A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems

Douglas C. Schmidt [a] Jaiganesh Balasubramanian [a]
Aniruddha Gokhale [a] Patrick Lardieri [b] Gautam Thaker [b]
Tom Damiano [b]

[a] *Dept of EECS, Vanderbilt University, Nashville, TN 37235, USA*
[b] *Advanced Technology Labs, Lockheed Martin, Cherry Hill, NJ 08002, USA*

Abstract

Total ship computing environments (TSCEs) are examples of enterprise distributed real-time and embedded (DRE) systems that can benefit from dynamic management of computing and networking resources for their sustained successful operation. An essential property of TSCEs is their ability to optimize and reconfigure system resources at runtime in response to changing mission needs and/or other situations, such as failures or system overload. This paper provides three contributions to the study of dynamic resource management (DRM) for enterprise DRE systems. First, we describe a standards-based multi-layered resource management (MLRM) architecture that provides DRM capabilities to enterprise DRE systems. Second, we show how our MLRM architecture has been applied to meet DRM needs of TSCEs. Third, we illustrate the results of experiments evaluating our MLRM architecture in the context of a representative TSCE.

Key words: Dynamic Resource Management, Enterprise DRE Systems, QoS-enabled Component Middleware

1 Introduction

1.1 Characteristics and Challenges of Enterprise DRE Systems

Enterprise distributed real-time and embedded (DRE) systems, such as shipboard computing environments [1], airborne command and control systems [2], and intelligence, surveillance and reconnaissance systems [3], are growing in

complexity and importance as more computing devices are networked together to help automate tasks previously done by human operators. These types of systems are characterized by stringent quality-of-service (QoS) requirements, such as low latency and jitter, expected in real-time and embedded systems, as well as high throughput, scalability, and reliability expected in enterprise distributed systems. In particular, these types of systems share the following characteristics:

- **Heterogeneity.** Enterprise DRE systems often run on a variety of computing platforms that are interconnected by different types of networking technologies with varying quality of service (QoS) properties. The efficiency and predictability of enterprise DRE systems built using different infrastructure components varies according to the type of computing platform and interconnection technology.
- **Networked embedded properties.** Enterprise DRE systems are often composed of multiple networked embedded subsystems. For example, a shipboard computing environment may have multiple *threat tracking subsystems* that colloborate with multiple *mission planner subsystems* (which make decisions on what actions to take depending on the threats being tracked) and multiple *effector subsystems* (which enact the decisions). There should be strong communication and colloboration between these networked embedded subsystems to achieve the critical functionalities of an enterprise DRE system.
- **Simultaneous support for multiple QoS properties.** Software controllers [4] are increasingly replacing mechanical and human control of enterprise DRE systems. These controllers must simultaneously support many challenging QoS constraints, including (1) *real-time requirements*, such as low latency and bounded jitter, (2) *availability requirements*, such as fault propagation/recovery across distribution boundaries, (3) *security requirements*, such as appropriate authentication and authorization, and (4) *physical requirements*, such as limited weight, power consumption, and memory footprint. For example, a distributed patient monitoring system requires predictable, reliable, and secure monitoring of personal health data that can be distributed in a timely manner to healthcare providers.
- **Large-scale system-of-systems operation.** The scale and complexity of enterprise DRE systems often makes it infeasible to deploy them in separate disconnected islands. The functionality of these systems must therefore be partitioned and distributed over a range of networks. For example, an urban bio-terrorist evacuation capability requires highly distributed functionality involving networks connecting command and control centers with bio-sensors that collect data from police, hospitals, and urban traffic management systems.
- **Dynamic operating conditions.** Operating conditions for enterprise DRE systems can change dynamically, resulting in the need for appropriate adaptation and resource management strategies for continued successful system

operation. In civilian contexts, for instance, power outages underscore the need to detect failures in a timely manner and adapt in real-time to maintain mission-critical power grid operations. In military contexts, likewise, a mission mode change or loss of functionality due to an attack in combat operations requires adaptation and possible resource reallocation to continue with mission-critical capabilities.

As evidenced from the discussion above, enterprise DRE systems have a range of QoS requirements that may vary at runtime due to planned [5] and unplanned [6] events. Examples of planned events include mission goal changes due to refined intelligence and planned task-completion exceeding mission parameters. Likewise, examples of unplanned events might include system runtime performance changes due to loss of resources, transient overload, and/or changes in algorithmic parameters (such as modifying an air threat tracking subsystem to have better coverage).

Due to the challenges of applying theoretically-grounded techniques for generating and adapting resource allocations to enterprise DRE systems, conventional development and quality assurance processes employ *ad hoc* techniques comprising heuristics and/or simple planning algorithms to solve specific resource management problems identified at design-time. Unfortunately, these techniques are poorly suited to handle dynamic events (such as equipment failure, mission changes, damage, or significant overload) that enterprise DRE systems encounter at runtime.

1.2 Addressing DRE System Challenges with Dynamic Resource Management

Approaches to resolving the challenges outlined in Section 1.1 have traditionally focused on conducting offline static resource-management analysis to determine how the system should respond to planned events and a limited set of unplanned events [7]. Such analysis is performed by complex analytical and simulation models of the systems, subjecting these models to workloads that represent various planned and unplanned events and determining how well the systems perform in the face of the emulated mission-level and environmental changes. As the size of the enterprise DRE systems and the environments in which they execute increases, however, offline static analysis of the system becomes intractable due to the explosion of possible state permutations and the limitations of analytical methods to scale seamlessly to cover large state-spaces [8].

Dynamic resource management (DRM) [9, 10, 11] is a promising paradigm for alleviating the limitations of static analysis and support different types of

applications running in enterprise DRE system environments - as well as to optimize and reconfigure the resources available in the system to meet the changing needs of applications at runtime. A key goal of DRM is to ensure that enterprise DRE systems can adapt dependably in response to dynamically changing conditions (*e.g., evolving mutli-mission priorities*) to ensure that computing and networking resources are best aligned to meet critical mission requirements. A key assumption in DRM technologies is that the levels of service in one dimension can be coordinated with and/or traded off against the levels of service in other dimensions to meet mission needs, *e.g.*, the security and dependability of message transmission may need to be traded off against latency and predictability.

Prior work [9, 10, 11] on DRM has focused on general techniques for QoS-enabled enterprise DRE systems or specialized techniques for niche real-time systems [12]. Our work in this paper augments this prior work by focusing on (1) providing a standards-based *multi-layer resource management* (MLRM) infrastructure for configurable DRM services and (2) developing and empirically validating theoretically-grounded technologies for applying these DRM services to mission-critical enterprise DRE systems. In particular, our approach to DRM described in this paper innovates across the following interrelated technology areas:

- **Task allocation and analysis techniques** that generate and validate a coordinated set of resource allocations to satisfy evolving mission functional and performance requirements.
- **QoS-enabled component middleware** that provides a configurable computing infrastructure capable of enforcing resource allocations, even in the presence of overload and failures, thereby enabling and sustaining effective enterprise DRE software operation.

The remainder of this paper describes how our MLRM architecture has demonstrated DRM capabilities in a *total ship computing environment* (TSCE), which is a grid of computers that manage many aspects of a ship's power, navigation, command and control, and tactical operations [1]. To make a TSCE an effective platform requires coordinated and standards-based DRM services that can support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization. The MLRM described in this paper was developed for the DARPA's *Adaptive and Reflective Middleware Systems* (ARMS) program (`dtsn.darpa.mil/ixodarpatech/ixo_FeatureDetail.asp?id=6`), which is applying DRM technologies to coordinate a grid of computers that manage and automate many aspects of Naval TSCEs. We describe and empirically evaluate how the ARMS MLRM manages the computing resources in TSCE dynamically and ensures proper execution of TSCE missions in response to mission mode changes and/or resource load changes and failures, as well as capability upgrades.

2 The ARMS Multi-layered Resource Management Middleware

Section 1 motivated the need for enterprise DRE systems to adapt dynamically to changes in mission goals, variations in resource availability, and capability upgrades. This section describes the component-based middleware used to implement the ARMS MLRM architecture, focusing on the design goals that guided our approach.

2.1 ARMS MLRM Design Goals

Providing effective DRM capabilities for enterprise DRE systems depends on several factors that span the domain- and solution-space. For example, solutions to domain-specific issues, such as adapting to mission mode changes, capability upgrades or resource failures, are impacted by the choice of technologies and platforms used in the solution space. Addressing the complex problem of DRM as a single unit, however, can become intractable due to the *tangling of concerns* across the domain- and solution-space. Hence, there is a need to address DRM problems at different levels of abstraction, yet maintain a coordination between these levels. These design goals motivate the ARMS MLRM framework described in this section, which we use to resolve the DRM challenges of TSCEs.

The goals of the ARMS MLRM design are to provide DRM solutions when missions change, resources fail or become available, failures occur due to battle-damage, or new capabilities are added to the system. Figure 1 illustrates the

		Mission Changes	Load Changes	Resource Changes	Capability Upgrades	Platform Upgrades
Domain Space	Application Adaptations	Alter behavior to specified mode changes	Graceful tolerance of in-spec variations	Mission critical capability preservation through specified failure and damage events	Typically not supported by individual applications	Typically not supported by individual applications
Solution Space	Allocation Adaptations	Mission performance allocations in response to evolving needs	Graceful tolerance of out-of-spec variations	Maximal capability restoration after arbitrary failure and damage events	Deployment of new mission capabilities into existing systems	Deployment of mission capabilities into heterogeneous environments
	Performance Adaptations	Mission performance optimizations in response to evolving needs	Graceful tolerance of out-of-spec variations	Capability preservation though arbitrary failure and damage events	Tuning of new mission capabilities into existing systems	Deployment of mission capabilities into heterogeneous environments

Fig. 1. Types of TSCE DRM Problems

types of DRM problems and operational context that we addressed to meet the needs of TSCEs:

• **Mission mode changes**, where the goal is to enable a much broader set of resource re-allocations beyond mode changes and behaviors typically provided by domain applications. Due to the changes in environmental conditions, the availability of hardware nodes in a TSCE vary. Moreover, the fraction of capacity available for each application component on a given node varies depending on the number, type, and workload of application components hosted in each node. Meeting these goals requires the ARMS MLRM to determine *at runtime* which components should actually run in response to mission mode changes. Moreover, the MLRM must tune application performance parameters dynamically using increasingly finer-grained precision, as opposed to a coarse-grained, discrete set of configurations.

• **Load changes**, where the goal is to tolerate *out-of-spec* variations gracefully. For example, a statically allocated TSCE responding to radar tracks may be provisioned to handle a maximum of no more than 400 tracks. A more effective TSCE design, however, should respond gracefully to overload situations, such as handling more than 400 tracks by reallocating and/or reprioritizing other components in accordance with mission importance parameters. The ARMS MLRM therefore ensures that available TSCE resources are allocated to the *currently most important* mission capabilities, allowing the TSCE to scale to even *out-of-spec* loads gracefully.

• **Resource changes**, where the goal is to restore *all* mission capabilities, even those that were not designed to tolerate specific failures. Through the automatic recovery from such failures, the ARMS MLRM maximizes TSCE resource utilization and performance, while simultaneously maximizing the value of mission capabilities. In addition, we wanted to expand mission capabilities as new resources become available due to requirement or environment changes.

• **Capability upgrades**, where the goal is to allow capabilities to be dynamically introduced into the system that were not planned initially. We wanted to capture the general behavior of the newly introduced artifacts so the ARMS MLRM can determine the resource requirements of these artifacts dynamically and make appropriate allocations. In particular, the ARMS MLRM should be able to make dynamic resource allocation decisions for the newly introduced artifacts, even when such capabilities were not part of the original system.

• **Platform upgrades**, where the goal is to support the *heterogeneous* environment in which long-lived TSCEs operate. For example, our TSCE is designed to support different versions of (1) distributed computing middleware, such as TAO/CIAO in C++, JacORB/OpenCCM in Java, and OrbRiver in Ada, (2) general-purpose POSIX-based operating systems, such as Linux and Solaris, (3) real-time operating systems, such as VxWorks and LynxOS, (4) hardware chipsets, such as x86, PowerPC, and SPARC processors, (5) a wide

range of high-speed wired interconnects, such as Gigabit Ethernet and VME backplanes, and (6) different transport protocols, such as TCP/IP and SCTP.

The ARMS MLRM services are implemented as a set of common and domain-specific middleware services (described in Section 2.3) that communicate using standards-based middleware (described in Section 2.2) that uses a wide range of standard/proprietary operating systems, networks, programming languages, and hardware.

2.2 The ARMS MLRM QoS-enabled Middleware Infrastructure

To simplify development and enhance reusability, the ARMS MLRM is based on a QoS-enabled component middleware infrastructure comprised of the *Component-Integrated ACE ORB* (CIAO) [13] and the *Deployment And Configuration Engine* (DAnCE) [14]. CIAO and DAnCE implement the *Real-time CORBA Component Model* (RT-CCM) [15], which combines standard Lightweight CCM [16] mechanisms (such as codified specifications for specifying, implementing, packaging, assembling, and deploying components) and standard Real-time CORBA [17] mechanisms (such as thread pools, priority preservation policies, and explicit binding capabilities) to simplify and automate the (re)deployment and (re)configuration of application components in enterprise DRE systems. CIAO and DAnCE use patterns [18, 19] to achieve highly portable and standards-compliant component middleware and reflective middleware techniques [20, 1] to support key QoS aspects for enterprise DRE systems.

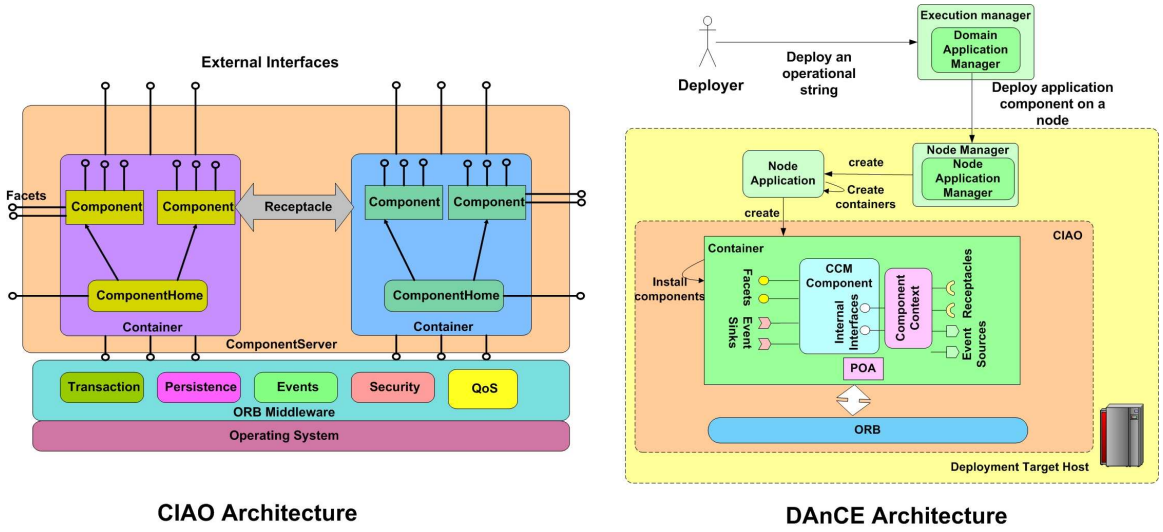

Fig. 2. **CIAO and DAnCE Architecture**

As shown in Figure 2, *components* in CIAO are implementation entities that collaborate with each other 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 can be used to exchange typed events with one or more components. *Containers* in CCM provide a run-time environment for one or more components that manages various pre-defined hooks and strategies (such as persistence, event notification, transaction, and security) used by the component(s). Each container is responsible for (1) initializing instances of the component types it manages and (2) connecting them to other components and common middleware services. There are two categories of components in CIAO: (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).

Component assemblies in CIAO are deployed and configured via DAnCE, which is an implementation of standard OMG *Deployment and Configuration* (D&C) specification [21] shown in Figure 2. DAnCE 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 assembly descriptors and deployment plans. DAnCE's runtime framework parses these descriptors and plans to extract connection and deployment information and then automatically deploys the assemblies onto the CIAO component middleware platform and establishes the connections between component ports.

2.3 The Component-based ARMS MLRM Design

The ARMS MLRM architecture integrates resource management and control algorithms based on the standards-based CIAO and DAnCE RT-CCM infrastructure described in Section 2.2 to achieve the enterprise DRE system challenges described in Section 1.2 and the ARMS MLRM design goals described in Section 2.1. These design goals, combined with the complexity of the TSCE domain, led us to model the DRM solution space as a layered architecture comprising components at different levels of abstraction. Figure 3 depicts the layers in the ARMS MLRM, which has hundreds of different types and instances of infrastructure components written in ~300,000 lines of C++ code and residing in ~750 files developed by different teams at different locations. The remainder of this section describes the structure and dynamics of the ARMS MLRM.

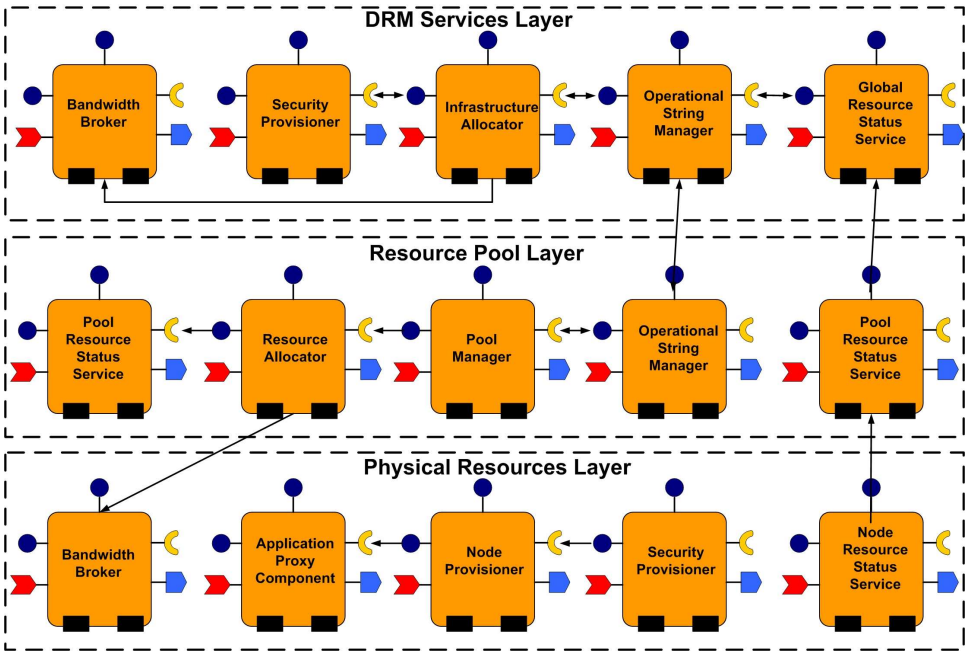

Fig. 3. Components in the ARMS MLRM

2.3.1 ARMS MLRM Structure

We first describe the key goals of each layer in Figure 3 and explain how the RT-CCM-based components within these layers help address the goals described in Section 2.1.

• The **DRM Services layer** is responsible for satisfying TSCE missions, such as allocating system computing and networking resources to respond to radar tracks and planning necessary actions or events. The goal of this layer is to maximize the mission coverage and reliability in response to the resource management problems shown in Figure 1. The DRM Services layer receives explicit resource management requests from applications, along with command and policy inputs. It decomposes mission requests on the TSCE into a coordinated set of software allocation requests on resource pools within the *Resource Pool layer*. The DRM Services layer also monitors and coordinates the execution of *operational strings*, which are sequences of components that capture the partial order and workflow of a set of executing software capabilities.

Key components in this layer include the (1) *Infrastructure Allocator*, which determines the resource pool(s) where a package's operational string(s) are deployed, (2) *Operational String Manager*, which coordinates the proper deployment of operational strings across resource pools. The DRM services layer can manipulate groups of operational strings by (1) deploying, stopping, and migrating them depending on their mode of operation, (2) deploying them in different configurations to support different modes, and (3) deploying them in different orders depending on mode transitions within the TSCE. The

ARMS MLRM manages operational strings based on their priority, *i.e.*, it (re)allocates resources for the operational strings satisfying higher priority goals. The MLRM may therefore stop or migrate lower priority operation strings if resources become scare so that higher priority operational strings remain operational.

• The **Resource Pool layer** is an abstraction for a set of computer nodes managed by a *Pool Manager* component. The *Pool Manager*, in turn, interacts with a *Resource Allocator* component to run algorithms that deploy application components to various nodes within the resource pool. The goal of this layer is to handle the two complementary capabilities:

(1) **Proactively allocate resources so that QoS requirements for all critical operational strings are satisfied within a single pool**. The DRM services layer must ensure that resource allocations satisfy QoS needs of operational strings in a TSCE. The ARMS MLRM employs importance-ordered, uni-dimensional bin-packing [22] of worst-case application CPU resources to allocate operational strings to computing nodes. It also employs a network provisioning algorithm [23] that allocates network bandwidth based on operation string interactions. The experiments in Section 3.2 demonstrate how unique infrastructure resource allocations were generated in response to different sequences of mission deployment requests (an example of the *mission mode change* problem in Figure 1). When appropriate metadata is provided, ARMS MLRM also performs end-to-end response time schedulabililty analysis [24] to verify that allocations can satisfy real-time deadlines of operational strings.

(2) **Reactively re-allocate resources or operational modes to tune deployed operational strings to current mission priorities.** The ARMS MLRM defines *Condition Monitors*, *Determinators*, and *Response Coordinators* components [25,26] to detect, verify, and recover from classes of execution and performance problems in deployed operational strings. These components deploy controllers for managing operational string overload (an example of a *load change* problem from Figure 1). The experiments in Section 3.3 and Section 3.4 demonstrate how these services help to (1) restore maximum capability after a resource pool failure (an example of a *resource change* problem from Figure 1), and (2) manage operational string overload to allow critical applications to continue operating.

• The **Physical Resources layer** deals with the specific instances of resources in the TSCE. The goal of this layer is to configure physical resources in accordance with dynamically-generated allocations to support the QoS needs of a mission. The ARMS MLRM configures OS process priorities and scheduling classes of deployed components across a variety of operating systems (*e.g.*, Linux, Solaris, and VxWorks) in a manner that preserves execution precedence

specified in the dynamically generated allocation.

2.3.2 ARMS MLRM Dynamics

The ARMS MLRM acts in response to incidences of resource management problems of the types shown in Figure 1. To illustrate the dynamic behavior of the ARMS MLRM, we next describe the two examples shown in Figure 4: a *proactive workflow* in response to a mission mode change request and a *reactive workflow* in response to a load change in an operational string [27]. The experimental results for both these workflows are described in Section 3.

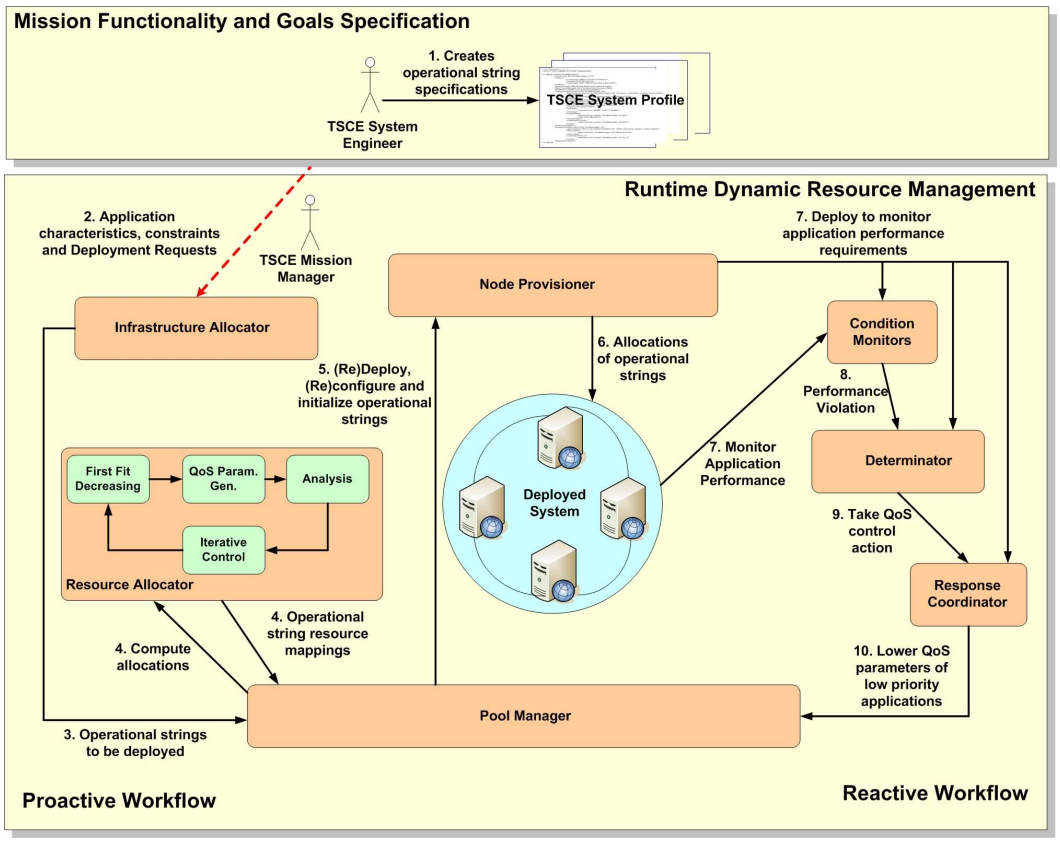

Fig. 4. **MLRM Workflow**

The sequence of activities happening in a typical TSCE environment as the system goes through proactive and reactive workflows are shown in Figure 4 and described below:

(1) TSCE mission management services selectively deploy and suspend mission software packages in response to tactical needs. A TSCE mission engineer designs the sets of operational strings to be deployed based on the current mission and also captures the system profile of all operational strings.

(2) Deploying a new mission package begins with a deployment request to the DRM Service layer's *Infrastructure Allocator*, which can split operational strings across resource to balance resource load or to satisfy various constraints (*e.g.*, place primary and replicas in different geographic locations).
(3) In accordance with each operational string's start-up order constraints, the DRM services layer's *Operational String Manager* directs the *Pool Manager* components in each resource pool to deploy one or more substrings.
(4) If the deployment policy is dynamic, each *Pool Manager* uses a *Resource Allocator* component in the *Resource Pool layer* to assign applications to hosts, generate OS priorities and scheduling classes, and coordinate with the *Bandwidth Broker* component in the *Resource Pool layer* to reserve network resources. The *Resource Allocator* obtains the existing resource allocations from the pool's *Resource Status Service* [28] and analyzes schedulabililty of the newly generated allocation to determine whether critical path deadlines can be satisfied.
(5) The *Pool Manager* directs the *Node Provisioner* components on each node within the *Resource Pool layer* to implement those allocation directives.
(6) The allocation directives done by the *Node Provisioner* include (1) launching new applications and configuring their QoS parameters (*e.g.*, OS priority, scheduling class, and Diffserv codepoint tagging) and (2) reconfiguring QoS parameters of previously deployed applications that are shared by the newly deployed operational string. After all pool allocations are complete, the *Operational String Manager* initiates the final start-up of the new operational string(s) by invoking their application-level `start()` methods in accordance with their ordering constraints.
(7) Deployed operational strings may declare various performance requirements in their metadata, *e.g.*, a minimum average CPU utilization for each application or an end-to-end deadline. The ARMS MLRM deploys *Condition Monitors* in the *Resource Pool layer* to detect whether such performance requirements are being satisfied
(8) When a *Condition Monitor* detects a performance requirement violation, it notifies a *Determinator* in the *Resource Pool layer*, which considers the violation in the broader context of mission importance, resource availability, and policy directives to determine whether the problem must be addressed and on what timeline.
(9) When the *Determinator* decides that a problem must be addressed it notifies a *Response Coordinator* in the *Resource Pool layer* to initiate recovery actions. We developed a coordinated set of *Condition Monitors*, *Determinators*, and *Response Coordinators* components to support the string overload problem described in the experiments discussed in Section 3.4. In this case, the *Condition Monitor* detected that (after an increase in external threats, which exceeded the design threshold) a critical application, in a high importance operational string, was being starved by a less

important competing operational string.

(10) The *Response Coordinator* addresses starvation by lowering the QoS parameters (*i.e.*, OS priority and scheduling class) of the competing applications using resource allocation services provided by the *Pool Manager*, *Resource Allocator*, and *Node Provisioner* components in the local resource pool. By reconfiguring the TSCE using these components, the ARMS MLRM helped ensure that the maximum number of mission-critical operational strings were scaled to meet out-of-spec load changes within available resource constraints.

3 Empirically Evaluating the ARMS MLRM

This section describes the design and results of experiments we conducted to empirically evaluate the ARMS MLRM architecture described in Section 2. We focus on the MLRM's capabilities for (1) dynamically managing computing and network resources to meet changing mission requirements and (2) configuring various QoS mechanisms within the middleware to satisfy QoS and operational requirements.

3.1 Hardware/Software Testbed and Evaluation Criteria

The experiments used up to six Sun SPARC Solaris computers, with two of the six being Sunfire V1280 servers with 12 multiprocessors and other four being desktop uniprocessors. All nodes ran the Solaris 8 operating system, which supports kernel-level multi-threading and symmetric multiprocessing. All nodes were connected over a 100 Mbps LAN and used version 0.4.1 of CIAO and DAnCE as the QoS-enabled component middleware infrastructure. The benchmarks ran in the POSIX real-time thread scheduling class [29] to enhance the consistency of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

Figure 5 shows the TSCE scenario that guided the experiments. In this scenario many sensors (ED) colloborate with planning processes (PLAN) and effectors (EFF) to detect different operating conditions and make adaptive and effective decisions to counteract harmful and adverse conditions. The sensors, planning processes, and effectors were implemented as application components using CIAO; groups of application components are connected in component assemblies to form operational strings. Each operational string in Figure 5 corresponds to a different set of TSCE capabilities, such as anti-cruise missile ship-self defense, mine avoidance, or crew entertainment. The MLRM therefore deploys different sets of operational strings depending on

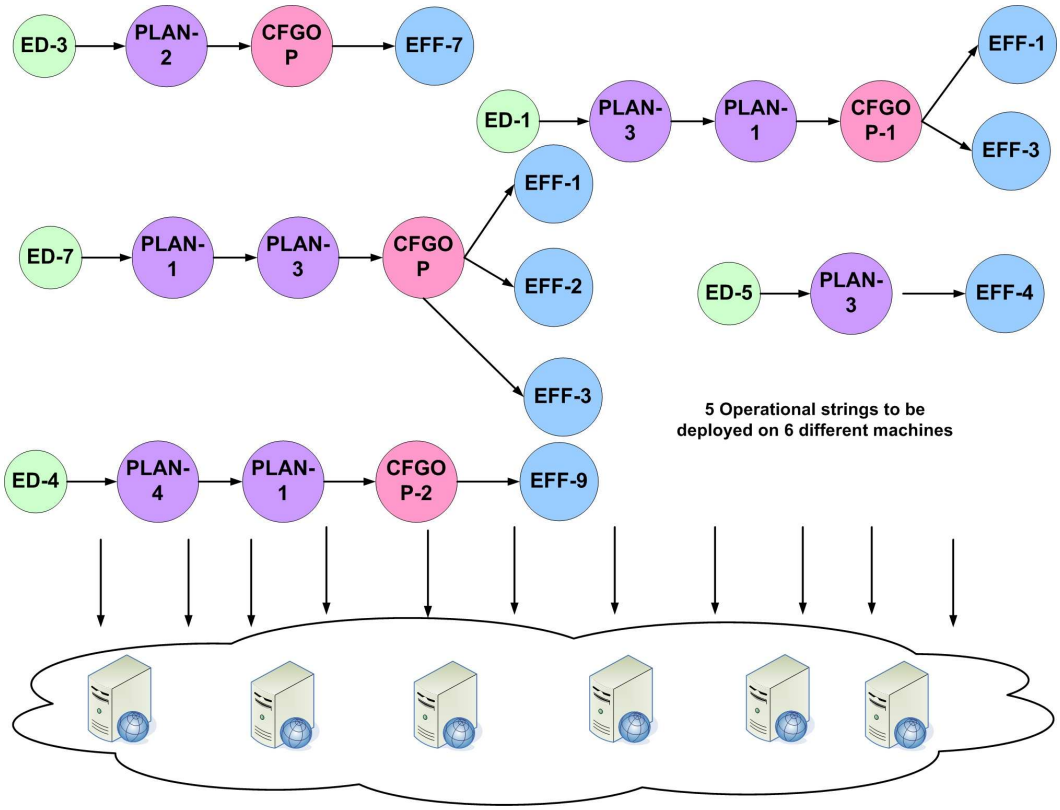

Fig. 5. **Operational Strings Deployed in ARMS MLRM Experiments**

the different contexts and mission modes. For example, a particular combination of sensors, planning processes, and effectors implemented as application components and connected together can work in a collaborative fashion to detect, plan, and implement a corrective action to correspond to an sensed event, such as sensing of unidentified airborne object and deciding what types of counter-measures to enact. These operational strings are managed by the ARMS MLRM as described in Section 2.3.

Our three main goals in conducting experiments on the TSCE scenario shown in Figure 5 were to:

(1) Evaluate the capability of the ARMS MLRM to deploy the operational strings in different configurations and in different orders so that the applications can be mapped or hosted into many different combinations of physical processors, thereby ensuring that MLRM can dynamically manage the available resources by redeploying operational strings to different hardware nodes as nodes become unavailable due to resource limitations or hardware failures. The goal was to demonstrate that at least 2 different dynamic allocations can be achieved based on prevailing conditions.
(2) Quantify the capability of MLRM to recover from data center failures and redeploy primary operational strings to available data centers in a timely manner. The goal was to recover in less than 1/2 the time of 4

minutes it takes to recover in a manually managed TSCE.

(3) Demonstrate the capability of MLRM to respond to an increasing number of threats by ensuring that higher priority and more important applications always operate, so that the utilization of resources are under control and always available for providing key TSCE capabilities.

The remainder of this section describes the experiments we conducted and evaluates how well the ARMS MLRM meets the stated goals.

3.2 MLRM Capability to Deploy Infinite Set of Configurations

We first present empirical results that demonstrate ARMS MLRM's ability to generate different deployment configurations of the same set of operational strings, depending on different input ordering of strings, different resource availabilities of the resources, and different resource requirements of the operational strings. We used the TSCE scenario described in Figure 5 to identify five operational strings that map to five different TSCE functionalities, including ship self defense, call for fire, and undersea warfare and others. Each operational string involved a subset of the applications shown and contain a *critical path*, *i.e.*, start-to-finish processing flow over which an end-to-end deadline must be met.

The goal of this experiment was to demonstrate that we could create a variety of different configurations of the same set of five operational strings by mapping those operational strings onto up to six different hardware nodes in response to (1) changes in the resource availabilities of the nodes, (2) changes in the events in the TSCE scenario that causes new strings to be deployed beyond the strings that are already deployed, (3) changes in the configurations of the strings in response to the mode changes in the TSCE scenario, and (4) changes in the size of the operational strings. The five operations strings comprise of 8, 9, 16, 17, and 20 application components for a total of 70 application components in the deployed system. Application components are written using the CIAO RT-CCM middleware and deployed and configured using DAnCE.

The results in Figure 6 demonstrate that the ARMS MLRM was able to generate six different configurations for those five operational strings deployed on the six different hardware nodes. The five dynamic deployments differ from each other by the different order in which various strings are deployed to satisfy different mission needs. It can be observed, however, that in all cases dynamic deployment – based on runtime use of bin-packing algorithms [30, 31] – leads to a fairly balanced workload across the processors. These results indicate that

- The ARMS MLRM allocation algorithms have the capability to generate

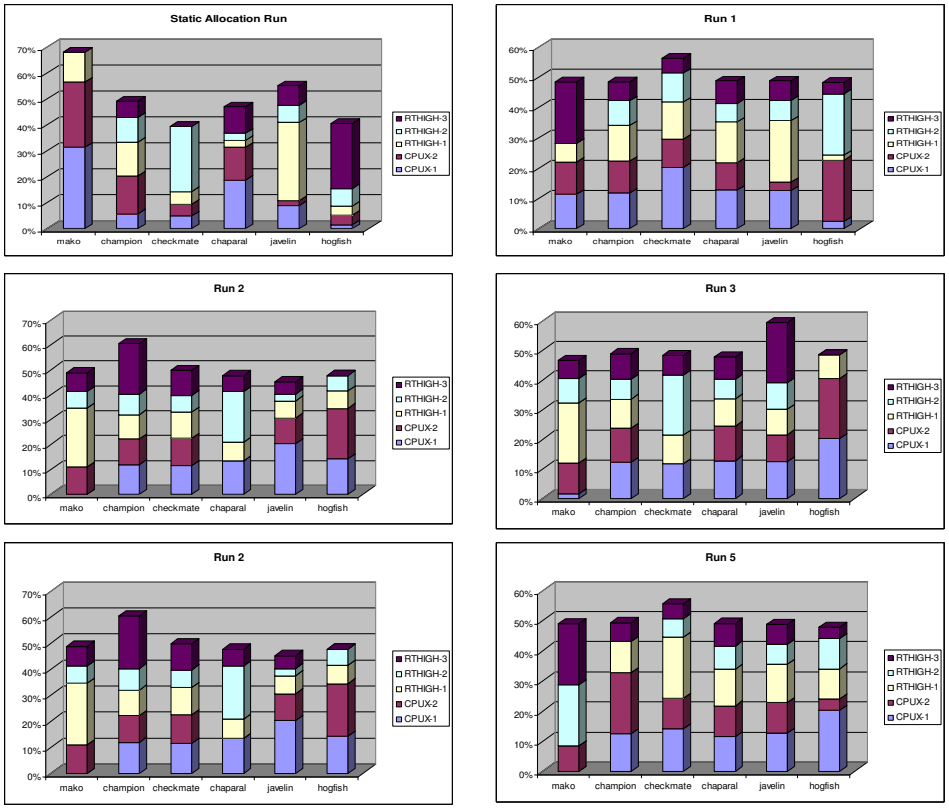

Fig. 6. **Different Configurations of Operational Strings in Six Nodes**

different sets of deployment configurations for the operational strings over the same set of hardware nodes. For static allocation the highest processor utilization was ∼65 percent and the lowest was ∼38 percent. In contrast, all dynamic allocations resulted in highest processor loading of less than 60 percent and no processor is loaded less than 40 percent. Keeping maximum utilization is as low as possible is a DRM goal that ensures adequate safety margins and system schedulabililty.

- With the help of the standards-based CIAO and DAnCE RT-CCM middleware that deploys and migrates operational strings, the ARMS MLRM can quickly generate different sets of allocations for operational strings depending upon resource availabilities and hardware failures. As a result, critical TSCE mission functionalities need not be compromised in the face of hazardous events that can trigger resource fluctuations and failures.

3.3 MLRM Capability to Recover from Failures

Section 2.3 described how the ARMS MLRM manages different sets of resources within a single pool and how the resources within a pool can be utilized efficiently to deploy operational strings by mapping them onto different nodes within a resource pool. To support active replication, the MLRM de-

ploys replica operational strings in different resource pools so that mission mode functionality can be available in a dependable manner even in the face of harmful events. To minimize the utilization of resources, however, not all operational strings are duplicated and only certain mission-critical operational strings are replicated and deployed under different resource pools.

Figure 7 shows how operational strings consisting of a combination of sensors and effectors in our TSCE are deployed together with their replicas across two different resource pools. The two resource pools are *Pool-1.A* and *Pool-1.B*

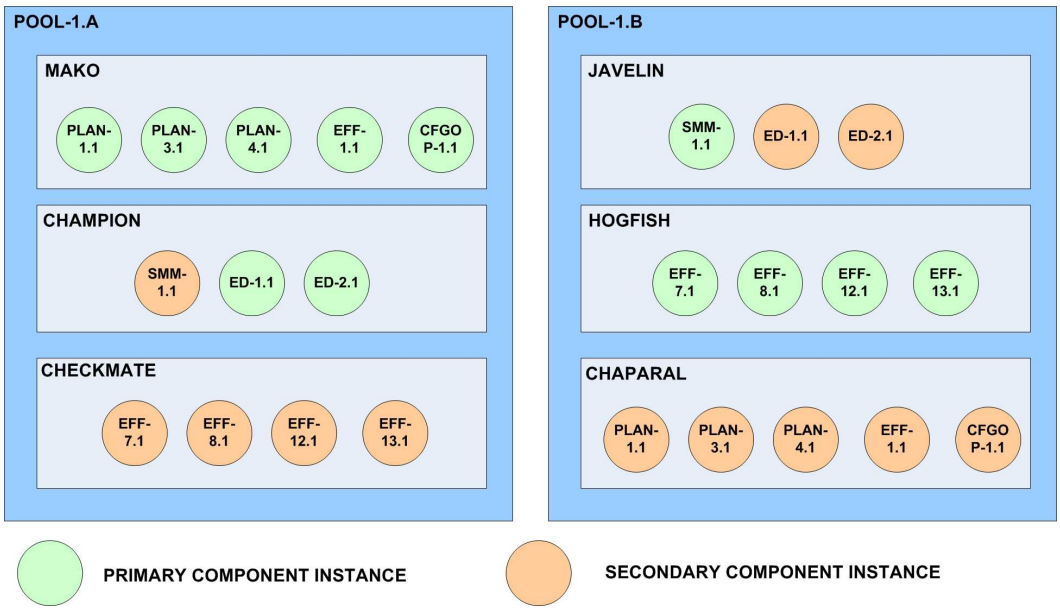

Fig. 7. **Operational Strings and Their Replicas Deployed in Different Resource Pools**

and each of the resource pools have three nodes each. The primary and the secondary replicas for a particular operational string are distributed across the two resource pools to ensure active replication. For example, the operational string deployed in the nodes *Mako* and *Champion* of the resource pool *Pool-1.A* have their replicas deployed in the nodes *Chaparal* and *Javelin* of the resource pool *Pool-1.B*. Below, we present empirical results that measure how fast the ARMS MLRM can react to pool failures to reconstitute the primary operational strings from the failed pools onto the pools that are safe, thereby ensuring that critical mission functionality is available throughout the lifecycle of the TSCE.

We used the TSCE scenario described in Figure 7 to identify the steps the ARMS MLRM must take to reconstitute the primary replicas from the *Pool-1.B* resource pool onto the *Pool-1.A* resource pool in the case of the total failure of the *Pool-1.B* resource pool. We focused on the time the ARMS MLRM needed to (1) detect the pool failure and decide what needs to be redeployed, (2) redeploy the strings in the available pools and their resources, (3) allow

the OS to spawn a processing environment for the operational strings, (4) give appropriate trigger signals to the strings so that they can be part of the global mission functionalities by colloborating with other strings, and (5) promote the new primaries and secondaries in the new deployment configurations.

The results in Figure 8 show that ARMS MLRM takes ~15 seconds to complete parts 1 and 2, which are pool failure detection and redeployment, respectively. The vast majority of this time is taken up in pool failure detection.

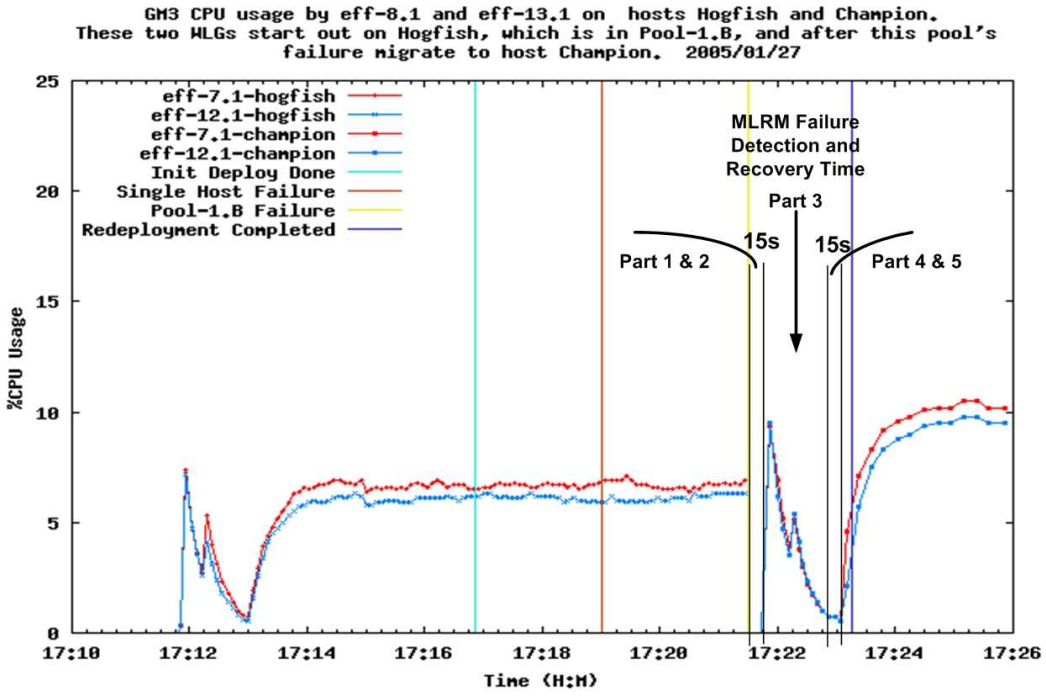

Fig. 8. **Redeployment of Effectors from Failed Pool**

Conservative timeout values were used for failure detection to minimize false alarms. With a more aggressive use of timeout, either with capabilities native to a compute node or with custom hardware, can reduce this number to well below a second. The time taken by part 2 (redeployment decisions) are well below a second. A relatively large gap is noted for part 3 (time to spawn new processes to host application components), which is handled by DAnCE and is outside the direct control of ARMS MLRM. Finally, we note another 15 second lag for parts 4 and 5 (initialization and triggering) of the redeployment process. The time to recover is therefore well below 1/2 the 4 minutes it took to recover a manually managed system.

Section 2.3 described how the ARMS MLRM manages different sets of resources within a single node in a pool and provides the capability to maintain the operation of critical mission capabilities by gracefully responding to overload conditions, such as handling more tracks and threats by offloading, reallocating, or reprioritizing other *not-so-important* operational strings operating in a node. Below, we present empirical results that show (1) how the ARMS MLRM can handle increased loads in a node to handle additional threats and tracks and (2) how the MLRM can do this gracefully by offloading less important operational strings from a node. The goal is to ensure that critical mission capabilities meet their QoS requirements in the face of increasing workloads.

Figure 9 shows a scenario in the TSCE where three nodes are used to deploy two operational strings that provide *general tracking* and *air threat tracking* capabilities. We measured the CPU utilization in the *Mako* node, where parts

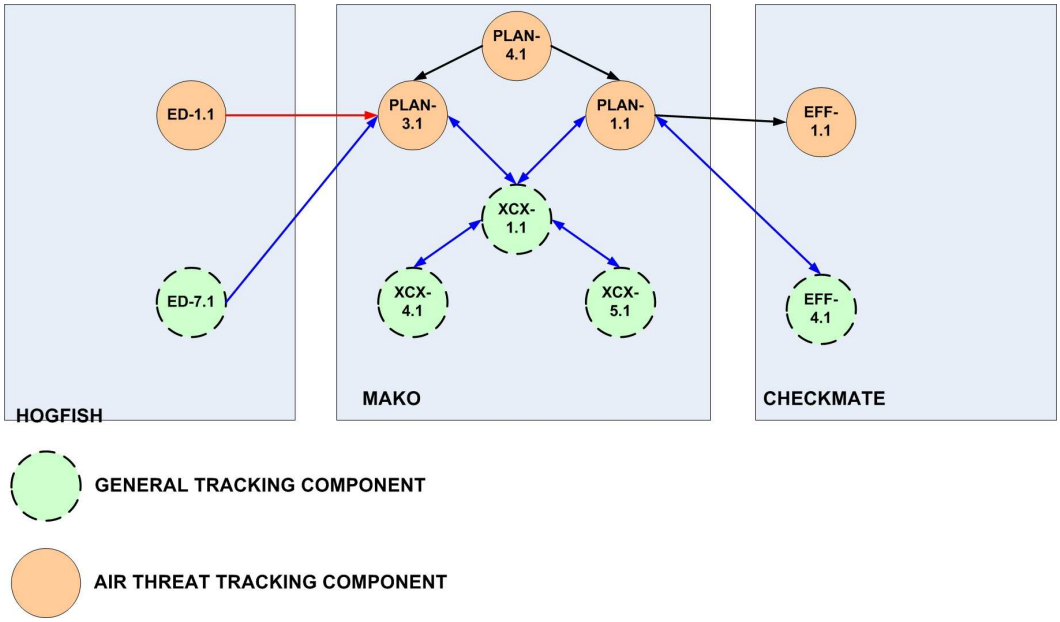

Fig. 9. **Different Threat Tracking Components in the ARMS MLRM**

of both the operational strings are deployed and launched. We also measured the *Mako's* CPU utilization as air threats started to increase, prompting the *ED-1.1* sensor to send reactive events and messages to *PLAN-3.1* planner on *Mako*.

The planner makes decisions and uses the effector *EFF-1.1* in the *Checkmate* node to take evasive actions. In this scenario, *Mako*'s CPU utilization went up sporadically as it processed a series of air threats. In the face of the increasing air threats, however, the critical operational functionalities provided by *Mako* (such as the planning process provided by the *PLAN-3.1* planner) need to

operate correctly and efficiently. To evaluate the benefits of DRM, we therefore compared the results adding and removing ARMS MLRM capabilities in the TSCE to highlight its capabilities.

Figure 10 shows the results of an experiment where the ARMS MLRM capabilities were disabled. Here we observe that as processing load for Plan-1.1

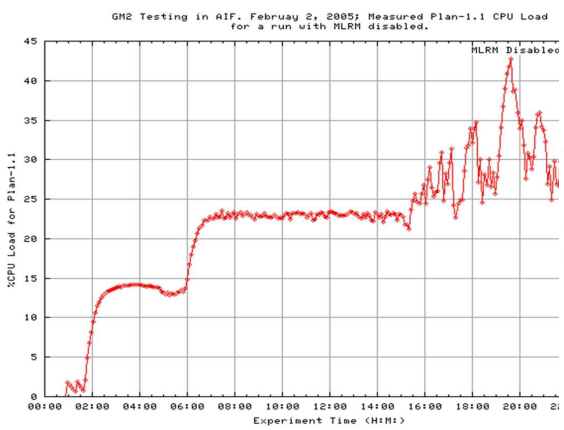

Fig. 10. **Handling Threats with MLRM Disabled**

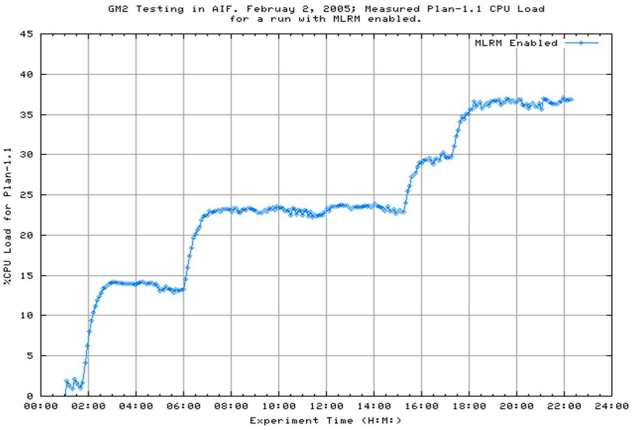

Fig. 11. **Handling Threats with MLRM Enabled**

increases beyond a certain point the system starts to become unstable with actual amount of resource consumed fluctuating significantly. In contrast, Figure 11 shows the results of enabling the ARMS MLRM DRM capabilities, where even as load for Plan-1.1 increases the MLRM makes the necessary dynamic adjustments to deliver steady resource availability for this application component.

In summary, the experimental evaluations in this section validated that the ARMS MLRM met its goals, *i.e.*:

- Multiple dynamic allocations were achieved - five were demonstrated.
- Recovery from a data center failure is achieved in well under 1/2 the time it takes for manual recovery
- Under increased load MLRM makes the necessary resource allocation adjustments to permit steady state increase in resource availability to critical applications without any instability.
- MLRM permits a range of automated, dynamic management of resources that greatly extend operational flexibility of the system, without requiring human intervention.

4 Related Work

A significant amount of research on dynamic resource management (DRM) appears in the literature, spanning several orthogonal dimensions. For example, DRM capabilities have been applied to specific layers, such as the application layer [32] or network layer [23]. DRM capabilities have also been applied either in standalone [33] or distributed [34] deployments. In this section we summarize related DRE research along the two dimensions.

Application-driven distributed DRM was described in [32], where the authors leverage the reconfigurability of their parallel applications to dynamically change the number of executing tasks and allocate resources based on changes in resource availability. This work deals with systems in which the arrival of jobs is unpredictable and highly variable, which is similar to the TSCE scenarios that are the focus of this paper. This work, however, is application-specific and provides a point solution that is not based on standard middleware platforms. In contrast, our ARMS MLRM work provides standards-based reusable middleware that are designed to support a broad class of enterprise DRE systems that can benefit from DRM.

The work described by [35] presents an integrated architecture for automatic configuration of component-based applications and DRM for distributed heterogeneous environments. The authors, however, do not describe how end-to-end QoS capabilities are maintained for mission-critical applications. In contrast, our work provides the results of empirical benchmarks that evaluate how well our MLRM supports end-to-end QoS.

Early work on middleware-based DRM solutions in the context of DRE shipboard computing systems is described in [34], where the authors describe the DeSiDeRaTa middleware that provides DRM for QoS-sensitive applications. This work and related efforts in [36] describe the characteristics necessary for DRM and provide the conceptual foundations for much of the ARMS MLRM efforts. Our work on MLRM extends this earlier work to support more finer grained DRE capabilities that ties mission needs to the management of physical resources, while also allowing for capability upgrades.

Network layer DRM capabilities conducted in the DARPA ARMS program are described in [23]. The network layer DRM solution in that paper focus on realizing DRM capabilities for one layer (the network layer) and for a specific resource (bandwidth). The MLRM approach described in this paper extends the network layer DRM capabilities to manage multiple resources, including CPU and network bandwidth, thereby providing a more comprehensive multi-layered and end-to-end solution.

5 Concluding Remarks

This paper described a standards-based, multi-layered resource management (MLRM) architecture developed in the DARPA ARMS program to support dynamic resource management (DRM) in enterprise distributed real-time and embedded (DRE) systems. The ARMS MLRM is designed to enable enterprise DRE systems to adapt to dynamically changing conditions (*e.g.*, during a tactical engagement) for the purpose of always utilizing the available computers and networks to the highest degree possible in support of mission needs under various operating conditions. The lessons learned while developing MLRM and applying it to a total ship computing environment (TSCE) include:

- The ARMS MLRM showed that DRM can be done using standards-based middleware technologies and can (1) handle dynamic resource allocations using many different configurations across a varied set of resource capacities, (2) provide reconfiguration and redeployment capabilities under the face of node and pool failures so that continuous availability can be achieved for critical functionalities and (3) provide QoS for critical operational strings even in the conditions of overload and resource constrained environments.
- Enterprise DRE systems that are provisioned statically require a great deal of engineering effort to create allocations, so changes are avoided as much as possible. The DRM capabilities provided by ARMS MLRM help alleviate much of this inflexibility and consistently achieves well-balanced allocation under varying conditions. In particular, the ARMS MLRM showed that the performance of its DRM services were better with MLRM enabled than with MLRM disabled.
- Although the ARMS MLRM provided an effective software infrastructure for providing DRM the coordination between many software architecture components is hard. We are therefore exploring model-driven tools [37] to develop a software infrastructure consisting of components aligned with standards-based middleware so that MLRM functions and workflow can be performed more effectively and productively.

At this phase of its development, MLRM has focused primarily on the functional architecture and software infrastructure needed to demonstrate the capabilities and benefits of dynamic resource management. In future work, more focus will be devoted to developing sophisticated resource management algorithms and a more flexible framework [38] that can allocate and control many DRM aspects. We are also developing an efficient MLRM testing methodology and infrastructure [39] so that integration and coordination problems can be easily identified, thereby enhancing the development and validation of enterprise DRE systems.

Acknowledgments

The experiments reported in this paper were conducted in conjunction with other colleagues on the DARPA ARMS program, including Stephen Cruikshank, Gary Duzan, Ed Mulholland, David Fleeman, Bala Natarajan, Rick Schantz, Doug Stuart, and Lonnie Welch.

References

[1] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, L. DiPalma, Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems, CrossTalk.

[2] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications, in: Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), IEEE, 2001, pp. 625–634.

[3] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, 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, 2004.

[4] K. Ogata, Modern Control Engineering, Prentice Hall, Englewood Cliffs, NJ, 1997.

[5] B. Li, K. Nahrstedt, A Control-based Middleware Framework for QoS Adaptations, IEEE Journal on Selected Areas in Communications 17 (9) (1999) 1632–1650.

[6] T. Abdelzaher, K. Shin, N. Bhatti, User-Level QoS-Adaptive Resource Management in Server End-Systems, IEEE Transactions on Computers 52 (5).

[7] L. Sha, R. Rajkumar, S. S. Sathaye, Generalized Rate Monotonic Scheduling Theory: A Framework for Developing Real-time Systems, Proceedings of the IEEE 82 (1).

[8] S. S, M. Krishnan, M. D. Smith, Using path profiles to predict http requests, in: Proceedings of the Seventh International World Wide Web Conference (WWW 98), Brisbane, Australia, 1998.

[9] L. Welch, B. Shirazi, B. Ravindran, C. Bruggeman, DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable, Real-Time Systems, in: Proceedings of the 15th IFAC Workshop on Distributed Computer Control Systems (DCCS'98), 1998.

[10] G. Nutt, S. Brandt, A. Griff, S. Siewert, T. Berk, M. Humphrey, Dynamically Negotiated Resource Management for Data Intensive Application Suites, in: IEEE Transactions on Knowledge and Data Engineering, 2000.

[11] J. Hansen, J. Lehoczky, R. Rajkumar, Optimization of Quality of Service in Dynamic Systems, in: Proceedings of the 9th International Workshop on Parallel and Distributed Real-Time Systems, 2001.

[12] S. Ghosh, R. Rajkumar, J. Hansen, J. Lehoczky, Adaptive QoS Optimizations for Radar Tracking, in: Proceedings of the 10 International Conference on Real-time and Embedded Computing Systems (RTCSA'04), 2004.

[13] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, C. D. Gill, QoS-enabled Middleware, in: Q. Mahmoud (Ed.), Middleware for Communications, Wiley and Sons, New York, 2003, pp. 131–162.

[14] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, A. Gokhale, DAnCE: A QoS-enabled Component Deployment and Conguration Engine, in: Proceedings of the 3rd Working Conference on Component Deployment, Grenoble, France, 2005.

[15] N. Wang, C. Gill, Improving Real-Time System Configuration via a QoS-aware CORBA Component Model, in: Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003, HICSS, Honolulu, HW, 2003.

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

[17] Object Management Group, Real-time CORBA Specification, OMG Document formal/02-08-02 Edition (Aug. 2002).

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

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

[20] N. Wang, C. Gill, D. C. Schmidt, V. Subramonian, Configuring Real-time Aspects in Component Middleware, in: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus, 2004.

[21] Object Management Group, Deployment and Configuration Adopted Submission, OMG Document ptc/03-07-08 Edition (Jul. 2003).

[22] J. W. S. Liu, Real-Time Systems, Prentice Hall, New Jersey, 2000.

[23] B. Dasarathy, S. Gadgil, R. Vaidhyanathan, K. Parmeswaran, B. Coan, M. Conarty, V. Bhanot, Network QoS Assurance in a Multi-Layer Adaptive

Resource Management Scheme for Mission-Critical Applications using the CORBA Middleware Framework, in: Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS), IEEE, San Francisco, CA, 2005.

[24] J. Sun, Fixed-priority end-to-end scheduling in distribured real-time systems, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champagne (1997).

[25] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, D. Bakken, Quo's Runtime Support for Quality of Service in Distributed Objects, in: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), IFIP, The Lake District, England, 1998.

[26] P. Stadzisz, J. Simao, M. Quinaia, A Pattern System to Supervisory Control of Automated Manufacturing System, in: Proceedings of the Third Latin American Conference on Pattern Languages of Programming, Porto de Galinhas, Brazil, 2003.

[27] Joseph K. Cross and Patrick Lardieri, Proactive and Reactive Resource Reallocation in DoD DRE Systems, in: Proceedings of the OOPSLA 2001 Workshop "Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems", 2001.

[28] J. Zinky, J. Loyall, R. Shapiro, Runtime Performance Modeling and Measurement of Adaptive Distributed Object Applications, in: Proceedings of the International Symposium on Distributed Objects and Applications (DOA'2002), Irvine, CA, 2002.

[29] Khanna, S., *et al.*, Realtime Scheduling in SunOS 5.0, in: Proceedings of the USENIX Winter Conference, USENIX Association, 1992, pp. 375–390.

[30] C. Kenyon, Best-fit bin-packing with random order, in: Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, 1996.

[31] B. S. Baker, A New Proof for the First-Fit Decreasing Bin-Packing Algorithm, in: Journal of Algorithms, 1985.

[32] J. E. Moreira, V. K. Naik, Dynamic resource management on distributed systems using reconfigurable applications, IBM Journal of Research and Development 41 (3) (1997) 303–330.
URL `citeseer.ist.psu.edu/moreira97dynamic.html`

[33] G. Banga, P. Druschel, J. Mogul, Resource Containers: A New Facility for Resource management in Server Systems, in: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI 99), 1999.

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

[35] F. Kon, T. Yamane, C. Hess, R. Campbell, M. D. Mickunas, Dynamic Resource Management and Automatic Configuration of Distributed Component Systems, in: Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), San Antonio, Texas, 2001, pp. 15–30.
URL citeseer.ist.psu.edu/341630.html

[36] B. Shirazi, L. R. Welch, B. Ravindran, C. Cavanaugh, B. Yanamula, R. Brucks, E. nam Huh, Dynbench: A dynamic benchmark suite for distributed real-time systems, in: IPPS/SPDP Workshops, 1999, pp. 1335–1349.
URL citeseer.ist.psu.edu/article/shirazi99dynbench.html

[37] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, 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), IEEE, San Francisco, CA, 2005.

[38] N. Shankar, J. Balasubramanian, D. C. Schmidt, G. Biswas, P. Lardieri, E. Mulholland, T. Damiano, A Framework for (Re)Deploying Components in Distributed Real-time and Embedded Systems, in: Poster paper in the Dependable and Adaptive Distributed Systems Track of the 21st ACM Symposium on Applied Computing, Dijon, France, 2005.

[39] J. Hill, J. Slaby, S. Baker, D. Schmidt, Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded Systems QoS, Tech. Rep. ISIS-05-604, Vanderbilt University (2005).