Design and Performance Evaluation of Configurable Component Middleware 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 ‡Dept. of Computer Science and Engineering,
Vanderbilt University, Nashville, TN 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 that execute in open environments where operational conditions, input workload, and resource availability cannot be characterized accurately a priori. Although QoS-enabled component middleware offers many desirable features, until recently it lacked the ability to allocate resources efficiently and configure platform-specific QoS settings based on utilization of system resources and application QoS. Moreover, it has also lacked the ability to monitor and enforce application QoS requirements.

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 the effectiveness of RACE in the context of NASA's Magnetospheric Multi-scale Mission system, which is a representative DRE system.

1 Introduction

Emerging trends and challenges. *Distributed real-time and embedded* (DRE) systems form the core of many mission-critical domains, such as shipboard computing environments, avionics mission computing, multi-satellite missions, and intelligence, surveillance and reconnaissance missions. *Quality of service (QoS)-enabled distributed object computing (DOC) middleware* based on standards like Real-time CORBA (RT-CORBA) and the Real-Time Specification for Java (RTSJ) have been used to develop such DRE systems. More recently, *QoS-enabled component middleware*, such as the Lightweight CORBA Component Model (CCM) [16] and PRiSm [21], have been used as the middleware for DRE systems.

*This work is supported in part by AFRL/IF Pollux Project, Lockheed Martin ATL, and Lockheed Martin ATC.

Compared to QoS-enabled DOC middleware, QoS-enabled component middleware capabilities enhance the design, development, evolution, and maintenance of DRE systems. Examples of additional capabilities offered by QoS-enabled component middleware 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.

In prior work, we developed a domain-specific modeling language (DSML) called the *Platform-Independent Component Modeling Language* (PICML) [1] that alleviates many complexities associated with developing component-based DRE systems using the *Component-Integrated ACE ORB* (CIAO) [24]. CIAO abstracts key Real-time CORBA QoS concerns (such as priority models, thread-to-connection bindings, and timing properties) into elements that can be configured declaratively via Lightweight CCM metadata (such as standards for specifying, implementing, packaging, assembling, and deploying components). PICML enables DRE system developers to (1) design component interfaces and compose applications by interconnecting components, (2) specify QoS and resource utilization characteristics of applications such as end-to-end deadlines, *estimated* CPU, memory, and network bandwidth utilization characteristics, (3) specify middleware, OS, and network configuration parameters, (4) specify *estimated* resource availability of the DRE system, (5) allocate resource to components that make up the application, and (6) generate deployment metadata used by component middleware to deploy applications.

Although CIAO and PICML raise the level of abstraction used to develop DRE systems relative to DOC middleware, unresolved challenges remain. In particular, when DRE system developers allocate resources to, and configure QoS settings of, applications using PICML, these operations are performed based on *estimated* resource utilization of applications and *estimated* availability of system resources. These estimates may be imprecise for open DRE systems

that execute in environments where operational conditions, input workload, and resource availability cannot be characterized accurately *a priori*.

In general, there are limited mechanisms in existing QoS-enabled component middleware platforms to (1) specify end-to-end QoS requirements and (2) monitor application behavior to ensure that these QoS requirements are met. Moreover, when applications are composed at runtime by intelligent mission planners [9], only end-to-end QoS requirements of the applications are specified. What is needed, therefore, are middleware-centric capabilities for allocating resources automatically and monitoring/(re)configuring QoS settings of applications to enforce their end-to-end QoS requirements.

Solution: A component-based adaptive resource management framework. To address the needs of DRE system developers outlined above, we have developed the *Resource Allocation and Control Engine* (RACE), which is an adaptive resource management framework built atop our CIAO QoS-enabled component middleware. As shown in Figure 1, RACE provides (1) *resource monitor* components that track utilization of various system resources, such as CPU, memory, and network bandwidth, (2) *QoS monitor* components that track application QoS, such as end-to-end delay, (3) *resource allocator* components that allocate resources to components based on their resource requirements and current availability of system resources, (4) *configurator* components that configure QoS parameters of application components, (5) *controller* components that compute end-to-end adaptation decisions to ensure that QoS requirements of applications are met, and (6) *effector* components 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. RACE's allocator and controller entities can be configured with multiple resource allocation and control algorithms. This paper provides two contributions to research on adaptive resource management for component-based DRE systems: (1) it describes the component-based design of the RACE framework and (2) we 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 motivates the use of RACE in the context of a representative DRE system; Section 3 describes the architecture of RACE and shows how it meets the QoS requirements of the DRE system described in Section 2; Section 4 compares our research on RACE with related work; and Section 5 presents concluding remarks.

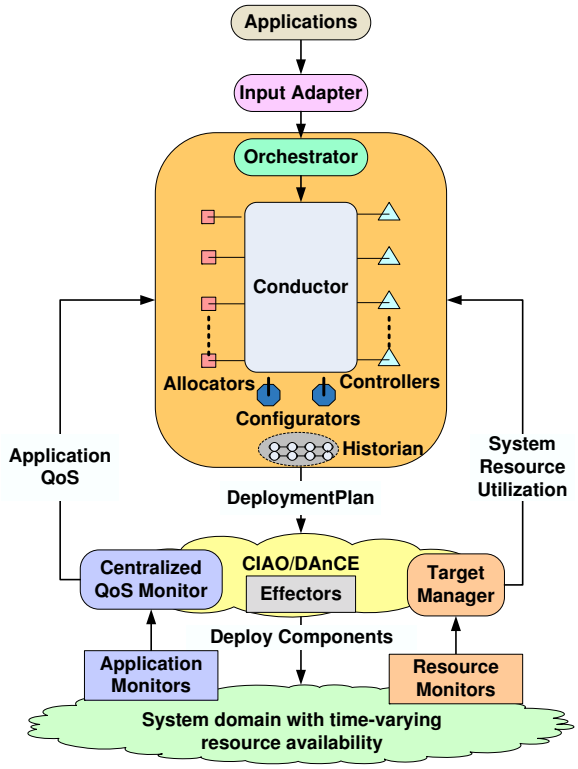

Figure 1: Resource Allocation and Control Engine

2 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. We first present an overview of the MMS mission and then describe the resource and QoS management challenges involved in developing the MMS mission using QoS-enabled component middleware.

2.1 MMS Mission Overview

The goal of the MMS mission is to study the microphysics of three fundamental plasma processes occurring in the earth's magnetosphere: magnetic reconnection, particle acceleration, and turbulence. MMS mission consists of a constellation of identical spacecraft that maintain a specific formation while orbiting over region of scientific interest (ROI). Since the plasma processes are inherently transient (especially magnetic reconnection), MMS missions requires reactive on-board autonomy to enable the spacecraft to transition between three modes of operation: *slow survey*, *fast survey*, and *burst*.

Slow survey mode is entered outside the ROI and enables minimal data acquisition (primarily for health monitoring). The fast survey mode is entered when the spacecraft are within a ROI, which enables data acquisition for all payload

sensors at a moderate rate. While in fast survey mode, the data from a subset of the payload sensors is analyzed on board to detect the likelihood of a transient plasma event. If any plasma activity is detected, all the spacecraft enter the burst mode and all payload instruments acquire data at high rates.

To address the challenges associated with efficient operation of the different configurations/modes outlined above and the transitions between them, on board intelligent mission planners such as *spreading activation partial order planner* (SA-POP) [9] have been developed. SA-POP decomposes overall mission goal(s) into sets of 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. These applications are classified into two classes (*important* and *best-effort*) based on the operation performed by the application. For example, applications that are responsible for the *guidance–navigation–control* of the spacecraft belong to the *important* class, where as data analysis applications belong to the best-effort class.

2.2 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: Efficient resource allocation to applications. Applications generated by SA-POP are *resource sensitive*, *i.e.*, end-to-end QoS is reduced significantly if the required type and quantity of resources are not provided to the applications at the right time. System resources should therefore be allocated in a timely fashion to components of applications such that their resource requirements are met. In open DRE systems like MMS, however, input workload affects utilization of system resources by, and QoS of, applications. These parameters of the applications may therefore vary significantly from their estimated values. Moreover, system resource availability, such as available network bandwidth, may also be time variant. A resource management framework like RACE must therefore support multiple resource allocation strategies to handle the needs of heterogeneous applications, which include guidance, navigation, control, data acquisition, data handling, and data analysis applications.

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 policies, (2) OS 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` codepoints 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.

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 without involving 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.

3 Structure and Functionality of RACE

RACE is built atop the QoS-enabled component middleware CIAO and *Deployment and Configuration Engine* (DAnCE) [5] , which are open-source implementations of the OMG Lightweight CCM [16], Deployment and Configuration (D&C) [15], and RT-CORBA [14] specifications.

As shown in Figure 1, 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 three challenges presented in the MMS case study from Section 2.

3.1 Efficient Resource Allocation

To allocate resources efficiently in an open DRE system, such as 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 2.2.

• **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 an XML file based on the `PackageConfiguration` schema defined by the OMG D&C specification [15]. When applications are generated during runtime by SA-POP, metadata is captured in an in-memory structure defined by the planner.

RACE can be configured using a DSML (such as PICML) along with an appropriate `InputAdapter` that 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 as relative priority) and QoS requirement(s) (such as end-to-end delay), and (4) mapping of components onto domain nodes.[1]

• **TargetManager.** As shown in Figure 1, RACE employs the `TargetManager` to obtain information regarding system resource utilization. `TargetManager` [18] 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.

• **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 mission-critical 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 runtime 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 [4] 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 [11], or PBFD) and associated resource overhead (such as CPU and memory utilization).

• **Orchestrator and Conductor.** After the metadata describing the application is parsed by RACE's `Input-Adapter`, 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` structure.

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 `Target-Manager` 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 [15]. 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

[1] 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`.

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 2.2.

3.2 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 2.2.

• **Configurators** determine values for various low-level platform-specific QoS parameters, such as middleware, OS, 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 policies, based on the class of the application (*important* and *best-effort*). The `OperatingSystemConfigurator` configures OS parameters, such as the Rate Monotonic Scheduling (RMS)-based [11] or Maximum Urgency First (MUF)-based [23] priorities of the *Component Servers* that host the components. 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 (re)configuring RACE using PICML.

• **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 before, this orchestration is captured in the `Composition` IDL structure and passed onto the `Conductor`, 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 components 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 2.2.

3.3 Dynamic System Adaptation

When resources are allocated to components at design-time by system designers using PICML, these operations are performed based on estimated resource utilization of applications and estimated availability of system resources. Allocation algorithms supported by RACE's `Allocators` map 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 or the relationship between the resource requirement and QoS of components that make up the application. In these systems, moreover, operational conditions and resource availability cannot be characterized accurately *a priori*.

To resolve the above described challenges, as well as the ones described in 2.2, RACE's control architecture employs a feedback loop to manage system resource and application QoS and ensure that (1) QoS requirements of applications are met at all times and (2) system stability by maintaining utilization of system resources below their specified set-points. The feedback loop in RACE's control architecture consists of three main components, `Monitors`, `Controllers`, and `Effectors`, as shown in Figure 2 and described below.

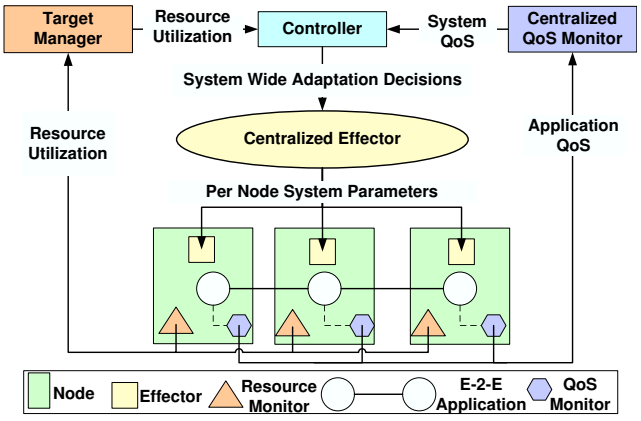

Figure 2: RACE's Feedback Control Loop

• **Monitors.** To ensure system stability and meet QoS requirements of applications, RACE's control architecture monitors both system QoS and resource utilization. As shown in Figure 2, RACE employs the Lightweight CCM's `TargetManager` to monitor system resource utilization.

CCM *containers* provide application components with an execution environment and enables them to communicate via the underlying middleware. Since each container is aware of all interactions of a component, 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 of our middleware (CIAO) to embed `Monitors`, called *application-QoS-monitors*, to measure end-to-end application delay.

CIAO and DAnCE 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 therefore be 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. *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*. 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 2, RACE's QoS `Monitors` are structured in the 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*.

• **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 [13], HySUCON [10], and FMUF [3]. 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 OS priorities of *component servers* that host the components), and network `diffserv` code-points of the component interconnections. RACE can be configured with new `Controllers` by using PICML since `Controllers` are also implemented as CCM components.

• **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 2, `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.

• **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). Thus, RACE resolves the challenges associated with runtime end-to-end QoS management identified in Section 2.2.

4 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.

4.1 QoS-enabled DOC Middleware

Design-time. RapidSched [26] enhances QoS-enabled DOC middleware, such as RT-CORBA, by computing and enforcing distributed priorities. RapidSched uses PERTS [12] 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 [17] motivated the need for adaptive resource management middleware. This work was further extended by QARMA [6], which provides resource management as a *service* for existing QoS-enabled DOC middleware, such as RT-CORBA. Kokyu [7] 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 [2] 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 [1].

4.2 QoS-enabled Component Middleware

Design-time. Cadena [8] 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 [22] is a DSML 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.

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 and Qoskets [19] 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 [9]), 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 [25] and HiDRA [20], 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 and QoS-enabled component middleware run-time platforms. RACE can be used to deploy and manage component-based applications that are composed at design-time via the PICML [1] DSML, as well as at run-time the SA-POP [9] intelligent mission planner (described in Section 2.1). Moreover, RACE's reusable entities, such as resource monitors, QoS monitors, and effectors, can be configured to incorporate a range of existing control algorithms, such as EUCON [13] and HySUCON [10], as well as future algorithms.

5 Concluding Remarks

This paper described 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. We demonstrated how RACE helps resolve key resource and QoS management challenges associated with a prototype of the NASA MMS system.

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 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. CIAO, DAnCE, and RACE are available in open-source for download at `http://deuce.doc.wustl.edu/Download.html`.

References

[1] 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)*, pages 190–199, San Francisco, CA, Mar. 2005. IEEE.

[2] 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*, pages 375–384, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

[5] 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 Working Conference on Component Deployment*, Grenoble, France, Nov. 2005.

[6] 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*, 03:116b, 2004.

[7] C. D. Gill. *Flexible Scheduling in Middleware for Distributed Rate-Based Real-time Applications*. PhD thesis, Department of Computer Science, Washington University, St. Louis, 2002.

[8] 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.

[9] 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.

[10] 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.

[11] 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)*, pages 166–171. IEEE Computer Society Press, 1989.

[12] 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. Technical report, Champaign, IL, USA, 1993.

[13] 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.*, 16(6):550–561, 2005.

[14] Object Management Group. *Real-time CORBA Specification*, OMG Document formal/02-08-02 edition, Aug. 2002.

[15] Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document ptc/03-07-08 edition, July 2003.

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

[17] B. Ravindran, L. Welch, and B. Shirazi. Resource Management Middleware for Dynamic, DependableReal-Time Systems. *Real-Time Syst.*, 20(2):183–196, 2001.

[18] 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.

[19] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal. Packaging Quality of Service Control Behaviors for Reuse. In *Proceedings of the 5^{th} IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, pages 375–385, Crystal City, VA, April/May 2002. IEEE/IFIP.

[20] 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.

[21] 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, May 2003. IEEE Computer Society.

[22] 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*, pages 58–69, Washington, DC, May 2003. IEEE.

[23] D. B. Stewart and P. K. Khosla. Real-time Scheduling of Sensor-Based Control Systems. In W. Halang and K. Ramamritham, editors, *Real-time Programming*. Pergamon Press, Tarrytown, NY, 1992.

[24] 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 Q. Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2003.

[25] 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*, pages 189–199, Washington, DC, USA, 2005. IEEE Computer Society.

[26] 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*, 00:34, 1999.