Addressing Design Challenges of (Re)Deploying Components for Distributed Real-time and Embedded Systems

Abstract

Middleware is increasingly being used to develop and deploy components in large-scale distributed real-time and embedded (DRE) systems. A key challenge in component deployment and execution for DRE systems is devising resource allocation and control algorithms that (1) map application components in DRE systems onto resources available on target nodes and (2) monitor performance to ensure QoS requirements are not violated. The design of placement and control algorithms in DRE systems today involves tedious, error-prone, and human-intensive programming. This paper provides two contributions to the R&D on middleware support to automate the deployment and control of components in DRE systems. First, it describes the design of a Resource Allocation and Control Engine (RACE), which is a middleware framework that integrates multiple resource management algorithms implemented using standard Lightweight CORBA Component Model (CCM) mechanisms for (re)deploying and (re)configuring application components in DRE systems. Second, it shows how RACE helps to decouple resource allocation and system adaptation logic from the time when this logic is applied to configure resource management algorithms.

1. INTRODUCTION

Components are units of implementation and composition that contain parameterizable executable code with quality of service (QoS) requirements (such as maximum latency and minimum throughput values) and resource consumption profiles (such as expected CPU and memory usage). Component-based technologies are increasingly used in large-scale distributed real-time and embedded (DRE) systems, such as shipboard computing environments [16], avionics mission computing systems [19], and intelligence, surveillance and reconnaissance systems [18]. In these systems, applications can be viewed as groups of domain-related tasks that can be implemented by parameterized and executable software components using component technologies, such as the OMG's Lightweight CORBA Component Model(CCM), J2EE, and .NET. Applications generate workflow sequences, which are represented as *operational strings* with each task in the sequence being mapped onto a parameterized and executable component.

Operational strings can be generated by planners that map desired mission goals to workflow sequences. These sequences in turn are deployed as the executable software components. For example, starting from a data collection goal for a spacecraft science mission, the planner may generate an operation string with the following executable components: *system and sensor initilization, sensor calibration, data collection, quality check, data fusion (from multiple sensors), data compression, and telemetry stream communication.*

Operational strings also capture *data dependencies* (*e.g.*, producer/consumer dependencies) and *control dependencies* (*e.g.*, sequential versus parallel execution of components). In addition, these strings capture a range of QoS requirements that may vary in response to runtime changes in mission goals and changes in system performance. Applications can adapt to these changes by running the components that comprise an operational string in different modes and dynamically reconfiguring and/or migrating application component implementation(s).

One approach to allocating and controlling operational strings in DRE systems is to tightly couple handcrafted algorithms and mechanisms [16]. This approach, however, often produces convoluted implementations that can increase algorithm complexity and memory footprint. A more effective design, therefore, is to develop a framework that enables different algorithms for allocating and controlling operational strings to reuse common, automated mechanisms that include (1) capabilities to parse metadata that describe operational string resource requirements and QoS characteristics, (2) monitors that track application and infrastructure performance and resource usage, (3) the ability to represent allocation/control algorithm policies via metadata and automatically configure the middleware to enforce these policies, and (4) the ability to (re)deploy and (re)configure application components in operational strings based on decisions made by allocation and control algorithms.

This paper describes our reusable framework, the *Resource Allocation and Control Engine* (RACE), which separates resource allocation and control *algorithms* from the underlying middleware deployment, configuration, and control *mechanisms* so that different algorithms can reuse common middleware mechanisms to (re)deploy components onto nodes and manage the node's resources among competing applications. RACE enables DRE system developers to con-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

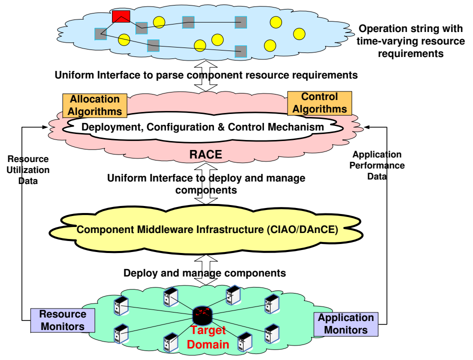

Operation string with time-varying resource requirements
Uniform Interface to parse component resource requirements
Allocation Algorithms
Control Algorithms
Deployment, Configuration & Control Mechanism
RACE
Resource Utilization Data
Application Performance Data
Uniform Interface to deploy and manage components
Component Middleware Infrastructure (CIAO/DAnCE)
Deploy and manage components
Resource Monitors
Target Domain
Application Monitors

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

figure allocation and control algorithms depending on the characteristics of operational strings being deployed and enables the use of multiple algorithms without needing to handcraft the mechanisms used to configure the algorithms.

Figure 1 shows how RACE can support multiple applications running in a variety of DRE system environments and allow applications with diverse QoS requirements to share resources simultaneously. RACE provides a range of resource allocation and control algorithms that use middleware deployment and configuration mechanisms to allocate resources to operational strings and control system performance after operational strings have been deployed.

RACE's algorithms determine how to deploy and redeploy operational strings of application components at system initialization and during runtime. Its allocation algorithms determine the initial component deployment by deciding how to map these components to the appropriate target nodes based on the availability of system resources. Likewise, RACE's control algorithms adapt the execution of an operational strings's components at runtime in response to changing environments and variations in resource availability and/or demand.

RACE uses mechanisms provided by the underlying middleware to perform the allocation and control decisions made by its algorithms. For example, RACE uses standard mechanisms defined by the Lightweight CORBA Component Model (CCM) [14] to (1) (re)deploy and (re)configure application components, (2) transition application components from idle states to operational states and monitor the performance of the DRE system, and (3) modify components and/or operational strings to realize the adaptation decisions of control algorithms.

The remainder of this paper is organized as follows: Section 2 motivates the need for RACE in the context of applications for DRE systems; Section 3 describes the design of RACE and shows how it leverages the OMG Deployment and Configuration (D&C) specification [13] in the Lightweight CCM standard; Section 4 illustrates how we are applying RACE to the applications described in Section 2; Section 5 compares our work on RACE with related research; and Section 6 presents concluding remarks.

2. MOTIVATING APPLICATION SCENARIOS

The computational architecture in Figure 1 is applicable to large-scale DRE systems that perform a number of

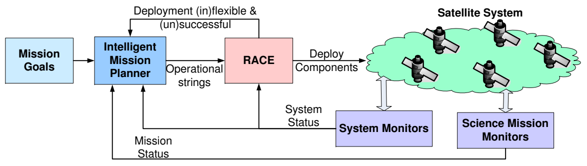


Figure 2: Architecture of Earth Science Enterprise System

coordination and heterogeneous data handling and analysis tasks. An example is NASA's Earth Science Enterprise (ESE) mission, whose goal is to collect significant amounts of atmospheric and earth surface data to enable computational models that can accurately predict climate, weather, and natural hazard occurrences. The architecture of a DRE system that implements the ESE mission is shown in Figure 2. Although these missions have traditionally been conducted using large, independently operated spacecraft, the goals of better physical coverage and richer data collection with a variety of sensors at lower cost motivates the deployment of large networked constellations of satellites [11].

In addition to deploying a constellation of satellites, modern science missions often operate in multiple application modes, such as signal space coverage, combination, and isolation, depending on the current task requirements. For example, the Global Precipitation Measurement (GPM) constellation [15] requires an evenly distributed network of orbiters to sample every point in the globe at periodic time intervals driven by the rate at which thunderstorms can form and dissipate. Depending on evolving weather conditions, the satellite may perform data collection in slow, medium, and fast modes of operation. In other situations, it may be important for separate platforms with different sensors to cooperate and analyze a phenomena, *e.g.*, the Cloudsat and Calipso spacecraft use different sensors to study the relationship between aerosols and precipitation [5]. In these applications, the spacecraft will switch to a special mode where they fly in close coordinated formations to capture different information from the same region.

Future spacecraft on science missions (*e.g.*, NASA's Leonardo-BRDF [20]) will have the ability to operate in multiple modes, such as the ones outlined above. They will also be capable of autonomously switching between modes as conditions and requirements change, *e.g.*, as a new storm system begins to form over the Gulf of Mexico. Mode switching implies that the satellite systems and their computational resources can be reconfigured dynamically during system operation.

A key challenge, therefore, is to develop *intelligent mission planners* [2], which decompose the overall science mission goal(s) into sets of application tasks that can be executed concurrently. As discussed in Section 1, these application tasks can be mapped to parameterized components that are characterized by execution profiles (such as CPU utilization, memory usage, and communication latency) and intelligent resource allocation and control algorithms that can configure application task placement on target nodes so that the overall performance and system utilization meets the QoS requirements. To support efficient resource allocation and adaptation for these operational strings and application scenarios, we need a reusable middleware architecture that resolves the following challenges:

1. Support for converting science mission goals to a sequence of end-to-end application tasks that

are represented as operational strings. The intelligent mission planner employs decision-theoretic and hierarchical task decomposition methods to operationalize mission goals in terms of prioritized navigation, control, data gathering, data fusion, data analysis, and telemetry stream generation tasks. In addition to initial system bootstrapping, the planner also incrementally generates new task sequences (Figure 2) in response to (1) changing mission goals and resource requirements or (2) degraded performance reported by the mission and system monitors. These tasks are indispensable units of computation and can be implemented as software components using standard component middleware technologies.

2. Support for describing the characteristics of the generated tasks in a standard deployment schema. The intelligent mission planner generates (1) the operational strings and a partial schedule for corresponding component deployment, (2) the data and control dependencies between the different components, (3) the execution profiles of each component, and (4) the resource requirements for each component. This application-specific information can be captured via XML metadata to support interoperability among various tools in a science mission [11] and enable the use of standard deployment middleware to deploy those components automatically. To ensure this XML description information is available for processing without incurring the overhead of parsing XML data dynamically, the middleware must provide mechanisms for parsing the metadata once and then storing the parsed information efficiently so it can be transferred quickly across multiple processes at runtime.

3. Support for storing component implementations in a repository for efficient retrieval. The components can be parameterized using different algorithmic implementations with different execution characteristics. Some characteristics tradeoff resource requirements, *e.g.*, memory versus CPU requirements, whereas others point to different application scenarios, *e.g.*, where the data collection rate will be higher during certain time intervals in predefined regions of interest. Still other characteristics may directly relate to a selected component implementation, *e.g.*, minimal memory footprint. During system initialization and at runtime, intelligent resource allocation and control algorithms can help optimize system performance and meet mission goals by selecting the appropriate algorithmic implementations. Support is therefore needed for effectively storing and retrieving component implementations.

4. Support for selecting resource allocation and control algorithms. A single resource allocation and control strategy will not handle all resource allocation and adaptation needs for heterogeneous application components, which include guidance, navigation, control, data acquisition, data handling, and data analysis algorithms [11]. The middleware should therefore provide mechanisms that select different resource allocation and control algorithms depending on the behavior, interactions, and priorities of operational strings composed of application components.

5. Support for sharing common middleware deployment framework. It is tedious and error-prone to hand-craft certain aspects of resource allocation and control algorithms, *e.g.*, locating component binaries and libraries, connecting components using the interaction specification information, and configuring underlying OS and middleware to ensure proper end-to-end QoS. Reimplementing these tasks manually for each algorithm leads to convoluted implementations, increased memory footprint, and longer system development and quality assurance cycles. Middleware should therefore provide mechanisms that efficiently and automatically (1) interact with an intelligent mission planner to convey the resource allocation and control decisions and (2) configure the underlying system resources to ensure end-to-end QoS requirements.

Section 3 describes how the our *resource allocation and control engine* (RACE) leverages the OMG Deployment and Configuration specification [13] in the Lightweight CCM standard [14] to address the challenges described above.

3. THE DESIGN OF RACE

RACE is built atop of CIAO and DAnCE, which are open-source implementations of the OMG Lightweight CCM [14], Deployment and Configuration (D&C) [13], and Real-time CORBA [12] specifications. This section presents a brief overview of CIAO and DAnCE, which are the standard middleware platforms underlying RACE (CIAO, DAnCE, and RACE are available from the CVS repository at `cvs.doc.wustl.edu`). It then describes how RACE enhances these platforms to provide a reusable framework for (re)deploying and (re)configuring components onto nodes and managing node resources among competing operational strings.

3.1 The Middleware Infrastructure Underlying RACE

Overview of CIAO. The OMG Lightweight CCM specification standardizes the development, configuration, and deployment of component-based applications that are not tied to any particular language, OS platform, or network. *Components* in Lightweight CCM are implemented by *executors* and collaborate with other components via *ports*, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components. There are two categories of components in Lightweight CCM: (1) *monolithic components*, which are executable binaries, and (2) *assembly-based components*, which are a set of interconnected components that can either be monolithic or assembly-based (note the intentional recursion).

CIAO is an open source implementation of Lightweight CCM and Real time CORBA [12] specifications by the OMG. The architecture of CIAO is designed based on (1) patterns for composing component-based middleware [21] and (2) reflective middleware techniques to enable mechanisms within the component-based middleware to support different QoS aspects.

Overview of DAnCE. In Lightweight CCM (and hence CIAO), component assemblies are deployed and configured via the OMG D&C [13] specification, which manages the mapping of application components onto nodes in a target environment. The information about the component assemblies and the target environment in which the application components will be deployed are captured in the form of standard XML descriptors. To support automatic deployment and configuration of components based on their descriptors, we developed the *Deployment And Configuration Engine* (DAnCE). DAnCE's runtime framework parses

XML assembly descriptors and deployment plans, extracts connection and deployment information from the descriptors and plans, and then automatically deploys the system into the CIAO component middleware platform and establishes the connections between component ports.[1]

3.2 The Structure and Functionality of RACE

The RACE architecture consists of the entities shown in Figure 3. These entities are implemented as CCM components using CIAO and are deployed via DAnCE. Each entity in the RACE architecture is described below:

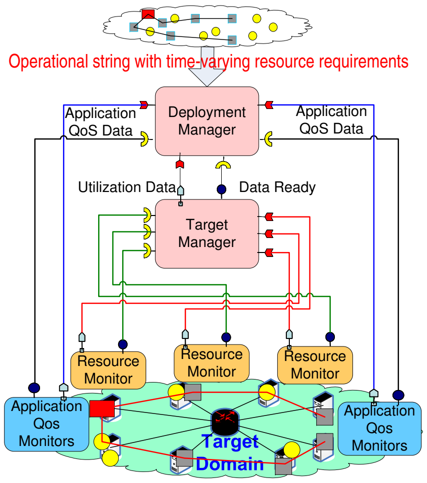


Figure 3: RACE Structure

- **ResourceMonitors** are CCM components that track resource utilization in a domain. One or more `ResourceMonitors` are associated with each domain resource, such as CPU and memory utilization monitors on each node and network bandwidth utilization monitors on interconnects and bridges.
- **ApplicationQoSMonitors** are CCM components that track the performance of application components by observing QoS properties, such as throughput and latency. One or more `ApplicationQoSMonitors` are associated with each type of application component.
- The **TargetManager** is a CCM component defined in the D&C specification [13] that receives periodic resource utilization updates from `ResourceMonitors` within a domain. It uses these updates to track resource usage of all resources within the domain. The `TargetManager` provides a standard interface for retrieving information pertaining to resource consumption of each component and an assembly in the domain, as well as the domain's overall resource utilization. The `TargetManager` provides information on resource utilization component ports in operational strings.
- The **DeploymentManager** is an assembly of CCM components that encapsulates and coordinates one or more allocation and control algorithms. This manager deploys assemblies by allocating resources to individual components in an assembly. After assemblies are deployed, the `DeploymentManager` manages the performance of (1) operational strings and (2) domain resource utilization. This manager ensures desired performance of the operational strings by performing the following actions to the components that make up the operational strings: (1) (re)allocating resources to the component, (2) modifying component parameters such as executional mode, and/or (3) dynamic replacing the component implementations.

[1]In the context of this paper, a *connection* refers to the high-level binding between an object reference and its target component, rather than a lower-level transport (*e.g.*, TCP) connection.

The `DeploymentManager` is the most novel contribution of RACE, so the remainder of this section focuses on its input/output handling, structure and functionality, and extensibility mechanisms.

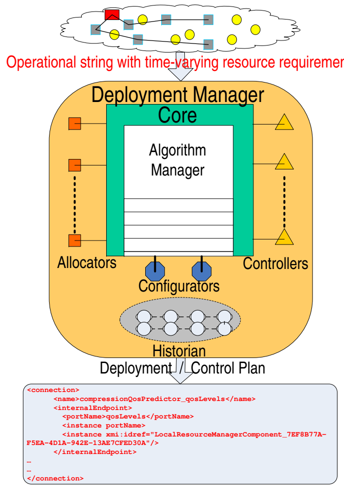
Input and output handling. Two types of inputs are processed by a `DeploymentManager`: *allocation algorithm inputs* and *control algorithm inputs*. Allocation algorithm inputs can be decomposed into static and dynamic inputs. Static inputs include (1) assembly(s) of components to deploy, along with their resource requirements, (2) topology of target domain, and (3) operational strings along with their QoS requirements. The static input is represented in XML descriptors generated off-line via domain-specific modeling tools, such as PICML [4], which can visually define, design, and configure CIAO-based applications. Dynamic inputs capture information regarding current resource utilization/availability in the target domain, which is provided by the `TargetManager`.

Control algorithm inputs are also decomposed into static and dynamic inputs. Static inputs include application end-to-end QoS requirements and bounds on system resource utilization. Dynamic inputs include runtime information from `TargetManager` and `ApplicationQoSMonitors` within the domain, which conveys information related to (1) domain resource utilization/availability and (2) the performance of application components in operational strings, respectively.

Upon receiving the inputs, the `DeploymentManager`'s allocation algorithm computes a feasible allocation of resources to various application components, which is captured in a *resource allocation plan*. If resource allocation to all the input components is not possible an error message is generated. Upon receiving the inputs, the control algorithm verifies whether the end-to-end QoS requirements of applications are met. If end-to-end requirements are met, no control action is necessary. If not, however, the control algorithm may recommend one or more of the following: (1) reallocation of resources to application components, which may involve recomputation of resource allocation, (2) modification of application execution properties, such as execution mode, and/or (3) modification of application component implementations by swapping in different implementations. These control/adaptation decisions are captured in a *control plan*.

A `DeploymentManager` processes the resource allocation plan from the allocation algorithm to produce a *deployment plan*, as defined by the D&C specification, which describes the nodes in a target environment and the type/number of components to be deployed on a node. Likewise, it processes the control plan from the control algorithm to produce a runtime adaptation plan that captures the recommended modification to the application components. These plans then become *policies* that CIAO and DAnCE *mechanisms* (Section 3.1) use to (re)allocate resources to applications and manage system performance.

Structure and functionality. The `DeploymentManager` is implemented as a CCM assembly-based component that is composed of the monolithic components shown in Figure 4 and described below:


Operational string with time-varying resource requirements
Deployment Manager
Core
Algorithm Manager
Allocators
Controllers
Configurators
Historian
Deployment / Control Plan

```
<connection>
      <name>compressionQosPredictor_qosLevels</name>
      <internalEndpoint>
        <portName>qosLevels</portName>
        <instance portName>
        <instance xmi:idref="LocalResourceManagerComponent_7EF8B77A-
F5EA-4D1A-942E-13AE7CFED30A"/>
      </internalEndpoint>
…
…
</connection>
```

Figure 4: Deployment Manager Structure

Allocators are CCM components that implement various resource allocation algorithms used during system initialization to allocate various domain resources, such as CPU, memory, and network bandwidth, among components. Example allocation algorithms include Bin-Packing [8] and Rate-Monotonic General Task Model [9]. Allocators map application components in operational strings to available domain resources via a deployment plan. Allocators are also designed to work with efficient QoS-enabled operating systems, such as the Class-based Kernel Resource Management [?], so that the component resource comsumptions can be allocated as planned and controlled dynamically at run-time. In this context, the allocators can be viewed as a sequence of planners, where members of the sequence could (1) use allocation algorithms, such as Bin-Packing [8], to decide which component needs to be placed on which node and (2) help DAnCE honour those allocation algorithm decisions by deriving component resource reservation mappings. DAnCE places components on nodes and uses the reservation mappings to instruct the OS to provide the necessary resources to the components.

Controllers are CCM components that implement various control algorithms used at runtime to adapt the execution of an application's components at runtime in response to changing operational context and variations in resource availability and/or demand. Example control algorithms include HySUCON [7] and FCS [10]. Controllers can make (1) *coarse-grained control decisions*, which apply to many/all nodes in a domain and can migrate components across nodes or reducing the priority of an operational string, and/or (2) *fine-grained control decisions*, which apply to individual nodes in a domain and can (1) reduce the resource allocation to sepcific component(s) on a node by modifying the component's resource reservation [?], (2) modify the execution mode of the component and/or (3) reconfigure a component's priority.

The **AlgorithmManager** is a CCM component that selects the appropriate `Allocator(s)` and `Controller(s)` that are employed to allocate resources and manage the performance of the application components in an operational string. The selection of algorithms depends on the characteristics and resource requirements conveyed in the metadata associated with an operational string.

Configurators are CCM components that automatically configure the middleware settings (such as threading policy, CORBA priority model and request processing policy) for application components in an operational string. The input to a `Configurator` includes (1) the behavioral characteristics and QoS requirements of each component in an operational string and (2) the deployment plan. The `Configurator` parses (1) the behavioral characteristics of the application components to understand the invocation behavior of the components, (2) the QoS requirements to understand the latency and throughput of such invocations, and (3) the deployment plan to understand the middleware resources present in each node of a domain. The output of a `Configurator` is a configuration plan that specifies to DAnCE and CIAO which middleware settings to configure automatically in each node.

The **Historian** is a CCM component that maintains the current mapping of resource allocations to application components in an operational string. It also maintains information pertaining to past successful and unsuccessful deployment and control plans. Although this information could be stored internally within each algorithm, the `Historian` supports the automated sharing of this information across multiple algorithms to enhance reuse.

Extensibility mechanisms. Our experience developing RACE indicates that one algorithm is not sufficient to manage QoS for DRE systems with many types of applications executing on heterogeneous distributed resources. The `DeploymentManager` therefore supports multiple implementations of resource allocation and control algorithms, as shown in Figure 4. These algorithms can differ in performance and behavior under dynamic operating conditions and application requirements.

The `DeploymentManager` uses the Component Configurator pattern [17] and SwapCIAO [3] to dynamically (re)configure the appropriate algorithms available to make resource allocation and control decisions, depending on operating conditions and application requirements. Together, this pattern and SwapCIAO enable the `DeploymentManager` to link and unlink its algorithm implementations at runtime without having to modify, recompile, statically relink, or shutdown or restart the RACE processes. Moreover, the ability to dynamic (un)link allocation and control algorithms into RACE allows multiple algorithm *policies* to share the same CIAO and DAnCE *mechanisms*, thereby simplifying the development, integration, and comparison of multiple allocation and control algorithms.

4. RESOLVING DRE SYSTEM REQUIREMENTS WITH RACE

To evaluate the pros and cons of the RACE framework described in Section 3, we are applying it to the science mission application scenarios described in Section 2. Figure 5 shows the sequence of actions performed by RACE as the intelligent mission planner generates the sets of operational strings to solve the goal(s) of the Global Precipitation Measurement

(GPM) science mission. The performance of science mis-

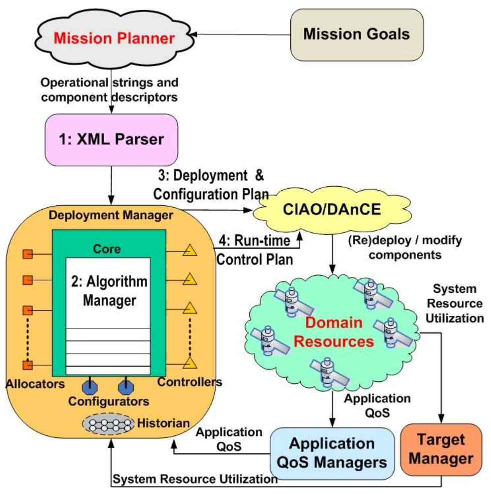


Figure 5: Apply RACE to the GPM Science Mission

sions depend on the application components producing the expected data collection, quality checking, fusion, compression, and transmission in a timely manner. To ensure the success of the mission, RACE performs following activities: (1) allocating adequate resources at appropriate times to the executable components of the operational strings, (2) monitoring resource utilization in the system, and (3) ensuring the QoS requirements specified for each operational string is met. This section describes how the intelligent mission planner and RACE work together to achieve the goals for the GPM multi-satellite DRE system presented in Section 2.

Converting science mission goals to a sequence of end-to-end application tasks that are represented as operational strings. In the GPM mission, an evenly distributed constellation of satellites cover the earth's surface and collect precipitation data in a synchronized manner. The intelligent mission planner employs decision-theoretic and hierarchical task decomposition methods to operationalize mission goals in terms of navigation, control, data gathering, data fusion, data analysis, and telemetry stream generation tasks. The planner can also incrementally generate new task sequences (see Figure 2) in response to changing mission goals and resource requirements, or degraded performance reported by the mission and system monitors.

Describing characteristics of the generated tasks in a standard deployment schema. The intelligent mission planner generates a nearly identical set of operational strings for each satellite, which capture their navigation, control, data capture, data analysis, and data transmission behaviors. Key QoS specifications emphasize synchronization in the data capture process. Other characteristics include the resource consumption profiles of the components, as well as the different modes of operation of the components. These characteristics are captured in the standard D&C [13] specification's *deployment plan*, so that all infrastructural components of RACE can interact using standard interfaces and data.

Efficiently extracting component resource requirements from the deployment plan. RACE parses the XML descriptors of each application component in the operational string to extract application resource and QoS requirements. This information is stored in an in-memory data structure, which the XML parser exposes to the `DeploymentManager` via a strongly typed interface. RACE therefore avoids the runtime overhead of parsing XML at each step, yet retains the information to make allocation and control decisions at initialization- and run-time.

Selecting the resource allocation and control algorithms. The `DeploymentManager` parses the in-memory data structure inputs provided by the XML parser and employs the `AlgorithmManager` that determines the set of `Allocators` and `Controllers` to use for the application components in the operational string. RACE then automatically deploys the corresponding `ApplicationQoSMonitors` and `ResourceMonitors` into the target environment, *i.e.*, the appropriate satellites in the constellation.

Deploying and configuring application components. Using the input from the (1) XML parser, (2) `ApplicationQoSMonitors`, (3) `TargetManager`, (4) `Allocators`, and (5) `Configurators`, RACE's `DeploymentManager` generates the deployment and configuration plans for the science mission's operational strings. Rather than generating individual deployment and configuration plans for each `Allocator` and `Configurator` pair, RACE generates a global deployment plan and conveys this plan to the CIAO and DAnCE middleware (Section 3.1). This separation of concerns allows RACE to use multiple `Allocators` without having to deploy the application components itself, based on the decisions made by the `Allocators`.

Updated application scenario. At some point, the mother satellite may determine that a storm system is developing over the Gulf of Mexico, so operators may decide to track this storm. As a first reorganization step, satellites in the vicinity are asked to accelerate their data collection rates. Meanwhile, part of the GPM system switches from the signal space coverage to the signal isolation mode, which reconfigures these satellites in the original constellation into a new tightly-coupled formation to track the expected path of the storm. The intelligent mission planner responds by generating a new set of operational strings, and the reallocation process is initiated by RACE. The control activities RACE performs in response to these varying operational conditions is summarized next.

System management. After application components are deployed, RACE monitors application performance and domain resource utilization using `ApplicationQoSMonitors` and the `TargetManager`. The accelerated data collection rates results in new QoS requirements for some of the application components. If the performance of an operational string or an individual application component falls below the QoS performance level specified by the mission planner, RACE's `Controllers` will intervene to manage and maintain domain resource utilization. RACE uses the underlying CIAO and DAnCE middleware to fine-tune application properties when applying the coarse-grained and fine-grained control decisions. Similarly, when the mission planner generates a new set of operational strings to implement the the tightly-coupled formation, RACE uses the configured `Allocators` and `Controllers` to allocate resources and manage and maintain domain resource utilization, respectively.

5. RELATED WORK

As component middleware becomes more pervasive, there has been an increase in focus on technologies, platforms, and tools for deploying components effectively within distributed systems. This section compares our work on RACE with related efforts.

The *Autonomic Deployment and Management Engine* (ADME) [6] provides a framework for deploying and autonomically managing application components in distributed systems. Allocating resources to application components in ADME is framed as a constraint solving problem, where domain resources are allocated to application components, subject to specified constraints. ADME uses a domain-specific constraint language called "DEclarative LAnguage for Describing Autonomic Systems" (DELDAS) to specify desired system performance as goals at design time. At runtime, the ADME infrastructure deploys and manages application components to satisfy these goals. RACE has similar motivations as ADME, though RACE provides a pluggable framework where multiple resource allocation and control algorithms can be (re)configured at runtime. RACE also focuses more on the (re)deployment and (re)configuration of QoS-enabled applications executing in DRE systems.

Plaint [1] is a tool that uses a temporal planner to manage and reconfigure a software system. A plan is defined as a sequence of execution steps that ensures desired system performance. Plaint generates to types of plans: (1) *deployment plans* that allocate resources to application components, and (2) *reconfiguration plans* that dynamically reconfigure systems in response to changes in their operation that may be attributed to factors such as external attacks that result in loss of critical application components. The output from various planning techniques can be viewed as deployment plans and control plans that RACE can execute to ensure desired system performance. RACE also augments this planning approach to system reconfiguration by providing the capability to link and unlink various planning mechanisms at runtime to handle system reconfiguration more transparently.

6. CONCLUDING REMARKS

This paper describes the design and application of a Resource Allocation and Control Engine (RACE), which is a middleware framework that integrates multiple resource management algorithms based on standard OMG Lightweight CORBA Component Model (CCM) and Deployment and Configuration capabilities for (re)deploying and (re)configuring operational strings consisting of application components in DRE systems. RACE manages system resource utilization and ensures QoS requirements of operational strings are met even under varying operational contexts and/or varying resource requirement/availability.

Our future work will apply CIAO, DAnCE, and RACE to a broader range of DRE systems, including more science applications presented in Section 2, as well as total ship computing systems [16]. We are evaluating the pros and cons of RACE qualitatively and quantitatively to document lessons learned and pinpoint opportunities for further optimization.

7. REFERENCES

[1] N. Arshad, D. Heimbigner, and A. L. Wolf. Deployment and Dynamic Reconfiguration Planning For Distributed Software Systems. In *Proc. of the 15th IEEE International Conference on Tools With Artificial Intelligence (ICTAI 2003)*, Sacramento, CA, USA, Nov. 2003.

[2] S. Bagchi, G. Biswas, and K. Kawamura. Task Planning under Uncertainty using a Spreading Activation Network. *IEEE Transactions on Systems, Man, and Cybernetics*, 30(6):639–650, Nov. 2000.

[3] J. Balasubramanian, B. Natarajan, D. C. Schmidt, A. Gokhale, G. Deng, and J. Parsons. Evaluating Techniques for Dynamic Component Updating. In *International Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, Oct. 2005.

[4] 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 *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Sym.*, San Francisco, CA, Mar. 2005.

[5] B. J. Clement and A. C. Barrett. Coordination Challenges for Autonomous Spacecraft. In *AAMAS-02 Workshop Notes on Towards an Application Science: MAS Problem Space and Their Implications to Achieving Globally Coherent Behavior*, Bologna, Italy, July 2002.

[6] A. Dearle, G. N. C. Kirby, and A. J. McCarthy. A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications. In *ICAC*, pages 300–301. IEEE Computer Society, 2004.

[7] 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.

[8] 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.

[9] J. Liebeherr, A. Burchard, Y. Oh, and S. H.Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44(12):1429–1442, 1995.

[10] C. Lu. *Feedback Control Real-Time Scheduling*. PhD thesis, University of Virginia, Charlottesville, VA, May 2001.

[11] NASA Science Mission Directorate. NASA Science Missions. `http://science.hq.nasa.gov/directorate/index.html`, 2004.

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

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

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

[15] C. S. Ruf, C. M. Principe, and S. P. Neek. Enabling Technologies to Map Precipitation with Near-Global Coverage and Hour-Scale Revisit Times. In *Proc. of IEEE Intl. Geoscience and Remote Sensing Symposium (IGARSS)*, Honolulu, HI, July 2000.

[16] 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*, Nov. 2001.

[17] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[18] 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.

[19] 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.

[20] G. Silverman, K. Bhasin, L. Capots, D. Enlow, and J. Sroga. Technology Drivers for Space-Based Science Communication. In *IEEE Military Communications Conference (MILCOM 2001)*, Vienna, Virginia, Oct. 2001.

[21] M. Volter, A. Schmid, and E. Wolff. *Server Component Patterns: Component Infrastructures Illustrated with EJB*. Wiley Series in Software Design Patterns, West Sussex, England, 2002.