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

N. Shankaran, J. Balasubramanian, D. Schmidt, and G. Biswas

Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN 37203, USA[*]

Abstract

Middleware is increasingly used to develop and deploy components in large-scale distributed real-time and embedded (DRE) systems. A key challenge in component deployment for DRE systems is devising resource allocation and control algorithms that map application components in DRE systems onto resources available on target nodes. Designing and evaluating these algorithms in a DRE system today, however, often involves tedious, error-prone, and human-intensive programming tasks. This paper provides two contributions to R&D on middleware support for automating the deployment 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 based on standard Lightweight CORBA Component Model (CCM) mechanisms for (re)deploying and (re)configuring application components in DRE systems. Second, it shows how developers of DRE systems can use RACE to decouple resource allocation and system adaptation logic from the time when this logic is applied to a system to configure the resource management algorithms.

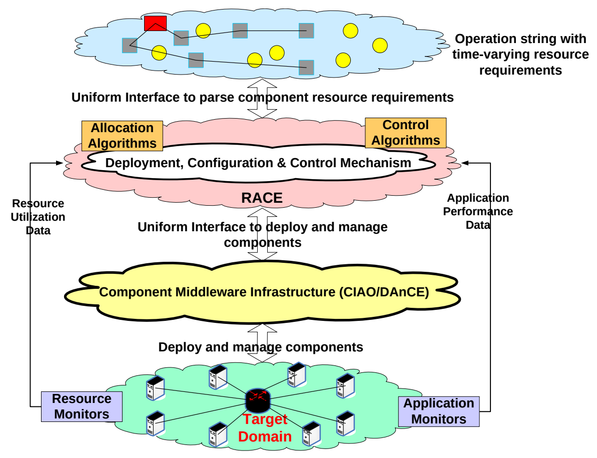
1 Introduction

Component-based technologies are increasingly being applied to large-scale distributed real-time and embedded (DRE) systems, such as shipboard computing environments [1], avionics mission computing systems [2], and intelligence, surveillance and reconnaissance systems [3]. Applications in these DRE systems can be defined as a set of *operational strings*, which capture interdependent computational component workflow sequences from a single application unit (such as the guidance system of a ship or spacecraft) or between multiple applications (such as the navigation and propulsion applications for a ship). Dependency relations can be (1) *data dependencies*, such as producer/consumer dependencies, and/or (2) *control dependencies*, where an application component can execute only after the completion of a preceding component.

Applications in DRE systems have a range of quality of service (QoS) requirements that may vary in response to (1) changes in mission goals at runtime, *e.g.*,

[*] This work is supported in part by funding from NSF, LMCO ATC, LMCO ATL, Raytheon, and Siemens.

due to new information or because certain tasks cannot be completed on time, and (2) changes in system performance, *e.g.*, due to loss of resources, transient overload and/or algorithmic properties. Applications can adapt to these changes by running the components in their operational strings in different modes and dynamically reconfiguring and migrating application component implementations.



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
Application Monitors
Target Domain

Fig. 1. Resource Allocation and Control of Applications in DRE Systems

To support different types of applications running in various DRE system environments – as well as to allow applications with diverse QoS requirements to share resources simultaneously – a range of resource allocation and control *algorithms* and *mechanisms* are needed to (1) allocate resources to operational strings and (2) control system performance after operational strings have been deployed, as shown in Figure 1.

Algorithms are responsible for deciding how best to deploy and redeploy operational strings of application components at system initialization and during runtime. Allocation algorithms determine the initial component deployment by deciding how to map these components to the appropriate target nodes. For example, an allocation algorithm could apportion CPU resources to components in such a way that avoids saturating these resources. Likewise, 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. For example, a control algorithm could (1) modify an application's current operating mode, (2) dynamically update component implementations, and/or (3) redeploy all or part of an operational string's components to other target nodes to meet end-to-end QoS requirements.

Mechanisms are responsible for performing the decisions of the algorithms, namely allocation and control decisions by the respective algorithms. For example, mechanisms can (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.

One way to develop resource allocation and control solutions for DRE systems is to tightly couple handcrafted algorithms and mechanisms. While this approach is common, it yields convoluted implementations that can increase the algorithm complexity and memory footprint. It also involves tedious and error-prone human-intensive programming tasks, such as (1) specifying and evaluating component resource requirements, (2) examining component behavioral and interaction characteristics to identify which resource management algorithm(s) are best suited for deployment, (3) identifying resources available on target platform(s), (4) devising application-specific monitors that identify changes in dynamic operating conditions, and (5) interacting with middleware infrastructure mechanisms that (re)configure and (re)deploy components based on decisions made by allocation and control algorithms.

A more effective design, therefore, is to develop a framework that enables different allocation and control algorithms to reuse common, automated mechanisms that include (1) capabilities to parse metadata that describe application 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 the application components based on the decisions made by the allocation and control algorithms.

This paper describes one such reusable framework, the *Resource Allocation and Control Engine* (RACE), that we have developed in conjunction with colleagues at Lockheed Martin Advanced Technology Lab and Advanced Technology Center. As shown in Figure 1, RACE separates resource allocation and control algorithms from the underlying middleware deployment, configuration, and control mechanisms so that different algorithms can reuse common mechanisms to (re)deploy components onto nodes and manage the node's resources among competing applications. These capabilities enable DRE system developers to configure allocation and control algorithms depending on the characteristics of the operational strings being deployed and enables the use of multiple algorithms without needing to handcraft the mechanisms used to configure the 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 [4] in the Lightweight CORBA Component Model (CCM) standard [5]; 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

Figure 1 presented a computational architecture applicable to large-scale DRE systems that perform a number of 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 studies have

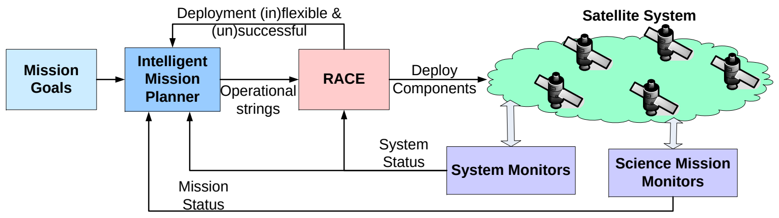


Fig. 2. Architecture of Earth Science Enterprise System

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 [6, 7].

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 [8] 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. In other situations, it may be important for separate platforms with different sensors to cooperate and analyze a phenomena, *e.g.*, Cloudsat and Calipso use different sensors to study the relationship between aerosols and precipitation [9]. In these applications, spacecraft fly in close coordinated formations to capture different information from the same region.

Future science missions will likely combine and simultaneously operate in all the modes of operation outlined above, while autonomously switching between modes as conditions and requirements change, *e.g.*, as a new storm system begins to form over the Gulf of Mexico. NASA's Leonardo-BRDF [10] is an example of such a multi-functional, multi-platform, and multi-configuration system. To achieve this capability, satellite systems and their computational resources will need to be reconfigured dynamically during system operation.

A key challenge, therefore, is to develop reusable DRE system middleware that supports efficient operation of the different configurations outlined above, and the transitions between these configurations. To address this challenge, we are developing an *intelligent mission planner* [11] that decomposes the overall science mission goal(s) into sets of tasks that can be executed concurrently. This mission planner employs decision-theoretic methods and other AI schemes (such as hierarchical task decomposition) to decompose mission goals into navigation, control, data gathering, and data execution tasks. In addition to initial plan generation, the planner can 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.

Each computational task sequence generated by the intelligent mission planner translates to an operational string of application components. The set of operational strings for an application can be represented as a partial-order dependency graph, where the application components form the nodes of the graph, and the edges capture the dependency relations. The graph generated by our intelligent mission planner captures the possible sequential and concurrent execution sequences of applications in a DRE system. To support efficient resource allocation and adaptation for these operational strings, we need a reusable middleware architecture that resolves the following challenges:

1. Support for efficient parsing of operational string resource requirements. The intelligent mission planner generates (1) the sequence of application components to be deployed for each operational string, (2) the interactions between components in a string and between strings, (3) the behavioral characteristics of each component, (4) the resource requirements for each component, and (5) the QoS characteristics of each component. To support interoperability among various tools in a science mission [6], such application-specific information can be captured via XML metadata. To ensure that this information is available for processing – and to prevent the runtime overhead of parsing XML data – the middleware must provide mechanisms for parsing the metadata once, and store the parsed information efficiently so it can be transferred quickly across multiple processes at runtime.

2. 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 [6]. 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.

3. Support for sharing common middleware deployment framework. Implementing resource allocation and control decisions of algorithms is error-prone and tedious, *e.g.*, it includes 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. A reusable middleware framework should therefore provide mechanisms that efficiently and automatically (1) interact with the 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 the design of a resource allocation and control engine that leverages the OMG Deployment and Configuration specification in the Lightweight CCM standard to address the challenges described above.

3 The Design of RACE

The architecture of the Resource Allocation and Control Engine (RACE) is shown in Figure 1. This section presents a brief overview of CIAO and DAnCE, which are the standard middleware platforms underlying RACE. It then describes how RACE enhances these platforms to provide a reusable framework for (re)deploying components onto nodes and managing node resources among competing operational strings.

3.1 RACE Middleware Infrastructure

As shown in Figure 1, the Resource Allocation and Control Engine (RACE) is a framework built atop of our CIAO and DAnCE, which are our open-source[1] implementations of the OMG Lightweight CCM [5], Deployment and Configuration (D&C) [4], and Real-time CORBA [12] specifications.

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

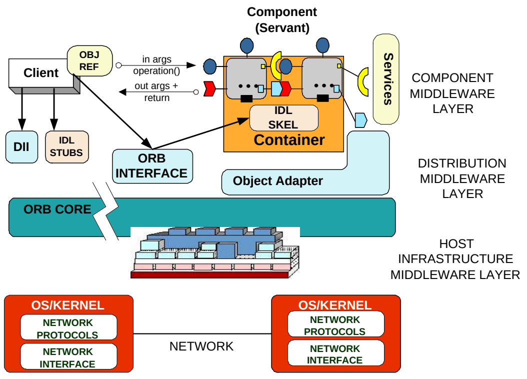


Fig. 3. CIAO Architecture

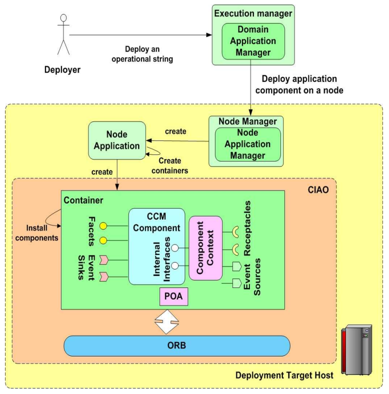


Fig. 4. DAnCE Architecture

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).

[1] CIAO and DAnCE are available from `www.dre.vanderbilt.edu/CIAO`.

A *container* in Lightweight CCM provides the runtime environment for one or more components that provides various pre-defined hooks and strategies, such as persistence, event notification, transaction, and security, to the component(s) it manages. Each container is responsible for initializing instances of the components it manages and mediating their access to other components and common middleware services. A *NodeApplication* is a component server factory that creates containers and provides the run-time process context.

CIAO [13] is an open-source implementation of the OMG Lightweight CCM and Real-time CORBA [12] specifications. CIAO's architecture (shown in Figure 3) is designed based on (1) patterns for composing component-based middleware [14] and (2) reflective middleware [15] techniques to enable mechanisms within the component-based middleware to support different QoS aspects [16].

Overview of DAnCE. In Lightweight CCM, component assemblies are deployed and configured via the OMG D&C [4] 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), whose architecture is shown in Figure 4. 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.[2]

DAnCE implements a set of runtime interfaces defined by the OMG D&C specification that handle the instantiation, installation, connection establishment, monitoring, and termination of components on nodes in a target environment. The D&C specification calls this target environment a *domain*, which consists of *nodes*, *interconnects*, *bridges*, and *resources*. The standard D&C interfaces implemented by DAnCE include (1) the `ExecutionManager` and `NodeManager`, which manage the lifecycle of the deployment process to help configure component servers on the nodes, install components into containers, and set up connections among components that may be distributed across multiple nodes, and (2) the `DomainApplicationManager` and `NodeApplicationManager`, which split a deployment plan across multiple domains so that various QoS concerns (such as security and fault isolation) can be enforced. The `ExecutionManager` and `DomainApplicationManager` operate at the global domain level, whereas the `NodeManager` and `NodeApplicationManager` operate at each node of the domain.

3.2 The Structure and Functionality of RACE

The RACE architecture consists of the entities shown in Figure 5. These entities

[2] 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.

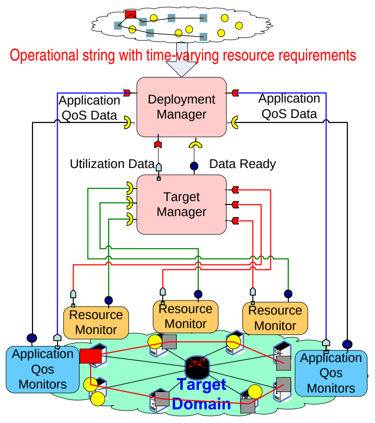
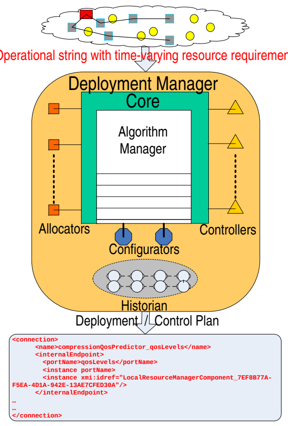


Fig. 5. RACE Structure


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>
```

Fig. 6. `DeploymentManager` **Structure**

are implemented as CCM components using CIAO and are deployed via DAnCE. We describe each entity below:

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.

TargetManager is a CCM component defined in the D&C specification [4] 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.

DeploymentManager is an assembly of CCM components that encapsulates and coordinates one or more allocation and control algorithms. The `Deployment-Manager` deploys assemblies by allocating resources to individual components in an assembly. After an assemblies is deployed, the `DeploymentManager` manages the performance of (1) operational strings and (2) domain resource utilization. The `Deployment Manager` ensures desired performance of the operational strings by performing the following actions to the components that make up the operational strings: (a) (re)allocating resources to the component, (b) modifying component parameters such as executional mode, and/or (c) dynamic replacing the component implementations. 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*, which 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 [17], 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 primarily dynamic and include run-time information from `TargetManager` and `ApplicationQoSMonitors` within the domain. This information conveys (1) domain resource utilization/availability and (2) performance corresponding to application components in operational strings.

A `DeploymentManager` processes allocation algorithm input to produce a resource allocation plan, known as a *deployment plan* in 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 control algorithm input to produce a runtime control plan composed of recommended adaptations, such as changes in application properties and/or reallocation of system resources to application components. These plans then become *policies* that the CIAO and DAnCE *mechanisms* described in Section 3.1 use to (re)allocate resources to applications and manage system performance.

Structure and functionality. The `DeploymentManager` is itself implemented as a CCM assembly-based component that is composed of the monolithic components shown in Figure 6 and described below:

- **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 [18] and Rate-Monotonic General Task Model [19]. Allocators map application components in operational strings to available domain resources via a deployment plan.
- **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 [20] and FCS [21]. 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 reduce the rate at which a component makes invocations on another component or 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 the 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 the middleware settings that need to be configured automatically in each node in a domain. This plan is used to configure the CIAO and DAnCE middleware to realize a concrete QoS-aware component middleware that attempts to satisfy the QoS goals of application components in operational strings.
- 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 resource allocation and control engines for DRE systems 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 6. These algorithms can differ in performance and behavior under dynamic operating conditions and application requirements.

The `DeploymentManager` uses the Component Configurator pattern [22] to dynamically (re)configure the appropriate algorithms available to make resource allocation and control decisions, depending on operating conditions and application requirements. This pattern enables the `DeploymentManager` to link and unlink its algorithm implementations at runtime without having to modify, recompile, statically relink, or shutdown/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

We are applying the RACE framework described in Section 3 to the science mission application scenarios described in Section 2. For example, Figure 7 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. Since the performance of science

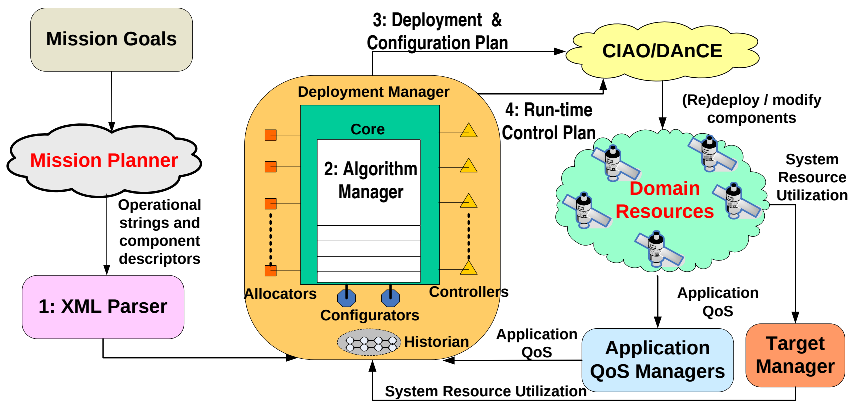


Fig. 7. Apply RACE to the GPM Science Mission

missions depend on the performance of their operational strings, RACE is responsible for (1) allocating resources to application components and assemblies that comprise these operational strings, (2) monitoring resource utilization in the system, and (3) ensuring the QoS requirements of these operational strings are met. This section describes how the intelligent mission planner and RACE together achieve the goals of the GPM multi-satellite DRE system presented in Section 2.

Initial application scenario. 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 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. Data captured by each satellite is collected in a central location, such as a "mother ship," that performs data processing and analysis before transmitting the processed information to earth. The QoS of the science mission thus depends heavily on timely data synchronization among the satellites and between satellites and the mother ship.

The mission planner generates several artifacts, including (1) operational string(s) to execute to achieve the science mission goals, (2) information that captures the interaction, behavioral, and resource requirements of components in the operational string(s), and (3) QoS requirement(s) associated with each operational string. This information is passed by the mission planner to RACE, which (re)allocates resources, (re)deploys, and manages the application components and assemblies to ensure QoS requirements of operational strings are met. Below, we summarize various activities performed by RACE to deploy and allocate resources to the science mission's operational strings.

1. Efficiently parse operational string resource requirements. RACE parses the XML descriptors of the GPM mission's components to obtain application QoS and resource requirements. This information is stored in an in-memory data structure, which the XML parser exposes to the `DeploymentManager` via a strongly typed interface. Therefore, RACE avoids the runtime overhead of parsing XML at each step, yet retains the information to make allocation and control decisions.

2. Selecting the resource allocation and control algorithms. The `Deployment-Manager` 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 present in the operational string. RACE then automatically deploys the corresponding `ApplicationQoS-Monitors` and `ResourceMonitors` into the target environment, *i.e.*, the appropriate satellites in the constellation.

3. Deploying and configuring application components. Using the input from the (1) XML parser, (2) `ApplicationQoSMonitors`, (3) `TargetManager`, (4) `Allocators`, and (5) `Configurators`, the `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 described in Section 3.1. This separation of concerns allows RACE to use multiple `Allocators` without having to perform the complex and tedious tasks of deploying the application components by 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 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 below.

4. System Management Once the application components are deployed, RACE monitors application performance and domain resource utilization using the `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, the `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 three recent related efforts.

The *Autonomic Deployment and Management Engine* (ADME) [23] 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.

[24] presents a component middleware framework for distributed systems that aids the process of dynamic system reconfiguration by (1) starting and stopping application components, (2) migrating components between hosts a domain, and (3) dynamically modifying component implementations to maintain the desired QoS of applications with varying operational contexts, resource requirements, and resource availability. Although this framework provides capabilities to transparently reconfigure distributed systems, it does not provide mechanisms to bootstrap the system by performing initial resource allocation to application components. RACE augments this approach by offering capabilities to plug-in various allocation algorithms that bootstrap the system and control algorithms that drive system reconfiguration.

Plaint [25] 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 run-time 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 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 of the science applications presented in Section 2, as well as total ship computing systems [1]. We plan to evaluate the pros and cons of RACE qualitatively and quantitatively.

References

1. Schmidt, D.C., Schantz, R., Masters, M., Cross, J., Sharp, D., DiPalma, L.: Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. CrossTalk (2001)
2. Sharp, D.C., Roll, W.C.: Model-Based Integration of Reusable Component-Based Avionics System. In: Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003. (2003)
3. Sharma, P., Loyall, J., Heineman, G., Schantz, R., Shapiro, R., Duzan, G.: 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 (2004)
4. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document ptc/03-07-08 edn. (2003)
5. Object Management Group: Light Weight CORBA Component Model Revised Submission. OMG Document realtime/03-05-05 edn. (2003)
6. NASA Science Mission Directorate: NASA Science Missions. `http://science.hq.nasa.gov/directorate/index.html` (2004)
7. Esper, J., Wiscombe, W., Neeck, S., Ryschkewitsch, M.: Leonardo-BRDF: A New Generation Satellite Constellation. In: 51st International Aeronautical Congress, Rio de Janerio, Brazil (2000)
8. Ruf, C.S., Principe, C.M., Neek, S.P.: 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 (2000)
9. Clement, B.J., Barrett, A.C.: 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 (2002)

10. Silverman, G., Bhasin, K., Capots, L., Enlow, D., Sroga, J.: Technology Drivers for Space-Based Science Communication. In: IEEE Military Communications Conference (MILCOM 2001), Vienna, Virginia (2001)
11. Bagchi, S., Biswas, G., Kawamura, K.: Task Planning under Uncertainty using a Spreading Activation Network. IEEE Transactions on Systems, Man, and Cybernetics **30** (2000) 639–650
12. Object Management Group: Real-time CORBA Specification. OMG Document formal/02-08-02 edn. (2002)
13. Wang, N., Schmidt, D.C., Gokhale, A., Rodrigues, C., Natarajan, B., Loyall, J.P., Schantz, R.E., Gill, C.D.: QoS-enabled Middleware. In Mahmoud, Q., ed.: Middleware for Communications. Wiley and Sons, New York (2003) 131–162
14. Volter, M., Schmid, A., Wolff, E.: Server Component Patterns: Component Infrastructures Illustrated with EJB. Wiley Series in Software Design Patterns, West Sussex, England (2002)
15. Schmidt, D.C.: Adaptive and Reflective Middleware for Distributed Real-time and Embedded Systems. In: EMSOFT 2001: First Workshop on Embedded Software., Lake Tahoe, CA (2001)
16. Wang, N., Gill, C., Schmidt, D.C., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus (2004)
17. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: 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 (2005)
18. Lehoczky, J., Sha, L., Ding, Y.: The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In: Proceedings of the 10th IEEE Real-Time Systems Symposium, IEEE Computer Society Press (1989) 166–171
19. Liebeherr, J., Burchard, A., Oh, Y., H.Son, S.: New strategies for assigning real-time tasks to multiprocessor systems. IEEE Trans. Comput. **44** (1995) 1429–1442
20. Koutsoukos, X., Tekumalla, R., Natarajan, B., Lu, C.: Hybrid Supervisory Control of Real-Time Systems. In: 11th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, California (2005)
21. Lu, C.: Feedback Control Real-Time Scheduling. PhD thesis, University of Virginia, Charlottesville, VA (2001)
22. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2. Wiley & Sons, New York (2000)
23. Dearle, A., Kirby, G.N.C., McCarthy, A.J.: A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications. In: ICAC, IEEE Computer Society (2004) 300–301
24. Chen, X., Simons, M.: A Component Framework for Dynamic Reconfiguration of Distributed Systems. In: CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment, London, UK, Springer-Verlag (2002) 82–96
25. Arshad, N., Heimbigner, D., Wolf, A.L.: 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 (2003)