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

Patrick Lardieri[†], Jaiganesh Balasubramanian[‡], Douglas C. Schmidt[‡],
Gautam Thaker[†], Aniruddha Gokhale[‡], and Thomas Damiano[†]
[†]Advanced Technology Labs, Lockheed Martin Corporation,
Cherry Hill, NJ 08002
[‡]Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN 37235

December 26, 2006

Abstract

Enterprise distributed real-time and embedded (DRE) systems can benefit from dynamic management of computing and networking resources 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 two 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 the results of experiments evaluating our MLRM architecture in the context of a representative enterprise DRE system for shipboard computing.

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

1 Introduction

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.

Enterprise DRE systems have a range of QoS requirements that may vary at runtime due to planned [4] and unplanned [5] 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).

Dynamic resource management (DRM) [6, 7] is a promising paradigm for supporting 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. The primary 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.

This paper describes a *multi-layer resource management* (MLRM) architecture we developed to demonstrate DRM capabilities in a shipboard computing environment. This environment consists of a grid of computers that manage many aspects of a ship's power, navigation, command and control, and tactical operations [1] using standards-based DRM services that support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization. Our MLRM 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 computing grid that manage and automate many aspects of shipboard computing. We describe and empirically evaluate how the ARMS MLRM manages computing resources dynamically and ensures proper execution of 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

This section describes the design and functionality of the component middleware used to implement the ARMS MLRM architecture.

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 applied to help resolve key DRM challenges in the ARMS program.

The goals of the ARMS MLRM design are to provide DRM solutions when missions change, resources fail or become available, failures occur due to damage, or new capabilities are added to the system. We addressed the following types of DRM problems to meet the needs of enterprise DRE systems:

• **Mission mode changes**, where the goal is to enable a much broader set of resource reallocations beyond mode changes and behaviors typically provided by domain applications. Meeting this goal 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. The ARMS MLRM therefore ensures that available resources are allocated to the *currently most important* mission capabilities, allowing the system 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 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 enterprise DRE systems operate. This environment involves diverse middleware, operating systems, CPUs, and networks.

The ARMS MLRM services are implemented as a set of common and domain-specific middleware services described in Section 2.2 that communicate using the standard middleware described in Section 2.3 that coordinates and encapsulates a wide range of operating systems, networks, programming languages, and hardware.

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

2.2 The Component-based ARMS MLRM Design

The ARMS MLRM architecture integrates resource management and control algorithms enabled by standard component middleware infrastructure described in Section 2.3 to achieve the enterprise DRE system challenges described in Section 1 and the ARMS MLRM design goals described in Section 2.1. These design goals, combined with the complexity of the enterprise DRE system domain, led us to model the DRM solution space as a layered architecture comprising components at different levels of abstraction. Figure 1 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.

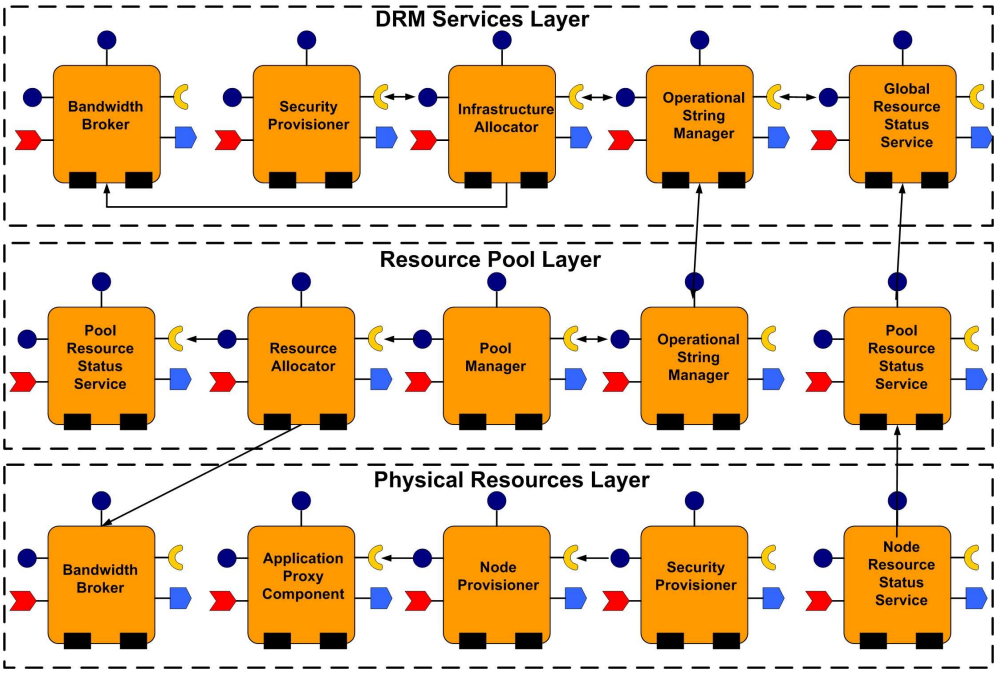

Figure 1: **Components in the ARMS MLRM**

2.2.1 ARMS MLRM Structure

We first describe the key goals of each layer of the ARMS MLRM structure (as depicted in Figure 1), including the DRM Services layer, the Resource Pool layer, and the Physical Resource layer, and explain how key components within these layers help address the goals described in Section 2.1. The dynamics of these key MLRM components are described further in Section 2.2.2.

- The **DRM Services layer** is responsible for satisfying shipboard computing missions, such as allocating system computing and networking resources to tracks items of interest and planning necessary actions. The goal of this layer is to maximize the mission coverage and reliability. The DRM Services layer receives explicit resource

management requests from applications, along with command and policy inputs. It decomposes mission requests 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 system. 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 mission critical and more important 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. The ARMS MLRM employs importance-ordered, uni-dimensional bin-packing [8] of worst-case application CPU resources to allocate operational strings to computing nodes. It also employs a network provisioning algorithm [9] 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. When appropriate metadata is provided, ARMS MLRM also performs end-to-end response time schedulabililty analysis [10] 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 [11] 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. 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 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 system. 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.

2.2.2 ARMS MLRM Dynamics

The ARMS MLRM acts in response to incidences of resource management problems. To illustrate the dynamic behavior of the ARMS MLRM, we next describe the two examples shown in Figure 2: 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 [12]. The experimental results for both these workflows are described in Section 3. The sequence of

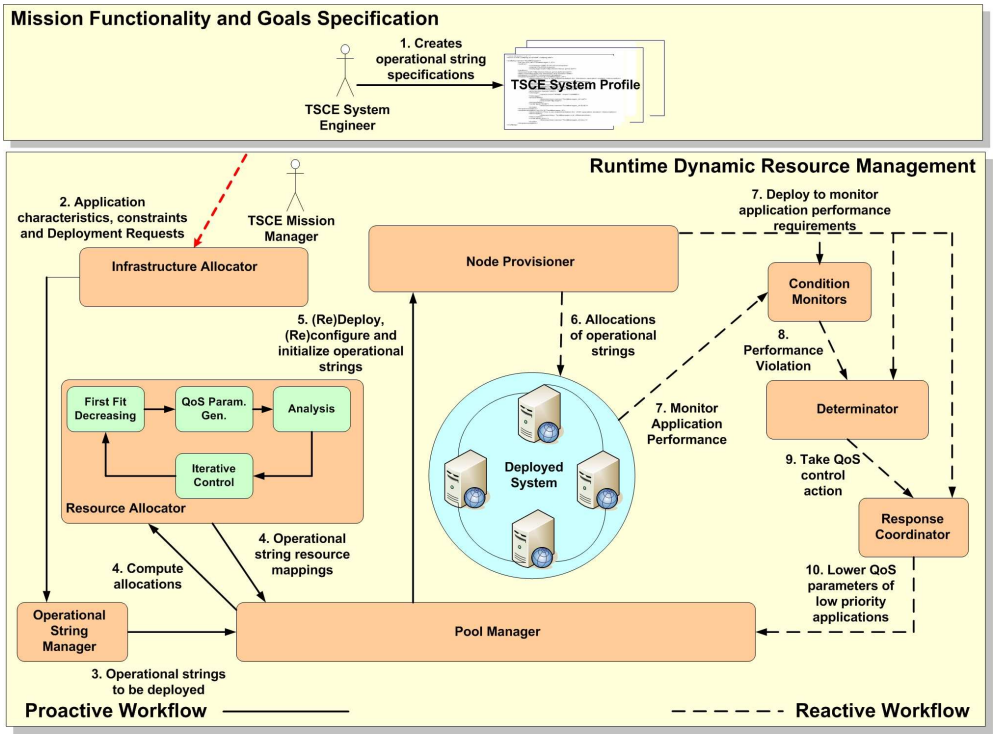

Figure 2: **MLRM Workflow**

activities happening in a typical shipboard computing environment as the system goes through proactive and reactive workflows are shown in Figure 2 and described below:

1. System mission management services selectively deploy and suspend mission software packages in response to tactical needs. To plan the deployment at design time, a mission engineer specifies the sets of operational strings to deploy 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 operational strings.

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* [13] 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 the threat response operational string was unable handle higher load because of competing equally important threat tracking strings.

10. The *Response Coordinator* addresses starvation by lowering the QoS parameters (*i.e.*, OS priority and schedul-

ing 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 system 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. In this case, the ARMS MLRM temporarily elevates the importance of the threat response string in accordance with changing mission priorities, at the cost of reducing the performance of the less important threat tracking string.

2.3 The ARMS MLRM QoS-enabled Middleware Infrastructure

To simplify development and enhance reusability, the ARMS MLRM components described in Section 2.2 are based on a QoS-enabled component middleware infrastructure consisting of the *Component-Integrated ACE ORB* (CIAO) [14] and the *Deployment And Configuration Engine* (DAnCE) [15]. CIAO and DAnCE implement the *Real-time CORBA Component Model* (RT-CCM). RT-CCM 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 MLRM and application components in enterprise DRE systems. CIAO and DAnCE use patterns [18] to achieve highly portable and standards-compliant component middleware and reflective middleware techniques [1] to support key QoS aspects for enterprise DRE systems.

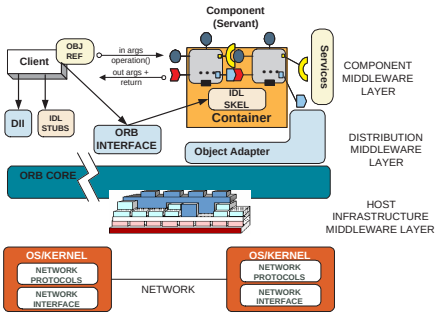

Figure 3: **CIAO Architecture**

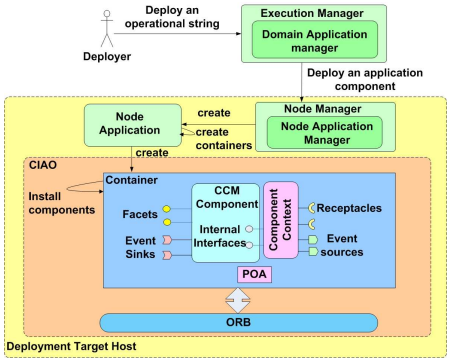

Figure 4: **DAnCE Architecture**

As shown in Figure 3, *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).

MLRM and application component assemblies developed using CIAO are deployed and configured via DAnCE, which implements the standard OMG *Deployment and Configuration* (D&C) specification [19] shown in Figure 4. DAnCE manages the mapping of MLRM and application components onto nodes in our target environment. The information about the component assemblies and the target environment in which the 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.

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 processors 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 via 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 [20] 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 scenario that guided the experiments. In this scenario many sensors (ED) collaborate with planning processes (PLAN), configuration processes (CFGOP), 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 shipboard computing capabilities. The MLRM therefore deploys different sets of

operational strings depending on 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 a 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 2.2.

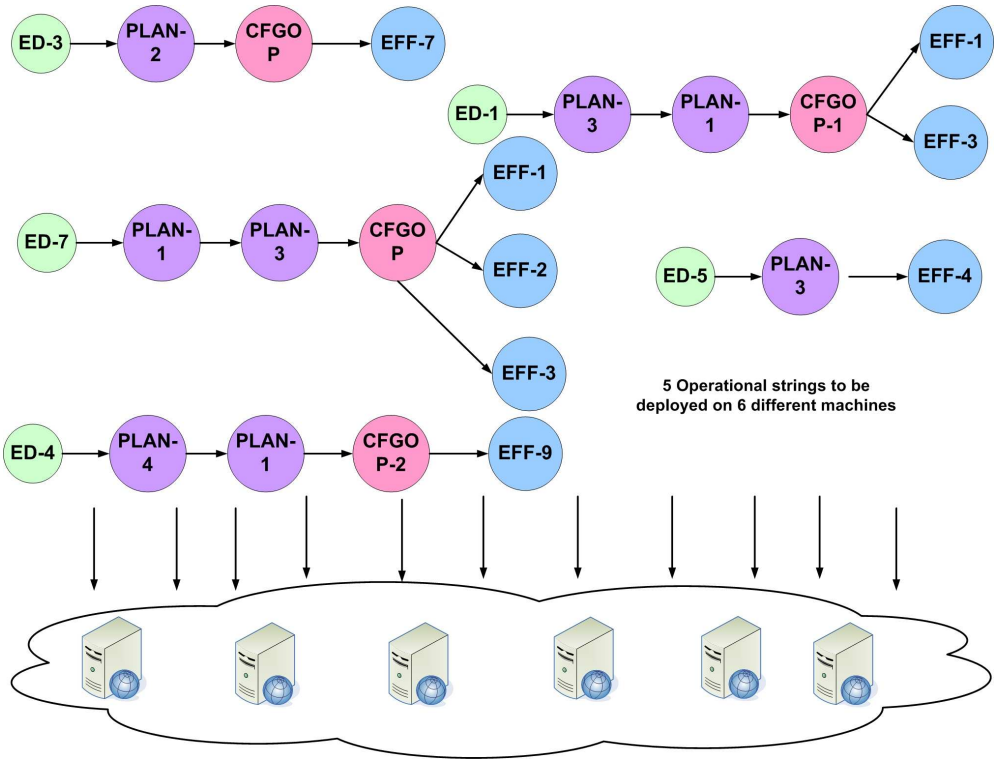

Figure 5: **Operational Strings Deployed in ARMS MLRM Experiments**

Our goals in conducting the experiments 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 half the time of 4 minutes it takes to recover in a manually managed system.

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 shipboard computing 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 scenario described in Figure 5 to identify five operational strings that map to five different functionalities. Each operational string involved 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 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 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 (not all 70 applications are shown in the Figure 6.) 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 [21] – leads to a fairly balanced workload across the processors. These results indicate that

- The ARMS MLRM allocation algorithms have the capability to generate 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

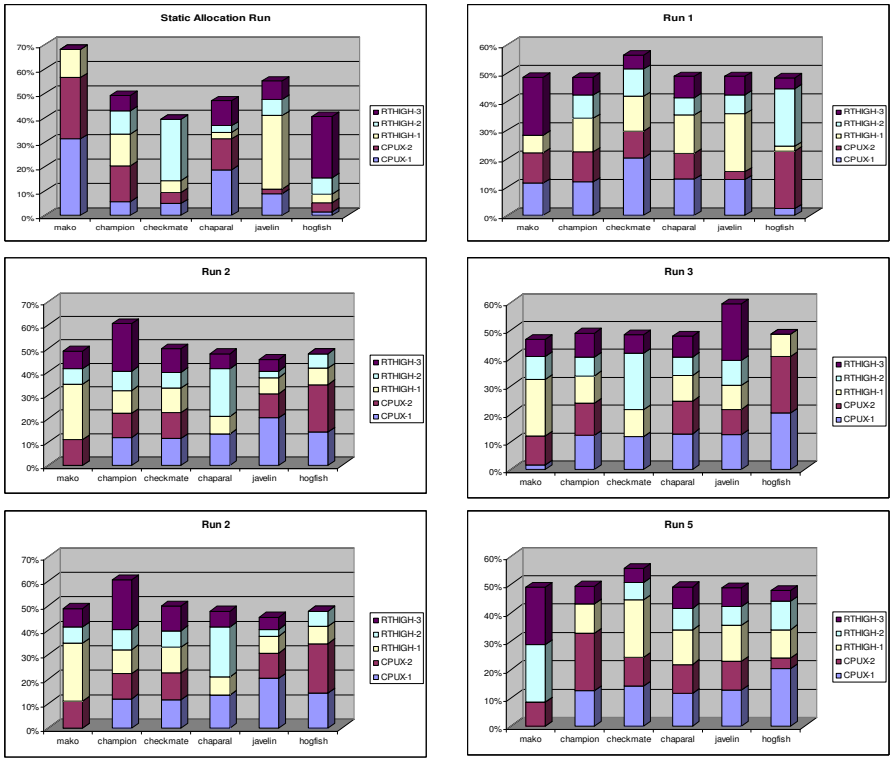

Figure 6: **Different Configurations of Operational Strings in Six Nodes**

depending upon resource availabilities and hardware failures. As a result, mission-critical 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 2.2 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 deploys 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, all operational strings may not be duplicated and only certain mission-critical operational strings would be replicated and deployed onto different resource pools. In tests results described here all strings were duplicated since the overall test size is modest and we had sufficient resources at hand.

Figure 7 shows how operational strings consisting of a combination of sensors and effectors in our scenario are deployed together with their replicas across two different resource pools. The two resource pools are *Pool-1.A* and *Pool-1.B* and each resource pool has three nodes. The primary and the secondary replicas for a particular operational string are distributed across the two resource pools to ensure reliable replication. For example, the

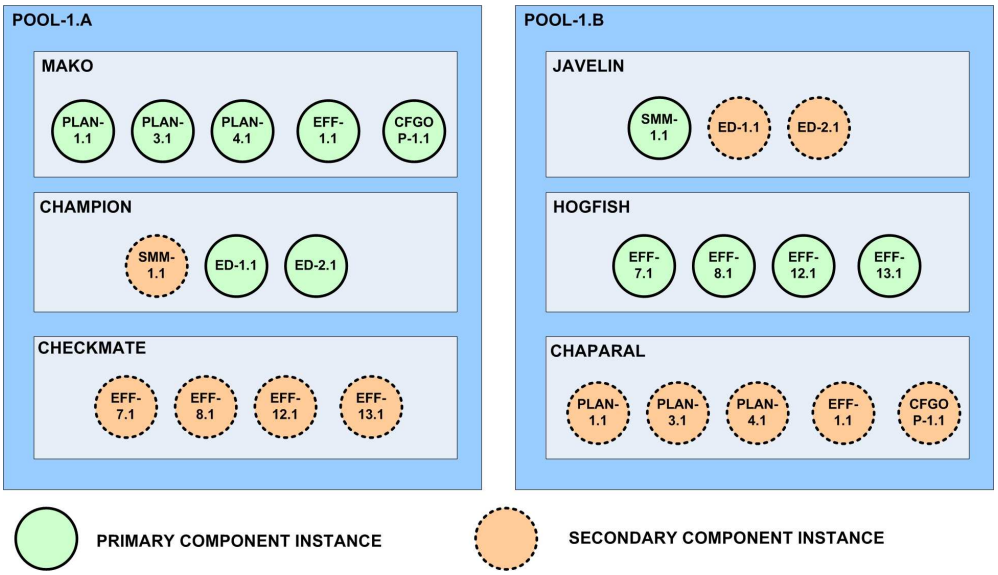

Figure 7: **Operational Strings and Replicas Deployed in Different Resource Pools**

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 reacts to pool failures to reconstitute the primary operational strings from the failed pools onto pools that are safe, thereby ensuring critical mission functionality is available throughout the system lifecycle.

As shown in Figure 7, the failure scenario begins with the failure of host *Checkmate* and results in the loss of secondary replicas for the primary applications in Pool 1.B. The ARMS MLRM is configured to *not* restart the failed secondaries for experimental purposes. We next failed Pool 1.B, which resulted in a loss of the primary applications in Pool 1.B. There is no active replication since the secondaries failed first, so the MLRM must restart the applications in the surviving resource pool to regain lost capability. As described in Section 2.2.1 and Section 2.2.2, the ARMS MLRM defines Condition Monitors, Determinators, and Response Coordinators to detect and recover from pool failures. We deployed Pool 1.B's failure detectors in Pool 1.A to monitor heartbeats from Pool 1.B's Resource Status Service[1]. When a failure was induced by disconnecting Pool 1.B's network from the router, the failure detectors detected the failure after missing three heartbeats, which triggered the Pool Failure Response Coordinator to start failure recovery actions.

As this scenario unfolded, 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 collaborating with other strings, and (5)

[1]This failure detection scheme was sufficient to evaluate the MLRM framework's failure and subsequent reactive resource management capabilities.

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. Conservative timeout values were

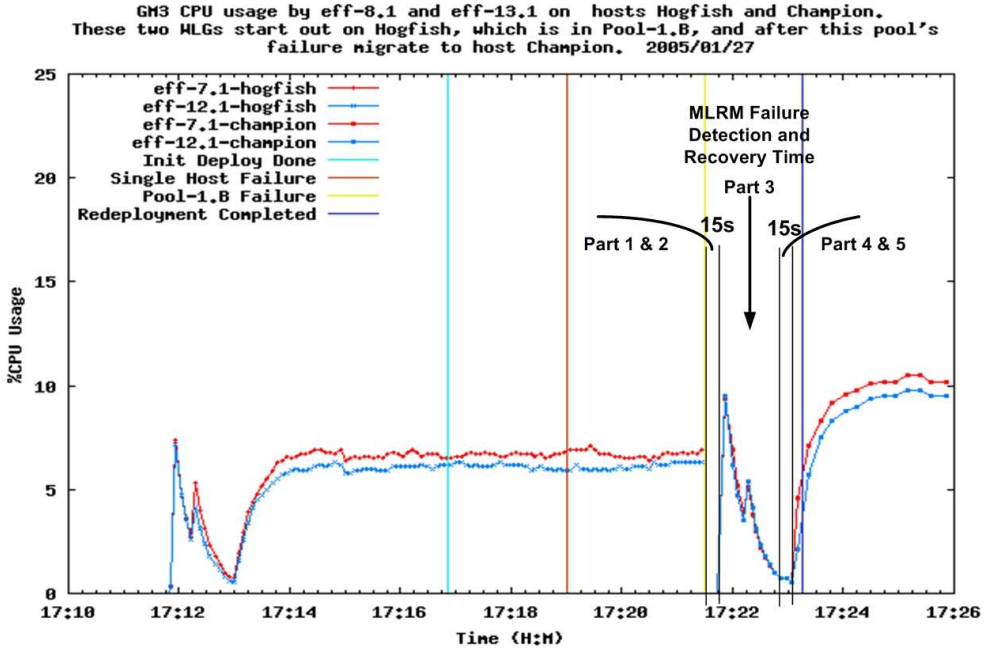

Figure 8: **Redeployment of Effectors from Failed Pool**

used for failure detection to minimize false alarms. A more aggressive use of timeouts (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) is well below a second. A relatively large gap is noted for part 3 (time to spawn new processes to host application components), which is due to overhead in DAnCE that will be fixed in a future release. 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 half the 4 minutes needed to recover a manually managed system.

3.4 MLRM Capability to Operate under Limited Resource Availabilities

Section 2.2 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 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 of

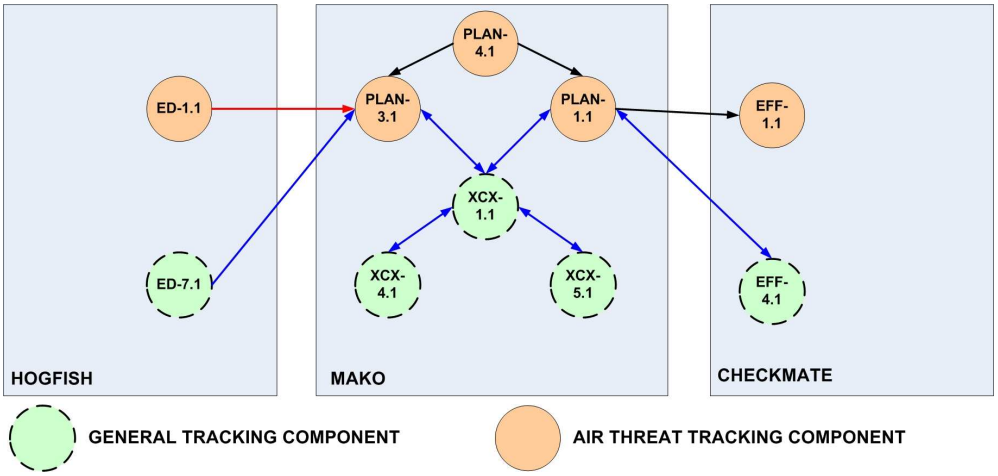

Figure 9: **Different Threat Tracking Components in the ARMS MLRM**

both the operational strings are deployed and launched. We also measured *Mako's* CPU utilization as air threats began to increase, which we accomplished by configuring the *ED-1.1* sensor to generate more messages – and thereby more reactive events – within the *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 to highlight its capabilities.

Figure 10 shows the results of an experiment that disabled the ARMS MLRM capabilities. As the processing load for Plan-1.1 increased beyond a certain point the system started to become unstable, with significant fluctuations in the amount of resources consumed. In contrast, Figure 11 shows the results of enabling the ARMS MLRM DRM capabilities. as load for Plan-1.1 increased the MLRM made the necessary dynamic adjustments to deliver steady resource availability for this 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 half the time it takes for manual recovery
- Under increased load MLRM makes the necessary resource allocation adjustments to permit steady state

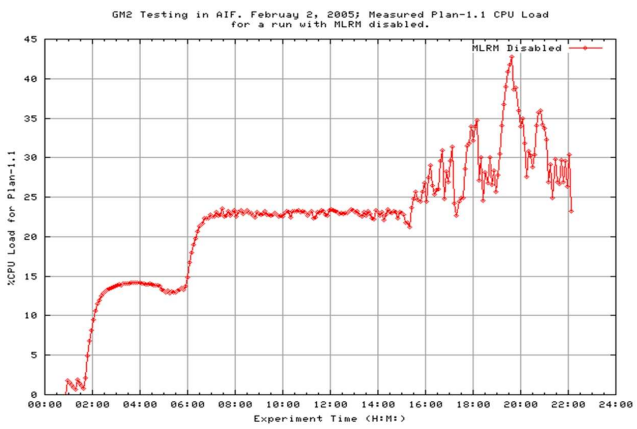

Figure 10: **Handling Threats with MLRM Disabled**

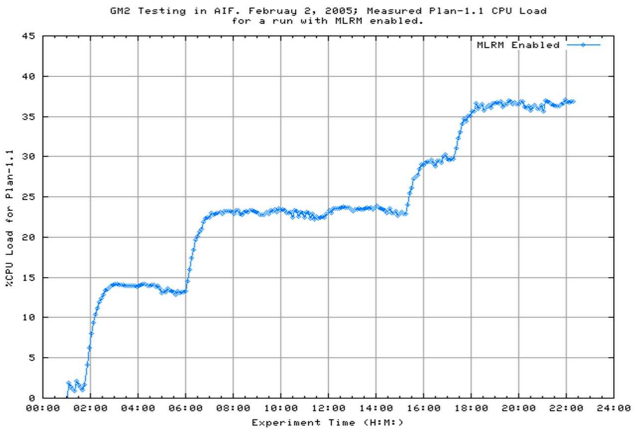

Figure 11: **Handling Threats with MLRM Enabled**

increase in resource availability to critical applications without any instability.

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 [22] or network layer [9]. DRM capabilities have also been applied either in standalone [23] or distributed [24] deployments. In this section we summarize related DRE research along the two dimensions.

Application-driven distributed DRM was described in [22], 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 shipboard computing 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 [25] 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 [24], where the authors describe the DeSiDeRaTa middleware that provides DRM for QoS-sensitive applications. This work and related efforts in [26] 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 [9]. 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 shipboard computing environment include:

- ARMS MLRM research and experiments shows that dynamic resource management (DRM) – using standards-based middleware technologies – is not only feasible, but can (1) handle dynamic resource allocations across a wide array of configurations and capacities, (2) provide continuous availability for critical functionalities – even in the presence of node and pool failures – through reconfiguration and redeployment 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 manual engineering effort to create and validate any (re)configuration and (re)allocation. This tedious and error-prone manual process marginalizes any advantage that might be gained by leveraging allocation and configuration as a means of application resiliency. The DRM capabilities provided by ARMS MLRM help alleviate much of this inflexibility by consistently achieving well-balanced allocation under varying conditions, and permiting a range of automated, dynamic management of resources that greatly extend operational flexibility of the system without human intervention. Moreover, our experiments clearly demonstrate that the performance of ARMS MLRM is enhanced when DRM services are enabled – and in fact allow MLRM to operate in the presence of failures that can not be accommodated by static resource management.
- Enterprise DRE systems have a significant number of software architecture components that require QoS, configuration, and deployment information. Although ARMS MLRM provides an effective software infras-

tructure for DRM, it is a complex task to capture the required application information for all components. To deal with this complexity, we are exploring model-driven development (MDD) tools [27] that enable a software infrastructure consisting of standards-based middleware and components so that MLRM functions and workflow can be performed more effectively and productively.

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 - The Journal of Defense Software Engineering.

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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).

[11] 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.

[12] 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.

[13] 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.

[14] 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, 2004, pp. 131–162.

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

[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/05-01-04 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] Object Management Group, Deployment and Configuration Adopted Submission, OMG Document mars/03-05-08 Edition (Jul. 2003).

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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, pp. 190–199.