from each *analysis* component is passed to a *comm* (communication) component, which transmits the data to the *ground* component at an appropriate time.

3.2 Challenges of Developing the MMS Mission using QoS-enabled Component Middleware

As discussed in Section 1, the use of QoS-enabled component middleware to develop DRE systems, such as the NASA MMS mission, significantly improves the design, development, evolution, and maintenance of these large-scale systems. In the absence of an adaptive resource management framework like RACE, however, several key challenges remain unresolved when using component middleware. Below we present the key resource and QoS management challenges associated with the MMS mission DRE system.

Challenge 1: Resource allocation to applications. Applications generated by SA-POP are *resource sensitive*, *i.e.*, end-to-end response time is increased significantly if components of an application do 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 by, and QoS of, applications. Therefore, utilization of system resources and QoS of applications may vary significantly from their estimated values. Due to the operating condition for open DRE systems, system resource availability, such as available network bandwidth, may also be time variant. A resource management framework like RACE should therefore (1) monitor the current utilization of system resources, (2) allocate resources in a timely fashion to application

components such that their resource requirements are met using resource allocation algorithms such as PBFD [40], 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.

Challenge 2: Configuring platform-specific QoS parameters. The QoS of 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 like RACE should therefore provide abstractions that shield developers and/or SA-POP from low-level platform-specific details and define higher-level QoS specification models. The system developers and/or intelligent mission planners would specify QoS characteristics of the application such as QoS requirements and relative importance, and the framework must configure platform-specific parameters accordingly.

Challenge 3: Monitoring end-to-end QoS and ensuring QoS requirements are met. To meet the end-to-end QoS requirements of applications, an adaptive resource management framework like RACE must provide monitors that track 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 delay*) can be tracked by the framework transparently to the application. The framework should also provide hooks into which application specific QoS monitors can be configured. The framework should enable the system to *adapt* to dynamic changes, such as variations in operational conditions, input workload, and/or resource availability, and thereby ensure that QoS requirements of applications are not violated.

4 Overview of CCM Middleware

RACE is built atop of the QoS-enabled component middleware CIAO and DAnCE, which are open-source implementations of the OMG Lightweight CCM [7], Deployment and Configuration (D&C) [41], and RT-CORBA [5] specifications, which are outlined below. CIAO abstracts key Real-time CORBA QoS concerns into elements that can be configured declaratively via Lightweight CCM metadata. DAnCE, which is a standard component middleware deployment and configuration mechanisms, uses this metadata to perform deployment, initialization, interconnection, and lifecycle management of applications components.

RT-CORBA adds QoS control to regular CORBA to improve application *predictability*, such as bounding priority inversions. RT-CORBA provides policies and mechanisms for configuring middleware features such as thread pools, priority models, protocol policies, and explicit binding. These capabilities address some, but by no means all, important DRE system development challenges. To address this issue, the OMG introduced the Lightweight CCM specification, which is built atop

the RT-CORBA specification and standardizes the development, configuration, and deployment of component-based applications. CCM is built atop CORBA object model, and therefore, system implementors are not tied to any particular language or platform for their component implementations. As shown in Figure 3, key entities of CCM-based component middleware include:

- **Component,** which encapsulates the behavior of the application. Components interact with clients and each other via *ports*, which are of four types: (1) *facets*, also known as provided interfaces, which are end-points that implement CORBA interfaces and accept incoming method invocations, (2) *receptacles*, also known as required connection points, that indicate the dependencies on end-points provided by another component(s), (3) *event sources*, which are event producers that emit events of a specified type to one or more interested event consumers, and (4) *event sinks*, which are event consumers and into which events of a specified type are pushed. The programming artifact(s) that provides the "business logic" of the component is called an *executor*.

- **Container,** which provides an execution environment for components with common operating requirements. The container also provides an abstraction of the underlying middleware and enables the component to communicate via the underlying middleware bus and reuse common services offered by the underlying middleware.

- **Component Home,** which is a factory [42] that creates and manages the life cycle for instances of a specified component type.

- **Component Implementation Framework (CIF),** which defines the programming model for defining and constructing component implementations using the Component Implementation Definition Language (CIDL). CIF automates the implementation of many component features which include generation of programming skeletons and association of components with component executors with their context and homes.

- **Component server,** which is a generic server process that hosts application containers. One or more components can be collocated in one component server.

Component middleware provides a standard "virtual boundary" around application component implementations that interact only via well-defined ports, defines standard container mechanisms needed to execute components in generic component servers, and specifies a reusable/standard infrastructure needed to configure and deploy components throughout a distributed system. Since the Lightweight CCM specification does not standardize the process of deployment, initialization, and interconnection of components, the OMG Deployment and Configuration (D&C) specification was introduced as an addendum to the CCM specification. Our open-source implementation of the D&C specification *Deployment and Configuration Engine* (DAnCE) enables the deployment and configuration of components in DRE systems.

5 Structure and Functionality of RACE

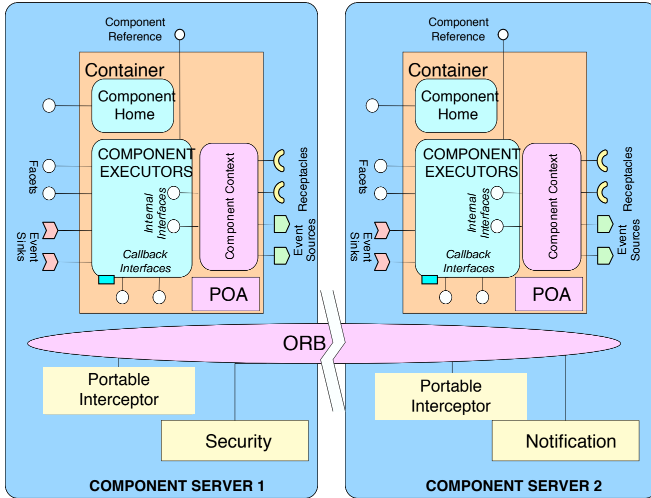


Figure 3: CCM Architectural Overview

As shown in Figure 4, RACE is composed of the following components: (1) `InputAdapter`, (2) `Orchestrator`, (3) `Conductor`, (4) `Allocators`, (5) `Controllers`, (6) `Configurators`, and (7) `Historian`. RACE also monitors application QoS and system resource usage via its `CentralizedQos-Monitor` and `TargetManager` components. All components of RACE are deployed and configured using DAnCE. This section motivates and describes the design of RACE by showing how it resolves the challenges presented in the MMS case study from Section 3.

5.1 Resource Allocation

To allocate resources to applications that execute in an open DRE system, such as the NASA's MMS mission system, RACE performs the following steps: (1) it parses the metadata that describes the application to obtain the resource requirement(s) of components that make up the application, (2) obtains current resource utilization from resource utilization monitors, and (3) selects and invokes an appropriate implementation(s) of resource allocation algorithm depending on the properties of the application and the overhead associated with the implementation(s). Below we describe the RACE components that work together to perform the steps outlined above and resolve the resource allocation challenges of the MMS mission as described in Section 3.2.

1. InputAdapter. End-to-end applications can be composed in many ways. For example, an application can be composed by using a DSML like PICML at system design-time and/or by an intelligent mission planner like SA-POP at run-time. When an application is composed using PICML, metadata describing the application is captured in a XML file based on the `PackageConfiguration` schema defined by the D&C specification [41]. When applications are generated during runtime by SA-POP, metadata is captured in an in-memory structure defined by the planner.

During design time, RACE can be configured using PICML and an `InputAdapter` appropriate for the system can be selected. For example, if applications in the system are constructed at design-time using PICML, RACE can be configured with the `PICMLInputAdapter`; if 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 managed internally by RACE. 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 mapping of components onto nodes need not be specified in the metadata that describes the application

which is given to RACE. If an mapping is specified, it is honored by RACE; if not, a mapping is determined at run-time by RACE's `Allocators`.

2. TargetManager. is an entity of the CCM component middleware as defined in the D&C specification [41] that monitors utilization of system resources. As shown in Figure 4, RACE employs the `TargetManager` to obtain information regarding system resource utilization. `TargetManager` uses a hierarchical design and receives periodic resource utilization updates from `ResourceMonitors` within the domain. It uses these updates to track resource usage of all resources within the domain. Our implementation of the `TargetManager`, *Bulls-Eye* [43], provides an uniform interface for retrieving information pertaining to resource consumption of each component, each node in the domain, as well as the domain's overall resource utilization.

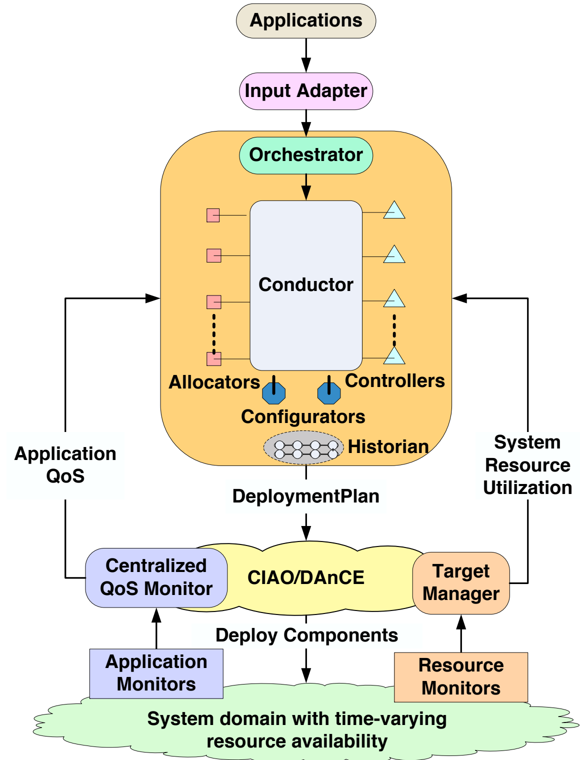


Figure 4: Structure and Interactions in RACE

3. Allocators are implementations of resource allocation algorithms that allocate various domain resources (such as CPU, memory, and network bandwidth) to components of an application by determining the mapping of components onto nodes in the system domain. For certain applications—usually the important ones—*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. Input to `Allocators` include the `E-2-E` IDL structure corresponding to the application and the current utilization of system resources. Since `Allocators` themselves are CCM components, RACE can be configured with new `Allocators` by using PICML.

The current version of RACE supports following algorithms as `Allocators`: (1) CPU allocator, (2) memory allocator, (3) network-bandwidth allocator, (4) PBFD allocator [40] that allocates CPU, memory, and network-bandwidth, and (5) static allocator. Metadata is associated with each allocator and captures its type (*i.e.*, static, single dimension bin-packing [44], or PBFD) and associated resource overhead (such as CPU and memory utilization).

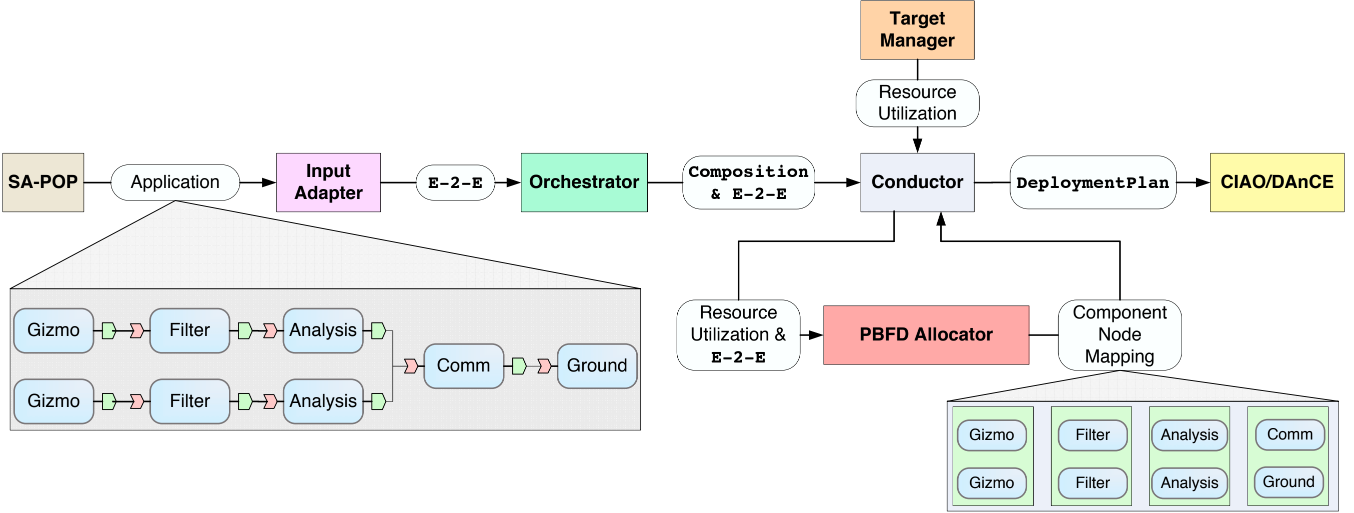


Figure 5: Resource Allocation to Application Components Using RACE

4. Orchestrator and Conductor. As shown in Figure 5, after the metadata describing the application is parsed by RACE's `InputAdapter`, the in-memory `E-2-E` IDL structure is passed onto the `Orchestrator`. This component processes the `E-2-E` structure to determine the types of resources (*e.g.*, CPU, memory, or network bandwidth) required and whether a static allocation is specified. If a static allocation is specified, the static allocator is selected; otherwise a dynamic allocator(s) is selected based on the type(s) of resources required. This selection process is captured in the `Composition` IDL structure shown in Figure 6. For example, consider the data gather-process application generated by SA-POP shown in Figure 2 and described in Section 3. Since these application components have both CPU and memory resource requirements, the `Orchestrator` selects the `PBFD Allocator` which is capable of allocating multiple resources to applications components. This selection is captured in a `Composition`.

As shown in Figure 5, the `Orchestrator` passes the `Composition` and the `E-2-E` to the `Conductor`, which then performs the desired orchestration by invoking the `Allocator(s)` specified in the `Composition`, along with the resource utilization information obtained from the `TargetManager` to map components onto nodes in the system domain. After resources are allocated to the application, the `Conductor` converts the application from RACE's internal `E-2-E` IDL structure into the standard `DeploymentPlan` IDL structure defined by the D&C specification [41]. The `DeploymentPlan` IDL structure is then passed to the underlying DAnCE middleware to deploy the components on the designated target nodes.

Since the elements of RACE are developed as CCM components, RACE itself can be configured using DSML tools, such as PICML. Moreover, new `InputAdapters` and `Allocators` can be plugged directly into RACE without modifying RACE's existing architecture. RACE can be used to deploy and allocate resources to applications that are composed at design-time and run-time. RACE's `Allocators` with inputs from the `TargetManger` allocates resource to application components based on runtime resource availability, thereby addressing the resource allocation challenge for DRE systems identified in Section 3.2.

```
module RACE
{
  // Configurator sequence.
  typedef sequence<Configurator> Configurator_seq;

  struct Composition
  {
    // Configurators that are required to process this application.
    Configurator_seq Configurators;

    // Recommended allocator.
    Allocator recommended_allocator;
  };
};
```

Figure 6: `Composition` IDL Structure

5.2 QoS Parameter Configuration

RACE shields application developers and SA-POP from low-level platform-specific details and defines a higher-level QoS specification model. Developers and/or SA-POP specify only QoS characteristics of the application, such as QoS requirements and relative importance, and RACE automatically configures platform-specific parameters accordingly. Below, we describe the RACE components that work together to provide these capabilities and resolve the QoS configuration challenges of the MMS mission described in Section 3.2.

1. 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) [44] or Maximum Urgency First (MUF) [45] scheduling algorithms. 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.

2. Orchestrator and Conductor. Based on the QoS properties of the application captured in the `E-2-E` IDL structure, the `Orchestrator` selects appropriate `Configurators` to configure QoS properties for the application. As shown in Figure 7, this orchestration is captured in the `Composition` IDL structure and passed onto the `Conductor` along with the

E-2-E IDL structure, which invokes the Configurators specified in the Composition to configure the system QoS parameters for the application.

RACE's Configurators, Orchestrator and Conductor coordinate with one another to configure platform-specific QoS parameters for applications appropriately. These entities provide higher level abstractions and shield system developers and SA-POP from low-level platform-specific details, thus resolving the challenges associated with configuring platform-specific QoS parameters identified in Section 3.2.

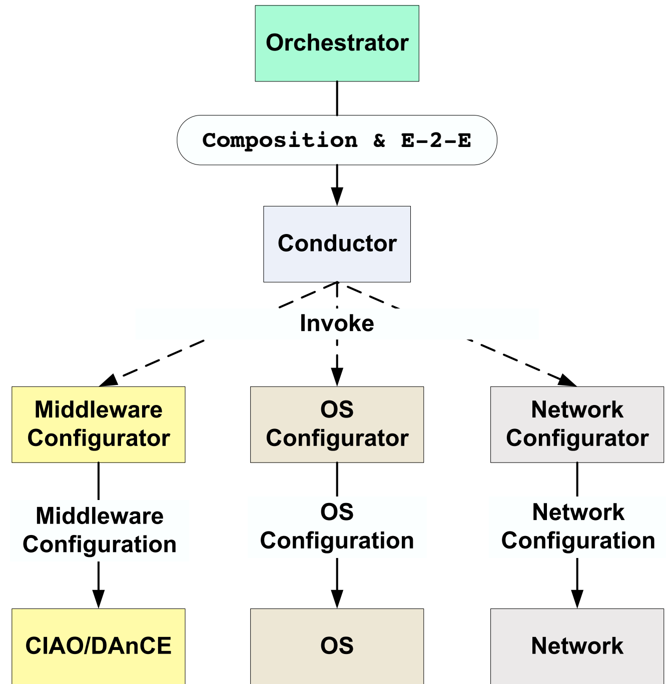


Figure 7: QoS Parameter Configuration with RACE

5.3 Runtime System Management

When resources are allocated to components at design-time by system designers using PICML, *i.e.* 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 resolve these challenges, as well as the ones described in 3.2, 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 8.

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 [17], DEUCON [18], HySUCON [19], and FMUF [20], 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. Below we describe the components RACE's control architecture.

1. Monitors. To ensure system stability and meet QoS requirements of applications, RACE's control architecture must monitor both system QoS and resource utilization. As shown in Figure 8, RACE employs the Lightweight CCM's `TargetManager` to monitor system resource utilization.

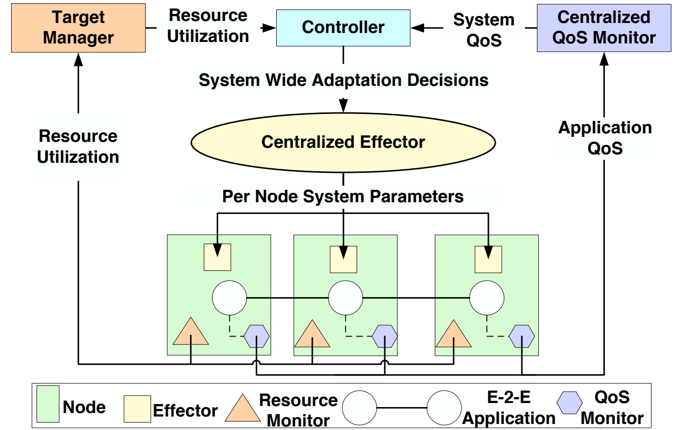


Figure 8: RACE's Feedback Control Loop

As described in Section 4, containers provide application components with an execution environment and enables them to communicate via the underlying middleware. Each container is aware of all the interactions of a component and the end-to-end delay of an application can therefore be measured in an application-transparent way. QoS properties, such as accuracy, precision, and fidelity of the produced output, are application-specific, however, and thus cannot be measured by the middleware without help from application components. We extended the container to embed `Monitors`, called *application-QoS-monitors*, to measure end-to-end application delay.

Since QoS-enabled CCM middleware currently implement inter-component interactions (both *facet/receptacle* interactions and *event source/sink* interactions) as two-way calls, end-to-end delay of an application can obtained by measuring the round-trip delay at the "source" of the application. *Application-QoS-monitors* use high resolution timers (`ACE_High_Res_Timer`) to measure this round-trip delay and periodically send the collected end-to-end delays to the *node-QoS-monitor* that is collocated on the same node using the `Node_QoS_Monitor` interface shown in Figure 9. *Node-QoS-monitors* in turn periodically send the collected end-to-end delay of all the applications on its node to the *centralized-QoS-monitor* using the `Centralized_QoS_Monitor` interface shown in Figure 9. Moreover, application specific QoS monitors can send QoS information to the central monitor by invoking the same interface. The update period of both *application-QoS-monitors* and *node-QoS-monitors* is configurable.

As shown in Figure 8, RACE's QoS `Monitors` are structured in the following hierarchical fashion: an *application-QoS-monitor* tracks the QoS of an application, a *node-QoS-monitor* tracks the QoS of all the applications running on its node, and the *centralized-QoS-monitor tracks* the QoS of all the applications running the entire domain, which captures the system QoS. RACE's `Controller(s)` obtain the system QoS from the *centralized-QoS-monitor* via the `Central_QoS_Monitor` interface shown in Figure 9.

2. Controllers enable a DRE system to adapt to changing operational context and variations in resource availability and/-or demand. The RACE `Controllers` implement various control algorithms that manage runtime system performance, including EUCON [17], DEUCON [18], HySUCON [19], and FMUF [20]. 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 (such as operating system priorities of *component servers* that host the components), and network `difserv` code-points of the component interconnections. `Controllers` are also implemented as CCM components. RACE can therefore be configured with new `Controllers` by using PICML.

```
module RACE
{
  interface Node_QoS_Monitor
  {
    /// Update period.
    attribute long interval;

    /// Oneway method call to push the collected QoS.
    oneway void push_QoS (in string QoS_id, in string Application_id, in any QoS);
  };

  interface Central_QoS_Monitor
  {
    /// No QoS information is available for the requested ApplicationID.
    exception ApplicationIdNotFound { };

    /// No QoS information is available for the requested QoSID.
    exception QoSIdNotFound { };

    /// Oneway method to push the collected QoS.
    oneway void push_QoS (in string QoS_id, in string Application_id, in any QoS);

    /// Retrieve the QoS information regarding a specific QoS of an application.
    any get_QoS (in string QoS_id, in string Application_id)
      raises (ApplicationIdNotFound, QoSIdNotFound);
  };
};
```

Figure 9: Interface Definition of *Node-QoS-Monitor* and *Centralized-QoS-Monitor*

3. 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 8, `Effectors` are designed hierarchically. The *centralized 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 hierarchical design of `ResourceMonitors` (`TargetManager`), `QoSMonitors`, and RACE's `Effectors` is scalable and can handle many applications and nodes in the domain.

4. Historian, Orchestrator, and Conductor. The `Historian` maintains the history of all deployed applications along with their QoS characteristics and mapping of components to nodes. The `orchestrator` employs the `Historian` to obtain information regarding the QoS characteristics of application that have been deployed in the system to select the appropriate controller to manage the system. For example, if all the deployed applications can be operated at various rates,

the Orchestrator selects the EUCON controller to manage the system. The Conductor invokes the controller selected by the Orchestrator to manage the DRE system.

RACE's monitoring framework, controllers, and effectors coordinate with one another and other 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 resolving the challenges associated with runtime end-to-end QoS management identified in Section 3.2.

6 Performance Results and Analysis

This section presents the testbed and experiment configuration inspired by the goals of the NASA MMS mission prototype that we developed to evaluate the empirical performance RACE. We describe our experiments and analyze the results to show the performance of this DRE system with and without RACE under varying operating condition and input workload. The results show that RACE performs effective end-to-end adaptation and yields a predictable and high-performance DRE system.

6.1 Hardware Testbed

Our experiments were performed on the ISISLab testbed at Vanderbilt University (www.dre.vanderbilt.edu/ISISlab). The hardware configuration consists of six nodes acting as the system domain. The hardware configuration of all the nodes was a 2.8 GHz Intel Xeon dual processor, 1 GB physical memory, 1GHz Ethernet network interface, and 40 GB hard drive. Redhat Fedora Core release 4 operating system along with real-time patches running in single processor mode was used for all the nodes.

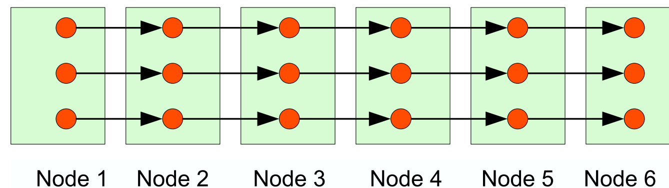


Figure 10: Experiment Topology

6.2 Summary of Evaluated Scheduling Algorithms

We studied the performance of the prototype MMS system under various configurations: including (1) a baseline configuration with no RACE usage at all and with (2) RACE's Rate Monotonic Scheduling (RMS) [44] configurator, (3) RACE's Maximum Urgency First (MUF) [45] configurator, and (4) RACE's MUF configurator and Flexible MUF (FMUF) [20] controller. The RMS and the MUF configurators assign priorities to all components (*component servers*) at deployment time based on the RMS and MUF policies, respectively. A disadvantage of RMS is that it cannot provide performance isolation for important applications [45]. During system overload caused by dynamic workload, a important applications with a low rate may miss deadlines, while a best-effort applications with a high rate may experience no missed deadlines.

In contrast, MUF provides performance isolation to important applications by dividing priorities into two classes [45]. All components belonging to important applications are assigned to the high-priority class, while all components belonging to best-effort applications are assigned to the low-priority class. Components within a same priority class are assigned priorities based on the RMS policy. Relative to RMS, however, MUF may cause priority inversion when an important application has a lower rate than best-effort applications. As a result, MUF may unnecessarily cause a best-effort application to miss its deadline, even when all tasks are schedulable under RMS.

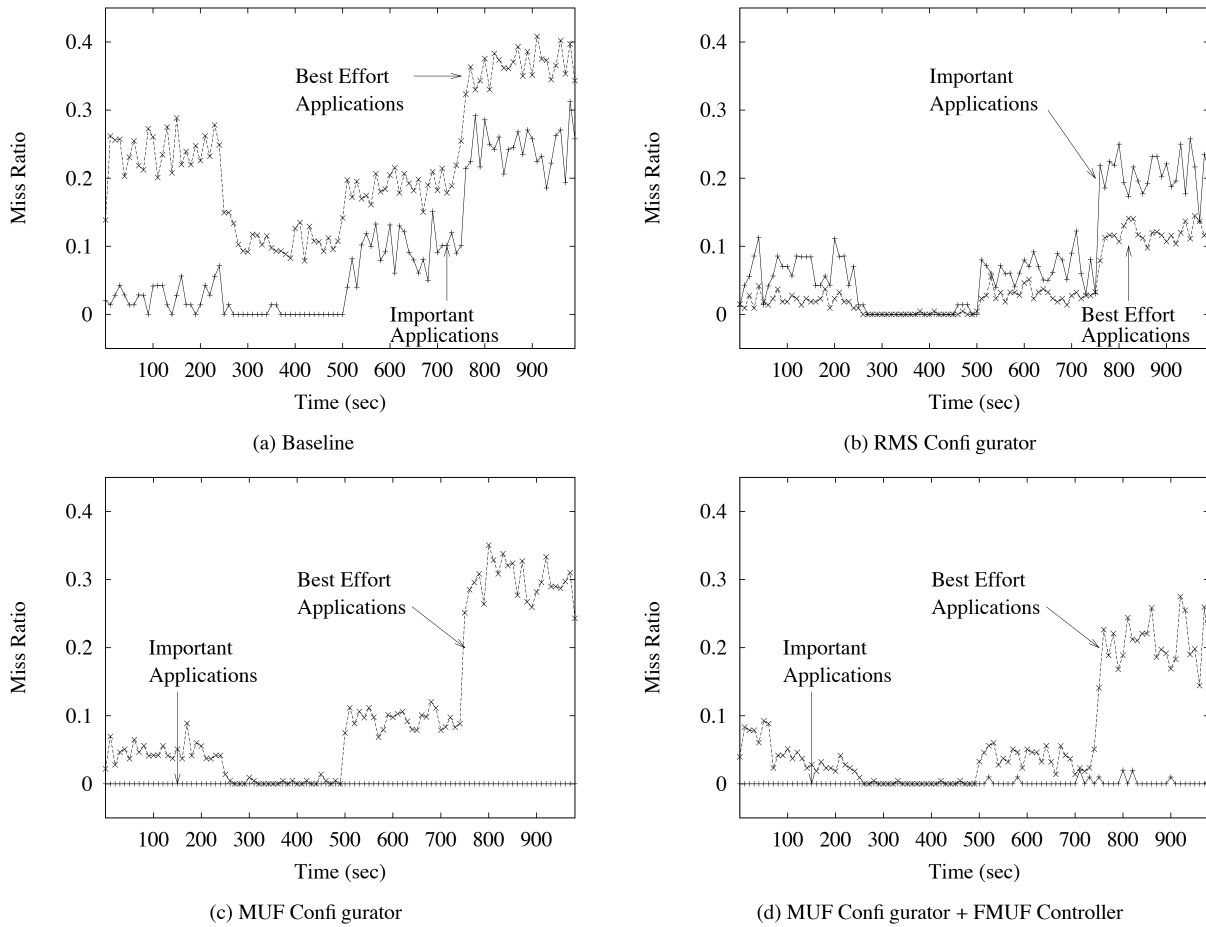
To address the limitation of MUF, RACE's FMUF controller provides performance isolation for important applications while reducing the deadline misses of best-effort applications. While both RMS and MUF assign priorities statically at deployment time, the FMUF controller adjusts the priorities of best-effort applications dynamically based on performance feedback. The FMUF controller can reassign best-effort applications 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. FMUF moves best-effort applications back to the low-priority class when the high-priority class experiences deadline misses. It can therefore effectively deal with workload variations caused by application arrivals and changes in application execution times.

6.3 Experiment Configuration

Our configurations of the prototype NASA MMS Mission DRE system consist of 11 periodic applications, 4 belonging to important (I-M) class and 7 belonging to best-effort (B-E) class. Each application is composed of 6 components (C1–C6) and is subjected to an end-to-end deadline equal to its period. Interconnection of components of applications is shown in Figure 10. Periods of applications along with the estimated execution times of components comprising the applications are described in Table 1. For all applications, static allocation as shown in Figure 10 was specified .

#	Estimated Execution Time (msec)						Period	Class
	C1	C2	C3	C4	C5	C6	(msec)	
1	55	40	65	55	40	65	700	I-M
2	90	65	70	90	65	70	1000	I-M
3	65	70	65	70	55	65	900	I-M
4	35	40	40	35	35	40	500	B-E
5	70	65	65	55	65	70	800	B-E
6	70	65	90	70	90	70	1200	B-E
7	40	55	35	40	40	65	600	B-E
8	65	55	55	70	40	55	700	B-E
9	70	65	65	90	70	65	900	B-E
10	35	40	35	35	40	35	400	B-E
11	65	55	65	70	65	70	700	I-M

Table 1: Properties of End-to-End Applications



(a) Baseline

(b) RMS Confi gurator

(c) MUF Confi gurator

(d) MUF Confi gurator + FMUF Controller

Figure 11: Deadline Miss Ratio

Since the applications described above do not support rate adaptation, RACE employs FMUF as the end-to-end adaptation strategy. Since RACE is a framework, however, other adaptation strategies/algorithms, such as HySUCON [19], can be implemented and employed in a similar way. Below, we evaluate the use of FMUF for end-to-end adaptation. Since the focus of this work is 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.

Our experiments were conducted over 1,000 seconds and we emulated the variation in operating condition and input workload by performing the following. At time $T = 0sec$, we deployed applications 1 through 10. At time $T = 250sec$, we decreased the execution time of all application components by 10 percent to emulate a decrease in input workload. At time $T = 500sec$, we deployed application 11, and at time $T = 750sec$, we increased the execution time of all application components by 10 percent to emulate an increase in input workload. 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.

As described in Sections 1 and 2, QoS-enabled component middleware previously lacked the ability to automatically (1) configure QoS settings of application components and (2) enforce their end-to-end QoS requirements. When the MMS DRE system described above was built atop CIAO/DAnCE directly without RACE, therefore, all application components were assigned default QoS settings, *i.e.* all applications were assigned the same middleware, operating system, and network policies and/or priorities. We use this configuration as the baseline to compare with performance of the system when built atop RACE.

6.4 Analysis of Empirical Results

We now present the results obtained from running the experiment described in Section 6.3 on our ISISlab DRE system testbed described in Section 6.1. We use deadline miss ratio as the metric to evaluate system performance under varying input workloads and operating conditions.

Comparison of QoS. Figures 11a, 11b, 11c, and 11d show the deadline miss ratio of applications when the system operated under the four configurations described in Section 6.3, *i.e.*, baseline configuration, with RACE's RMS configurator, RACE's MUF configurator, and RACE's MUF configurator along with FMUF controller, respectively. These figures show that under all the four configurations, deadline miss ratio of applications (1) reduced at $T = 250sec$ due to the decrease in the input work load, (2) increased at $T = 500sec$ due to the introduction of application 11, and (3) further increased at $T = 750sec$ due to the increase in the input workload. These results demonstrates the impact of fluctuation in input workload and operating conditions on system performance.

Figure 11b shows that when RACE's RMS configurator was used to configure the operating system priorities of component servers, deadline miss ratio of important applications were higher than that of best-effort applications due to reasons explained earlier. Figures 11c and 11d show that when RACE's MUF configurator is used (both individually and along with FMUF controller), deadline miss ratio of important applications were nearly zero throughout the course of the experiment. Figures 11a and 11c demonstrate that RACE improves QoS of our DRE system significantly by configuring platform-specific parameters appropriately.

As described in [20], the FMUF controller responds to variations in input workload and operating conditions (indicated by deadline misses) by dynamically adjusting the priorities of the best-effort applications (*i.e.* moving best-effort applications into or out of the high-priority class). Figures 11a and 11d demonstrate the impact of the RACE's controller on the performance of the system.

Our conclusion from analyzing the results above is that RACE significantly improves the performance of our prototype MMS DRE system even under varying input workload and operating conditions. These benefits result from configuring platform-specific QoS parameters appropriately and performing effective end-to-end adaptation, which were carried out by RACE's `MUF Configurator` and `FMUF Controller` respectively. Moreover, the RACE framework addresses the challenges of building component-based DRE systems identified in Section 3.2.

6.5 Overhead of the RACE Framework.

The runtime overhead of the RACE framework can be decomposed into *monitoring overhead* (the average increase in end-to-end delay as a result of RACE's monitoring framework, as perceived by the application) and *control overhead* (the average execution time of RACE's controller). To measure monitoring overhead, we first instrumented application components with high resolution timers (`ACE_High_Res_Timer`) to measure the end-to-end delay. We next obtained the average end-to-end delay of applications described in Section 6.3 when the system was executed with and without RACE's monitoring framework. We then computed RACE's monitoring overhead as the difference between these two average end-to-end execution times. To measure the control overhead, we also instrumented RACE's controller with a high resolution timer (`ACE_High_Res_Timer`). This timer measured the execution time of the controller during every sampling period. We computed the control overhead as the average of the collected execution times.

From running the experiment described in Section 6.3 on our DRE system testbed, the average monitoring overhead was 37.97 μs and average control overhead of the FMUF controller was 799.82 ns. Average monitoring and control overhead are 0.0645% and 0.0013% of the average estimated execution times of components shown in Table 1. These results demonstrate that the runtime overhead of RACE is small and acceptable for DRE systems.

7 Concluding Remarks

In this paper, we described RACE, which is an adaptive resource management framework that provides end-to-end adaptation and resource management for open DRE systems built atop QoS-enabled component middleware. We demonstrated how RACE helps resolve key resource and QoS management challenges associated with a prototype of the NASA MMS system. We also discussed results from empirical studies of the overhead associated with RACE.

Since the elements of RACE are designed and implemented as CCM components, RACE itself can be configured using DSML tools, such as PICML. Moreover, new `InputAdapters`, `Allocators`, `Configurators`, and `Controllers` can be plugged into RACE using PICML, and without any modifications to the existing architecture. RACE can be used to deploy, allocate resources to, and manage performance of, applications that are composed both at design time as well as at runtime. Moreover, due to the ease with which RACE can be configured, RACE can be employed in a wide range of DRE systems.

Our experience building a prototype of a representative DRE system atop RACE shows that it yields in a predictable and high performance system, even in the face of changing operational conditions, workloads, and resource availability. CIAO, DAnCE, and RACE are available in open-source for download at `deuce.doc.wustl.edu/Download.html`.

References

[1] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma, "Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems," *CrossTalk - The Journal of Defense Software Engineering*, Nov. 2001.

[2] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[3] D. Suri, A. Howell, N. Shankaran, J. Kinnebrew, W. Otte, D. C. Schmidt, and G. Biswas, "Onboard Processing using the Adaptive Network Architecture," in *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, College Park, MD, June 2006.

[4] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, and G. Duzan, "Component-Based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems," in *Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004.

[5] *Real-time CORBA Specification*, OMG Document formal/05-01-04 ed., Object Management Group, Aug. 2002.

[6] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-time Specification for Java*. Addison-Wesley, 2000.

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

[8] D. C. Sharp, E. Pla, K. R. Luecke, and R. J. H. II, "Evaluating Real-time Java for Mission-Critical Large-Scale Embedded Systems," in *IEEE Real-time and Embedded Technology and Applications Symposium*. Washington, DC: IEEE Computer Society, May 2003.

[9] *The Common Object Request Broker: Architecture and Specification, Revision 2.6*, Object Management Group, Dec. 2001.

[10] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, no. 4, pp. 265–290, November/December 1996.

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

[12] Sun Microsystems, "Enterprise JavaBeans Specification," java.sun.com/products/ejb/docs.html, Aug. 2001.

[13] Anne Thomas, Patricia Seybold Group, "Enterprise JavaBeans Technology," java.sun.com/products/ejb/white_paper.html, Dec. 1998, Prepared for Sun Microsystems, Inc.

[14] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill, "QoS-enabled Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed. New York: Wiley and Sons, 2004, pp. 131–162.

[15] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[16] N. Wang and C. Gill, "Improving Real-time System Configuration via a QoS-aware CORBA Component Model," in *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2004*. Kona, HW: HICSS, Jan. 2004.

[17] C. Lu, X. Wang, and X. Koutsoukos, "Feedback Utilization Control in Distributed Real-time Systems with End-to-End Tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 6, pp. 550–561, 2005.

[18] X. Wang, D. Jia, C. Lu, and X. Koutsoukos, "Decentralized utilization control in distributed real-time systems," in *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 133–142.

[19] X. Koutsoukos, R. Tekumalla, B. Natarajan, and C. Lu, "Hybrid Supervisory Control of Real-time Systems," in *11th IEEE Real-time and Embedded Technology and Applications Symposium*, San Francisco, California, Mar. 2005.

[20] Y. Chen and C. Lu, "Flexible Maximum Urgency First Scheduling for Distributed Real-Time Systems," Washington University in St. Louis, Tech. Rep. WUCSE-2006-55, October 2006.

[21] X. Wang, C. Lu, and X. Koutsoukos, "Enhancing the Robustness of Distributed Real-Time Middleware via End-to-End Utilization Control," in *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 189–199.

[22] 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 the 11th Real-time Technology and Application Symposium (RTAS '05)*. San Francisco, CA: IEEE, Mar. 2005, pp. 190–199.

[23] V. F. Wolfe, L. C. DiPippo, R. Bethmagalkar, G. Cooper, R. Johnston, P. Kortmann, B. Watson, and S. Wohlever, "RapidSched: Static Scheduling and Analysis for Real-Time CORBA," *WORDS*, vol. 00, p. 34, 1999.

[24] J. W. Liu, J. Redondo, Z. Deng, T. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha, and W. Shih, "PERTS: A Prototyping Environment for Real-Time Systems," Champaign, IL, USA, Tech. Rep., 1993.

[25] B. Ravindran, L. Welch, and B. Shirazi, "Resource Management Middleware for Dynamic, DependableReal-Time Systems," *Real-Time Syst.*, vol. 20, no. 2, pp. 183–196, 2001.

[26] L. R. Welch, B. A. Shirazi, B. Ravindran, and C. Bruggeman, "DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable Real-time Systems," in *IFACs 15th Symposium on Distributed Computer Control Systems (DCCS98)*. IFAC, 1998.

[27] D. Fleeman, M. Gillen, A. Lenharth, M. Delaney, L. Welch, D. Juedes, and C. Liu, "Quality-Based Adaptive Resource Management Architecture (QARMA): A CORBA Resource Management Service," *IPDPS*, vol. 03, p. 116b, 2004.

[28] C. D. Gill, "Flexible Scheduling in Middleware for Distributed Rate-Based Real-time Applications," Ph.D. dissertation, Department of Computer Science, Washington University, St. Louis, 2002.

[29] K. Bryan, L. C. DiPippo, V. Fay-Wolfe, M. Murphy, J. Zhang, D. Niehaus, D. T. Fleeman, D. W. Juedes, C. Liu, L. R. Welch, and C. D. Gill, "Integrated CORBA Scheduling and Resource Management for Distributed Real-Time Embedded Systems," in *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 375–384.

[30] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[31] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis, "VEST: An Aspect-based Composition Tool for Real-time Systems," in *Proceedings of the IEEE Real-time Applications Symposium*. Washington, DC: IEEE, May 2003, pp. 58–69.

[32] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, pp. 44–51, November 2001.

[33] 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)*. Washington, DC: IEEE, May 2003.

[34] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[35] 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)*. Crystal City, VA: IEEE/IFIP, April/May 2002, pp. 375–385.

[36] P. Manghwani, J. Loyall, P. Sharma, M. Gillen, and J. Ye, "End-to-End Quality of Service Management for Distributed Real-Time Embedded Applications," *ipdps*, vol. 03, p. 138a, 2005.

[37] J. Kinnebrew, N. Shankaran, G. Biswas, and D. Schmidt, "A Decision-Theoretic Planner with Dynamic Component Reconguration for Distributed Real-Time Applications," in *Poster paper at the Twenty-First National Conference on Artificial Intelligence*, Boston, MA, July 2006.

[38] N. Shankaran, X. Koutsoukos, C. Lu, D. C. Schmidt, and Y. Xue, "Hierarchical Control of Multiple Resources in Distributed Real-time and Embedded Systems," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS 06)*, Dresden, Germany, July 2006.

[39] S. Bagchi, G. Biswas, and K. Kawamura, "Task Planning under Uncertainty using a Spreading Activation Network," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 30, no. 6, pp. 639–650, Nov. 2000.

[40] D. de Niz and R. Rajkumar, "Partitioning Bin-Packing Algorithms for Distributed Real-time Systems," *International Journal of Embedded Systems*, 2005.

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

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[43] N. Roy, N. Shankaran, and D. C. Schmidt, "Bulls-Eye: A Resource Provisioning Service for Enterprise Distributed Real-time and Embedded Systems," in *Proceedings of the 8th International Symposium on Distributed Objects and Applications*, Montpellier, France, Oct/Nov 2006.

[44] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *Proceedings of the 10th IEEE Real-time Systems Symposium (RTSS 1989)*. IEEE Computer Society Press, 1989, pp. 166–171.

[45] 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. Tarrytown, NY: Pergamon Press, 1992.