Efficient and Deterministic Application Deployment in Component-based Enterprise Distributed Real-time and Embedded Systems

William R. Otte[a,b], Aniruddha Gokhale[a], Douglas C. Schmidt[a]

[a]*Institute for Software Integrated Systems, EECS, Vanderbilt University, 1025 16th Ave S, Suite 102 Nashville, TN 37212*
[b]*Corresponding Author, wotte@dre.vanderbilt.edu*

Abstract

Context: Component-based middleware, such as the *Lightweight CORBA Component Model*, is increasingly used to implement enterprise distributed real-time and embedded (DRE) systems. In addition to supporting the quality-of-service (QoS) requirements of individual DRE systems, component technologies must also support bounded latencies when effecting deployment changes to DRE systems in response to changing environmental conditions and operational requirements.

Objective: The goals of this paper are to (1) study sources of inefficiencies and non-deterministic performance in deployment capabilities for DRE systems and (2) devise solutions to overcome these performance problems.

Method: The paper makes two contributions to the study of the deployment and configuration of distributed component based applications. First, we analyze how conventional implementations of the OMG's Deployment and Configuration (D&C) specification for component-based systems can significantly degrade deployment latencies. Second, we describe architectural changes and performance optimizations implemented within the *Locality-Enhanced Deployment and Configuration Engine* (LE-DAnCE) implementation of the D&C specification to obtain efficient and deterministic deployment latencies.

Results: We analyze the performance of LE-DAnCE in the context of component deployments on 10 nodes for a representative DRE system consisting of 1,000 components and in a cluster environment with up to 100 nodes. Our results show LE-DAnCE's optimizations provide a bounded de-

ployment latency of less than 2 seconds for the 1,000 component scenario with just a 4 percent jitter.

Conclusion: The improvements contained in the LE-DAnCE infrastructure provide an efficient and scaleable standards-based deployment system for component-based enterprise DRE systems. In particular, deployment time parallelism can improve deployment latency significantly, both during pre-deployment analysis of the deployment plan and during the process of installing and activating components.

---

1. Introduction

Emerging trends and challenges. Component-based software engineering techniques are increasingly applied to develop enterprise *distributed real-time and embedded* (DRE) systems, such as air-traffic management [1], shipboard computing environments [2], and distributed sensor webs [3]. These systems are often characterized as "open" since applications running in them must contend not only with changing environmental conditions (such as changing power levels, operational nodes, or network status), but also evolving operational requirements and mission objectives [4].

To adapt to changing environments and operational requirements, it may be necessary to change the deployment and configuration characteristics of these DRE systems dynamically. Examples of potential adaptations include deployment or tear down of individual component instances, changing connection configuration, or altering QoS properties in the target component runtime. As a result of stringent *quality-of-service* (QoS) requirements in DRE systems, it is important that any deployment and configuration changes occur as quickly and predictably as possible, *i.e.*, with short and bounded deployment latencies.

Not only are timely and dependable runtime deployment and configuration changes essential in DRE systems, even initial application startup times can be an important metric. For example, in highly energy-constrained systems (such as distributed sensor networks), a common power saving strategy may involve completely deactivating field hardware and periodically restarting it to take new measurements or activate actuators [5]. In such environments, deployments must be fast and time-bounded.

To support these requirements, the efficiency and QoS (*e.g.*, latency of deployment) provided by the deployment infrastructure should be considered alongside the component middleware used to develop DRE systems.

Standards, such as the OMG *Deployment and Configuration* (D&C) specification [6] for component-based applications, have emerged in recent years. The OMG D&C specification provides comprehensive development, packaging, and deployment frameworks for a range of component middleware.[1]

In prior work, we developed the *Deployment and Configuration Engine* (DAnCE) [8], which focused solely on separating concerns defined by the OMG D&C specification and demonstrating its feasibility without concern for timeliness and deterministic behavior. After applying DAnCE to a range of representative DRE systems [2, 5], however, we found the lack of appropriate optimizations and architectural limitations of the OMG D&C specification yielded performance bottlenecks that adversely impacted deployment latencies and hindered reproducibility. Moreover, these performance bottlenecks stemmed from more than just limitations with the original DAnCE implementation, but involve inherent architectural limitations with the OMG D&C specification itself.

Solution approach → Architectural optimizations to D&C to leverage parallelism and reduce serialized phasing. To overcome the limitations described above, this paper motivates and describes architectural enhancements we made to the OMG D&C specification to achieve deterministic deployment latencies for enterprise DRE systems. Our solution is called the *Locality-Enhanced Deployment and Configuration Engine* (LE-DAnCE), which significantly extends our earlier work on DAnCE by the following enhancements:

- **Enhancement 1.** Reducing latency due to loading of XML-based deployment plans by serializing them into an appropriate in-memory format.
- **Enhancement 2.** Adopting a new algorithmic approach to runtime plan analysis to better take advantage of opportunities for parallelization.
- **Enhancement 3.** Providing a novel architectural approach to deployment called the *Locality Manager* that allows LE-DAnCE to reduce serialized phasing of deployment activities.

[1] Although originally developed for the *CORBA Component Model* (CCM) [7], the OMG D&C specification is defined via a UML metamodel that is generic with respect to the actual target component platform and therefore applicable to other component models, such as Enterprise JavaBeans or the Fractal component model.

This paper extends our previous work [9], which focused on minimizing a key source of significant deployment latency: *the runtime plan analysis needed to determine how large deployments should be split amongst several nodes and to determine installation ordering.* This paper augments this prior work by investigating the impact of runtime analysis on deployment latency for enterprise DRE systems. We also report the results of experiments conducted in a cluster environment to evaluate how leveraging the embarrassingly parallel nature of such runtime analysis can significantly reduce deployment latency incurred by this phase of the deployment process.

Paper organization. The remainder of this paper is organized as follows: Section 2 summarizes the OMG D&C specification and analyzes key sources of overhead stemming from architectural limitations with the OMG D&C specification and naïve implementation techniques adopted in our original DAnCE implementation of the specification; Section 3 describes how we addressed these sources of overhead in LE-DAnCE, focusing on deployment latency; Section 4 analyzes the results of experiments we conducted to compare LE-DAnCE with DAnCE; Section 5 compares our research with related work on deploying and configuring large-scale distributed applications; and Section 6 presents concluding remarks and lessons learned.

2. Impediments to Predictable Deployment Latency

This section presents an overview of the OMG *Deployment and Configuration* (D&C) specification for component-based applications and describes how an implementation of this specification called the *Deployment and Configuration Engine* (DAnCE) [8] supports the separation of concerns espoused in the D&C specification. We pinpoint key sources of overhead that impact deployment latencies in DRE systems and expose the architectural limitations in the D&C specification that exacerbate these overheads.

2.1. Overview of the OMG D&C Standard

The OMG D&C specification provides standard interchange formats for metadata used throughout the component-based application development lifecycle, as well as runtime interfaces used for packaging and planning. These runtime interfaces deliver deployment instructions to the middleware deployment infrastructure via a *component deployment plan*, which contains the complete set of deployment and configuration information for component instances and their associated connection information. During DRE system

initialization, such information must be parsed, components deployed on the nodes, and the system activated in a timely and deterministic manner.

Below we focus on the standard interfaces, metadata, and architecture used for runtime deployment and configuration in implementations compliant with the D&C specification. Throughout this section we refer to the timeliness of the deployment infrastructure to execute the deployment plan as the "deployment latency," which includes the time starting when a deployment plan is provided to the deployment infrastructure to the time at which all deployment instructions have been executed and the system activated. We refer to the reproducibility of the timeliness in deployment as the deterministic nature of deployment.

2.1.1. Runtime D&C Architecture

The runtime interfaces defined by the OMG D&C specification for deployment and configuration consists of the two-tier architecture shown in Figure 1. This architecture consists of (1) a set of global (system-wide) enti-

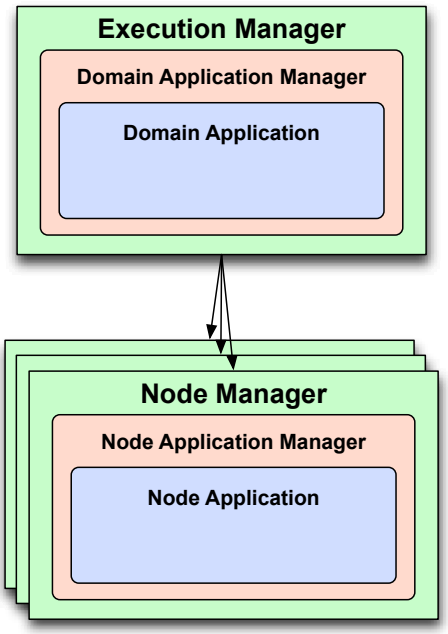

Figure 1: OMG D&C Architectural Overview and Separation of Concerns

ties used to coordinate deployment and (2) a set of local (node-level) entities used to instantiate component instances and configure their connections and

QoS properties. Each entity in these global and local tiers correspond to one of the following three major roles:

- **Manager.** This role (known as the *ExecutionManager* at the global-level and as the *NodeManager* at the node-level) is a singleton daemon that coordinates all deployment entities in a single context. The Manager serves as the entry point for all deployment activity and as a factory for implementations of the *ApplicationManager* role.

- **ApplicationManager.** This role (known as the *DomainApplicationManager* at the global-level and as the *NodeApplicationManager* at the node-level entity) coordinates the lifecycle for running instances of a component-based application. Each ApplicationManager represents exactly one component-based application and is used to initiate deployment and teardown of that application. This role also serves as a factory for implementations of the *Application* role.

- **Application.** This role (known as the *DomainApplication* at the global-level and the *NodeApplication* at the node-level entity) represents a deployed instance of a component-based application. It is used to finalize the configuration of the associated component instances that comprise an application and begin execution of the deployed component-based application.

2.1.2. D&C Deployment Data Model

In addition to the runtime entities described above, the D&C specification also contains an extensive data model that is used to describe component applications throughout their deployment lifecycle. The metadata defined by the specification is intended for use as

- An interchange format between various tools (*e.g.*, development tools, application modeling and packaging applications, and deployment planning tools) applied to create the applications and

- Directives that describe the configuration and deployment used by the runtime infrastructure.

Most entities in the D&C metadata contain a section where configuration information may be included in the form of a sequence of name/value pairs,

where the value may be an arbitrary data type. This configuration information can be used to describe everything from basic configuration information (such as shared library entry points and component/container associations) to more complex configuration information (such as QoS properties or initialization of component attributes with user-defined data types).

This metadata can broadly be grouped into three categories: *packaging*, *domain*, and *deployment*. Packaging descriptors are used from the beginning of application development to specify component interfaces, capabilities, and requirements. After implementations have been created, this metadata is further used to group individual components into assemblies, describe pairings with implementation artifacts, such as shared libraries (also known as dynamically linked libraries), and create packages containing both metadata and implementations that may be installed into the target environment. Domain descriptors are used by hardware administrators to describe capabilities (*e.g.*, CPU, memory, disk space, and special hardware such as GPS receivers) present in the domain.

Both the domain and packaging metadata are then used by a planning agent (either a human or automated software tool) to map the described component instances onto the underlying runtime environment through the creation of the third type of metadata supported by the OMG D&C standard: the *component deployment plan*, which contains the information shown in Figure 2 and described below:

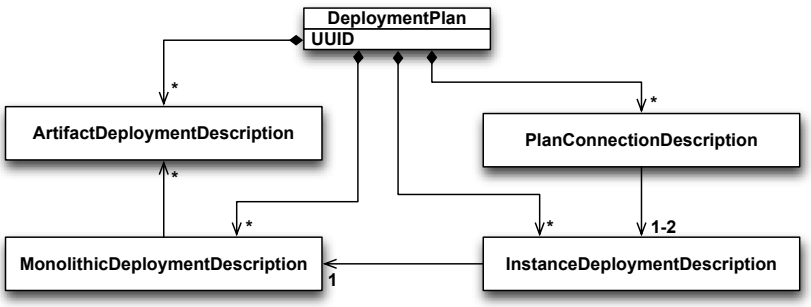

Figure 2: Component Deployment Plan

- **Implementation Artifact Descriptions (IAD)**. The IAD section of the deployment plan describes the various artifacts that must be present on a node for successful component deployment. Artifacts include—but

are not limited to—executable files and shared libraries that provide binary implementations of components.

- **Monolithic Deployment Descriptions (MDD)**. The MDD section references all IAD entries necessary for one particular component type. It also contains additional configuration information that is necessary for all instances of that type, *e.g.* entry points and factory functions used to load the implementation from shared libraries.

- **Instance Deployment Descriptions (IDD)**. IDD entries represent concrete instances deployed into the domain. This section of the metadata describes the node in which a particular component should be instantiated and contains additional configuration properties that should be applied to that instance, *e.g.*, QoS configuration information.

- **Plan Connection Descriptions (PCD)**. The PCD section describes all connections that must be established as part of the deployment. These entries reference application IDD entries that are part of a particular connection and contains additional information (such as port names and QoS configuration) that may be necessary for the connection to be successfully established.

- **Plan Locality Constraints (PLC)**. The PLC section describes constraints on how individual component instances may be collocated inside processes and/or containers. Constraints are established between groups of instances and may be one of three values: (1) no constraint, (2) same process, or (3) different process. These constraints are evaluated at deployment time and used by the node-level infrastructure to determine the number of application processes to spawn in which to run all the components defined in the deployment plan.

The OMG D&C standard suggests that all metadata be serialized to an XML format for on-disk storage and for use as an interchange format between the various tools used for application development and planning. This XML format must be converted into the native binary format used in the interfaces of the runtime infrastructure, however, so the deployment infrastructure can use it.

2.1.3. OMG D&C Deployment Process

Component application deployments are performed in a four phase process codified by the OMG D&C standard. The *Manager* and *ApplicationManager* are responsible for the first two phases and the *Application* is responsible for the final two phases, as described below:

1. **Plan preparation.** In this phase, a deployment plan is provided to the *ExecutionManager*, which (1) analyzes the plan to determine which nodes are involved in the deployment and (2) splits the plans into "locality-constrained" plans, one for each node containing information only for the corresponding node. These locality-constrained plans have only instance and connection information for a single node. Each *NodeManager* is then contacted and provided with its locality-constrained plan, which causes the creation of *NodeApplicationManagers* whose reference is returned. Finally, the *ExecutionManager* creates a *DomainApplicationManager* with these references.
2. **Start launch.** When the *DomainApplicationManager* receives the start launch instruction, it delegates work to the *NodeApplicationManagers* on each node. Each *NodeApplicationManager* creates a *NodeApplication* that loads all component instances into memory, performs preliminary configuration, and collects references for all endpoints described in the deployment plan. These references are then cached by a *DomainApplication* instance created by the *DomainApplicationManager*.
3. **Finish launch.** This phase is started by an operation on the *DomainApplication* instance, which apportions its collected object references from the previous phase to each *NodeApplication* and causes them to initiate this phase. All component instances receive final configurations and all connections are then created.
4. **Start.** This phase is again initiated on the *DomainApplication*, which delegates to the *NodeApplication* instances and causes them to instruct all installed component instances to begin execution.

2.2. Sources of Deployment Latency Overheads

The remainder of this section pinpoints the sources of overhead that impact deployment latencies in the context of the OMG D&C specification described above. We use our open-source DAnCE [8] OMG D&C implementation as a vehicle to demonstrate these sources of overhead. The major

sources of latency stem from multiple complexities in the OMG D&C standard, including the processing of deployment metadata from disk in XML format and an architectural ambiguity in the runtime infrastructure that encourages suboptimal implementations.

2.2.1. Challenge 1: Minimizing the Latency of Parsing Deployment Plans

Component application deployments for OMG D&C are described by a data structure that contains all the relevant configuration metadata for the component instances, their mappings to individual nodes, and any connection information required. This deployment plan is serialized on disk in a XML file whose structure is described by an XML Schema defined by the D&C specification. This XML document format presents significant advantages by providing a simple interchange format for exchanging deployment plan files between modeling tools [10]. This format is also easy to generate and manipulate using widely available XML modules for popular programming languages and it enables simple modification and data mining by text processing tools such as Perl, grep, sed, and awk.

Processing these deployment plan files during deployment and even runtime, however, can lead to substantial deployment latency costs, as shown in Section 4.1.2. This increased latency stems from the following sources:

- XML deployment plan file sizes grow substantially as the number of component instances and connections in the deployment increases, which causes significant I/O overhead to load the plan into memory and to validate the structure against the schema to ensure that it is well-formed.
- The XML document format cannot be directly used by the deployment infrastructure because the infrastructure is a CORBA application that implements OMG *Interface Definition Language* (IDL) interfaces. Hence, the XML document must first be converted into the IDL format used by the runtime interfaces of the deployment framework.

In enterprise DRE systems, component deployments that number in the thousands are not uncommon. Moreover, component instances in these domains will exhibit a high degree of connectivity. Given the structure of deployment plans outlined in Section 2.1.2, both these factors contribute to large plans.

While the latency source described above is most immediately applicable to initial application deployment, it can also present a problem during

potential re-deployment activities at application runtime that involve significant changes to the application configuration. While deployment plan files that represent re-deployment or re-configuration instructions may not be as large as for the initial deployment, the responsiveness of the deployment infrastructure during these activities is even more important to ensure that an application continues to meet its stringent QoS and end-to-end deadlines during online modifications. Section 3.1 describes how LE-DAnCE resolves the challenge of minimizing the latency associated with parsing deployment plans by pre-processing large plans offline into a portable binary representation.

2.2.2. Challenge 2: Optimizing Runtime Plan Analysis

After a component deployment plan has been loaded into an in-memory representation, it must then be analyzed by the middleware deployment infrastructure before any subsequent deployment activity is performed. This analysis occurs during the plan preparation phase described in Section 2.1.3 to determine (1) the number of deployment sub-problems that are part of the deployment plan and (2) which component instances belong to each sub-problem. As mentioned earlier, the output of this analysis process is a set of "locality-constrained" sub-plans. A locality-constrained sub-plan contains all the necessary meta-data to execute a deployment successfully, and as such contains copies of the information contained in the original plan (described in Section 2.1.2).

The runtime plan analysis is actually conducted twice during the plan preparation phase of deployment: once at the global level and again on each node. Global deployment plans are split according to the node that the individual instances are assigned to. This two-part analysis results in a new sub-plan for each node that only contains the instances, connections, and other component meta-data necessary for that node.

Each sub-plan is then transmitted to the appropriate node, which begins its own analysis process. At the node level, the plan is split according to an evaluation of the PLC elements. This split yields a number of new sub-plans created for each process that will be created on that node. Again, each sub-plan contains only the components and connection meta-data necessary for that process.

The algorithm for splitting plans used by our DAnCE implementation of the D&C specification is straightforward. For each instance (IDD) in the plan, the algorithm determines which sub-plan should contain it and

retrieve the appropriate (or create a new) sub-plan data structure. The IDD is then copied to the sub-plan, along with the appropriate MDD and IAD (discovered via references described earlier). Finally, the algorithm searches through the PCD and PLC entries in the global plan that reference the IDD under consideration and similarly copies them.

While this approach is conceptually simple, it is fraught with accidental complexities that yield the following inefficiencies in practice:

1. **Reference representation in IDL**. Deployment plans are typically transmitted over networks, so they must obey the rules of the CORBA IDL language mapping. Since IDL does not have any concept of references or pointers, some alternative mechanism must be used to describe the relationships shown in Figure 2. The deployment plan stores all the major elements in sequences, so references to other entities can be represented with simple indices into these sequences. While this implementation can follow references in constant time, it also means these references become invalidated when plan entities are copied to sub-plans, as their position in deployment plan sequences will most likely be different. It is also impossible to determine if the target of a reference has already been copied without searching the sub-plan, which is time-consuming.
2. **Memory allocation in deployment plan sequences**. The CORBA IDL mapping requires that sequences be stored in consecutive memory addresses. If a sequence is resized, therefore, its contents will most likely be copied to another location in memory to accommodate the increased sequence size. With the approach summarized above, substantial copying overhead will occur as plan sizes grow.
3. **Inefficient parallelization of plan analysis**. The algorithm described above would appear to benefit greatly from parallelization, as the process of analyzing a single component and determining which elements must be copied to a sub-plan is independent of all other components. Multi-threading this algorithm, however, would likely not be effective because access to sub-plans to copy instance meta-data must be serialized to avoid data corruption. In practice, component instances in the deployment plan are usually grouped according to the node and/or process since deployment plans are often generated from modeling tools. As a result, multiple threads would likely compete for a lock on the same sub-plan, which would cause the "parallelized"

algorithm to run largely sequentially.

Section 4.1.3 quantifies these inefficiencies in DAnCE for a representative DRE environment and Section 3.2 describes how the plan analysis process has been improved in LE-DAnCE to provide more efficient and deterministic performance.

2.2.3. Challenge 3: Overly Serialized Execution of Deployment Actions

The complexities presented in this section involve the serial (non-parallel) execution of deployment tasks. The related sources of latency in DAnCE exist at both the global and node level. At the global level, this lack of parallelism results from the underlying CORBA transport used by DAnCE. The lack of parallelism at the local level, however, results from the lack of specificity in terms of the interface of the D&C implementation with the target component model that is contained in the D&C specification.

The D&C deployment process presented in Section 2.1.3 enables global entities to divide the deployment process into a number of node-specific subtasks. Each subtask is dispatched to individual nodes using a single remote invocation, with any data produced by the nodes passed back to the global entities via "out" parameters that are part of the operation signature described in IDL. Due to the synchronous (request/response) nature of the CORBA messaging protocol used to implement DAnCE, the conventional approach is to dispatch these subtasks serially to each node. This approach is simple to implement in contrast to the complexity of using the CORBA *asynchronous method invocation* (AMI) mechanism [11].

To minimize initial implementation complexity, we used synchronous invocation in an (admittedly shortsighted) design choice in the initial DAnCE implementation. This global synchronicity worked fine for relatively small deployments with less than ∼100 components. As the number of nodes and instances assigned to those nodes scaled up, however, this global/local serialization imposed a substantial cost in deployment latency.

This serialization problem, however, is not limited only to the global/local task dispatching; it exists in the node-specific portion of the infrastructure, as well. The D&C specification provides no guidance in terms of how the NodeApplication should interface with the target component model, such as the CORBA Component Model (CCM), instead leaving such an interface as an implementation detail. In DAnCE, the D&C architecture was implemented using three processes, as shown in Figure 3. The ExecutionManager and

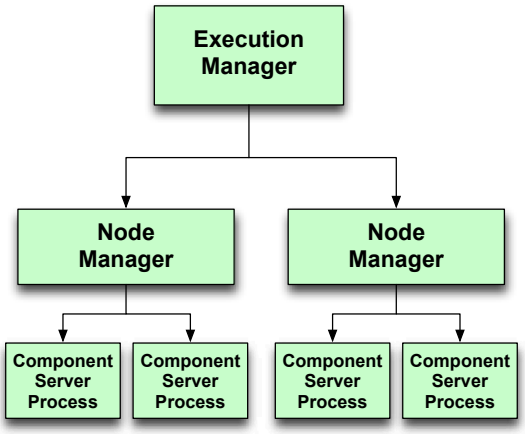

Figure 3: Simplified Serialized DAnCE Architecture

NodeManager processes instantiate their associated ApplicationManager and Application instances in their address spaces. When the NodeApplication installs concrete component instances it spawns one (or more) separate application processes as needed. These application processes use an interface derived from an older version of the CCM specification that allows the NodeApplication to instantiate containers and component instances individually. This approach is similar to that taken by CARDAMOM [12] (which is another open-source CCM implementation) that is tailored for enterprise DRE systems, such as air-traffic management systems.

The DAnCE architecture shown in Figure 3 was problematic with respect to parallelization since its NodeApplication implementation integrated all logic necessary for installing, configuring, and connecting instances directly (as shown in Figure 4), rather than performing only some processing and delegating the remainder of the concrete deployment logic to the application process. This tight integration made it hard to parallelize the node-level installation procedures for the following reasons:

- The amount of data shared by the *generic deployment logic* (the portion of the NodeApplication implementation that interprets the plan) and

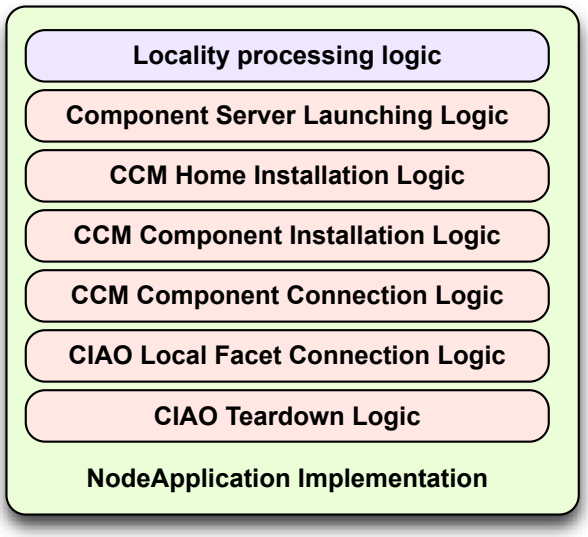

Figure 4: Previous DAnCE NodeApplication Implementation

the *specific deployment logic* (the portion which has specific knowledge of how to manipulate components) made it hard to parallelize their installation in the context of a *single* component server since that data must be modified during installation.

- Groups of components installed to separate application processes were considered as separate deployment sub-tasks, so these groupings were handled sequentially one after the other.

Section 3.3 describes how LE-DAnCE resolves the challenge of overly serialized execution of deployment actions by leveraging asynchronous features of the underlying CORBA middleware to parallelize at the global level. This section also describes how LE-DAnCE's LocalityManager it is used to improve parallelism at the node level.

3. Overcoming Deployment Latency Bottlenecks in LE-DAnCE

This section describes the enhancements we developed for the *Locality-Enhanced Deployment and Configuration Engine* (LE-DAnCE). LE-DAnCE is our reimplementation of the OMG D&C standard that addresses the challenges with DAnCE outlined in Section 2.2. An overview of the architectural differences between DAnCE and LE-DAnCE is shown in Figure 5.

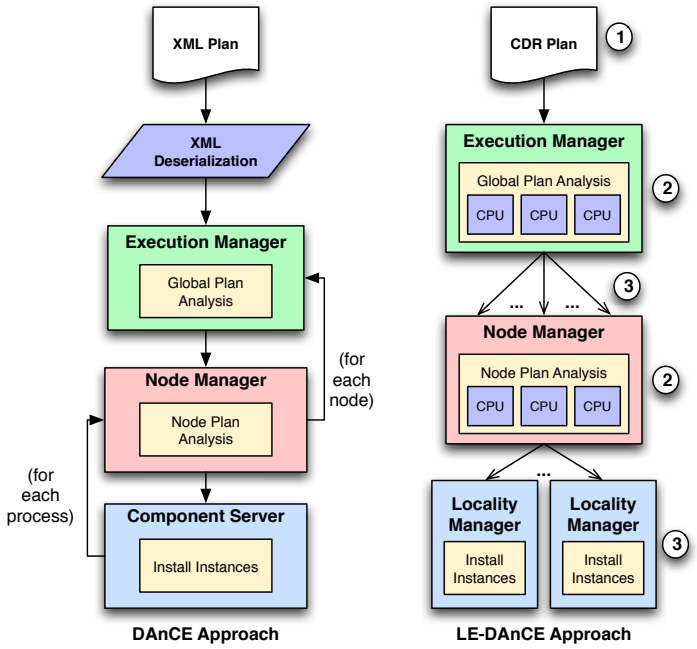

Figure 5: Solution Overview

Section 3.1 describes how we reduced deployment latency arising from the challenge of processing the XML-based deployment descriptors outlined in Section 2.2.1, represented by item 1 in Figure 5. Section 3.2 describes a more efficient plan analysis process to resolve the challenge described in Section 2.2.2, represented by item 2 in the figure. Section 3.3 then describes techniques used by LE-DAnCE to increase deployment and configuration parallelism to overcome the challenge of deployment latency bottlenecks in DAnCE outlined in Section 2.2.3, represented by item 3 in the figure.

3.1. Improving Runtime Plan Loading

There are two general approaches to resolving the challenge of XML parsing outlined in Section 2.2.1.

1. Optimize the XML-to-IDL processing capability. DAnCE uses a vocabulary-specific XML data binding [13] tool called the *XML Schema Compiler* (XSC). XSC reads D&C XML schemas and generates a C++-based interface to XML documents built atop the *Document Object Model* (DOM) XML programming API. DOM is a time/space-intensive approach

since the entire document must first be processed to construct a tree-based representation of the document prior to initiating the XML-to-IDL translation process.

An alternative is to use the *Simple API for XML* (SAX), which uses an event-based processing model to process XML files as they are read from disk. While a SAX-based parser would reduce the time/space spent building the in-memory representation of the XML document, the performance gains are typically too small to invest the substantial development time required to refactor the DAnCE configuration handlers, which serve as a bridge between the XSC generated code and IDL. In particular, a SAX-based approach would still require a substantial amount of text-based processing at runtime. Moreover, deployment plan files have substantial amounts of internal cross-referencing, which would require the processing of entire documents before any actual XML-to-IDL conversion could occur.

2. Preprocess the XML files for latency-critical deployments. This optimization approach converts the deployment plan into its runtime IDL representation and serializes the result to disk using the *Common Data Representation* (CDR) [14] binary format defined by the CORBA specification. The platform-independent CDR binary format used to store the deployment plan on disk is the same format used to transmit the plan over the network at runtime. The advantage of this approach is that it leverages the heavily optimized de-serialization handlers provided by the underlying CORBA implementation to create an in-memory representation of the deployment plan data structure from the on-disk binary stream.

Due to the shortcomings of the first approach outlined above, LE-DAnCE implements the second approach via a tool we developed that leverages the existing DOM-based XML-to-IDL conversion handlers in DAnCE. Deployment plans that have been serialized to XML can be converted automatically offline to an on-disk binary CDR format. Our experimental results, presented in Section 4.1.2, show that LE-DAnCE substantially improves the runtime processing of XML plans compared with DAnCE.

3.2. Improving Runtime Plan Analysis

Section 2.2.2 outlined a significant source of deployment latency: *the plan analysis step that produces node- and process-specific sub plans.* While this analysis step seems "embarrassingly parallel" (*i.e.*, easily separated into parallel sub-problems with little or no effort), accidental complexities of the original DAnCE design made this decomposition impractical due to lock

contention when accessing generated sub-plans. This contention was particularly problematic for large deployment plans, where threads manipulating the sub-plan spent a substantial portion of time in critical sections resizing sequences that store plan meta-data. To address this challenge, we considered two potential approaches. First, we considered a more efficient in-memory representation of plan meta-data. Second, we considered a new algorithmic approach to analyzing the plan to ease parallelization. The pros and cons of these approaches are summarized below.

In-memory representation efficiency. Some difficulties outlined above could be alleviated by using different techniques to represent plan meta-data in memory. For example, pointers/references could be used instead of sequence indices to refer to related data structures, potentially removing the need to carefully rewrite references when plan entities are copied between plans. Likewise, an associative container (such as an STL map) instead of a sequence could store plan objects, thereby increasing the efficiency of inserting plan entities into sub-plans.

While these and other similar options are tempting, they are not desirable for LE-DAnCE since we would be forced to either (1) insert yet another conversion step into the deployment process to translate between this new representation and something that could be marshaled for transmission to other deployment entities or (2) deviate from the D&C standard and make our implementation non-conferment.

New algorithmic approach to runtime analysis. To provide some optimization to the plan analysis process without either of the above undesirable outcomes, we elected to re-evaluate LE-DAnCE's algorithmic approach to (1) minimize the need to resize sequences in the sub-plans during analysis and (2) minimize the need to serialize access to common data structures for multi-threaded implementations.

LE-DAnCE's revised approach to runtime analysis is described in the following phases (which is contrasted with our previous work [9]):

1. **Phase 1: Determine the number of sub-plans to produce.** In this phase, a single thread iterates over all component instances contained in the deployment plan to determine the number of necessary sub-plans. When this operation is performed at the global level, it simply requires a constant time operation per instance. When performed at the local level, it requires that locality constraints (described in Section 2.1.2) be evaluated. Since this phase is potentially time consuming

the results are cached for later use.

2. **Phase 2: Preallocate data structures for sub-plans**. Using information gleaned in phase 1 above, preallocate data structures necessary to assemble sub-plans. As part of this preallocation it is possible to reserve memory for each sequence in the sub-plan data structure to avoid repeated resizing and copying. Statistics are collected in phase 1 to estimate these lengths efficiently.
3. **Phase 3: Assemble node-specific sub-plans.** This phase of the new analysis process is similar to the algorithm described in Section 2.2.2. The main difference is that the cached results of the pre-analysis phase are used to guide the creation of sub-plans. Instead of considering each instance in order (as the original DAnCE implementation did), LE-DAnCE fully constructs one sub-plan at a time, by processing instances on a per-node basis. This approach simplifies parallelizing this phase by dedicating a single thread per sub-plan and eliminates any shared state between threads, except for read-only access to the original plan. It is therefore unnecessary to use any locking mechanism to protect access to the sub-plans.

The approach described above simplified the parallelization of LE-DAnCE by leveraging the OpenMP parallel programming API [15]. OpenMP consists largely of a set of preprocessor directives that are used to decorate loop constructs in the implementation, and describe the type of parallelization, as well as any shared or private state that should exist between threads. These directives are then interpreted by the compiler, which transparently creates the necessary logic to dispatch and support multiple threads.

3.3. Parallelizing Deployment Activities

To support parallelization of deployment activities at the node level, we enhanced the OMG D&C standard by adding a LocalityManager to LE-DAnCE. This LocalityManager unifies all three deployment roles outlined in Section 2.1.1 and functions as a replacement for the component server in Figure 3. More coverage of LE-DAnCE's LocalityManager appears in [16].

Overview of the LE-DAnCE locality manager. The LE-DAnCE node-level architecture (*e.g.*, NodeManager, NodeApplicationManager, and NodeApplication) now functions as a node-constrained version of the global portion of the OMG D&C architecture. Rather than having the NodeApplication directly triggering installation of concrete component instances,

this responsibility is now delegated to LocalityManager instances. The node-level infrastructure performs a second "split" of the plan it receives from the global level by grouping component instances into one or more application processes. The NodeApplication then spawns a number of LocalityManager processes and delegates these "process-constrained" (*i.e.*, containing only components and connections apropos to a single process) plans to each application process in parallel.

Unlike the previous DAnCE NodeApplication implementation, the LE-DAnCE LocalityManager functions as a generic application process that strictly separates concerns between the general deployment logic needed to analyze the plan and the specific deployment logic needed to install and manage the lifecycle of concrete component middleware instances. This separation is achieved using entities called *Instance Installation Handlers*, which provide a well-defined interface for managing the lifecycle of a component instance, including installation, removal, connection, disconnection, and activation. Installation Handlers are also used in the context of the NodeApplication to manage the life-cycle of LocalityManager processes.

Figure 6 shows the startup process for a LocalityManager instance. During the start launch phase of deployment, an Installation Handler hosted in the NodeApplication spawns a LocalityManager process and handles the initial handshake to provide configuration information. The NodeApplication then instructs the LocalityManager to begin deployment by invoking `preparePlan()` and `startLaunch()` hook methods. During this phase, the LocalityManager will examine the plan to determine what instance types must be installed (*e.g.*, container, component, or home). After loading the appropriate Installation Handlers, the LocalityManager will delegate the actual installation process for these instances via the Installation Handler's `install_instance()` method.

Using the Locality Manager to reduce serialized phasing. LE-DAnCE's new LocalityManager and Installation Handlers make it substantially easier to parallelize than DAnCE. Parallelism in both the LocalityManager and NodeApplication is achieved using an entity called the *Deployment Scheduler*, which is shown in Figure 7. The Deployment Scheduler combines the Command pattern [17] and the Active Object pattern [18]. Individual deployment actions (*e.g.*, instance installation, instance connection, *etc.*) are encased inside an Action object, along with any required metadata. Each individual deployment action is an invocation of a method on an Installation Handler, so these actions need not be rewritten for each potential

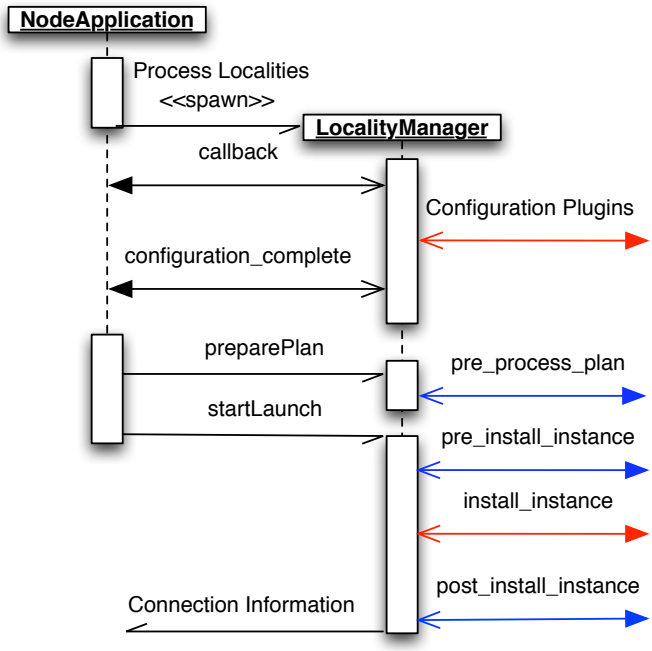

Figure 6: LocalityManager Startup Sequence

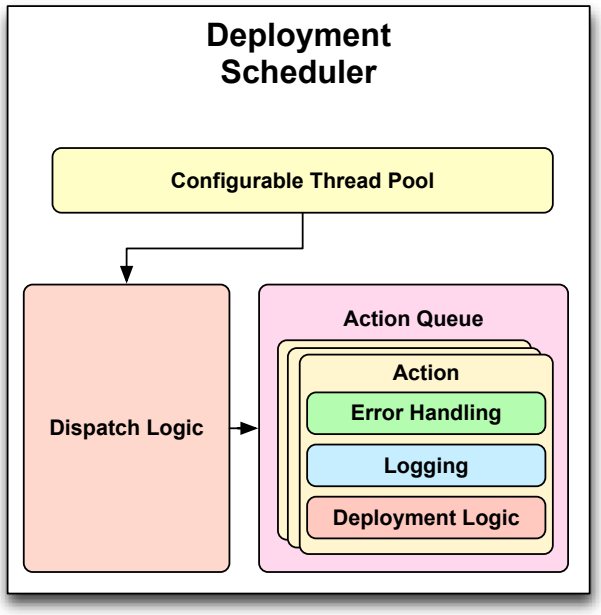

Figure 7: DAnCE Deployment Scheduler

deployment target. Error handling and logging logic is also fully contained within individual actions, further simplifying the LocalityManager.

Individual actions (*e.g.*, install a component or create a connection) are scheduled for execution by a configurable thread pool. This pool can provide user-selected, single-threaded, or multi-threaded behavior, depending on application requirements. This thread pool can also be used to implement more sophisticated scheduling behavior, *e.g.*, a priority-based scheduling algorithm that dynamically reorders the installation of component instances based on metadata present in the plan.

The LocalityManager determines which actions to perform during each particular phase of deployment and creates one Action object for each instruction. These actions are then passed to the deployment scheduler for execution while the main thread of control waits on a completion signal from the Deployment Scheduler. Upon completion, the LocalityManager reaps either return values or error codes from the completed actions and completes the deployment phase.

To provide parallelism between LocalityManager instances on the same node, the LE-DAnCE Deployment Scheduler is also used in the implementa-

tion of the NodeApplication, along with an Installation Handler for LocalityManager processes. Using the Deployment Scheduler at this level helps overcome a significant source of latency whilst conducting node-level deployments. Spawning LocalityManager instances can take a significant amount of time compared to the deployment time required for component instances, so parallelizing this process can achieve significant latency savings when application deployments have many LocalityManager processes per node.

3.4. Planned Enhancements to LE-DAnCE

Section 1 discussed how the ability to redeploy and reconfigure applications in DRE environments at runtime is critical to effective management of QoS properties of applications, including maintaining end-to-end deadlines of applications. This section described how LE-DAnCE contributes to this goal by providing bounded and short redeployment latencies incurred by the D&C toolchain. In particular, after a redeployment decision has been made, LE-DAnCE can quickly and effectively apply it.

There are at least two other key capabilities needed to redeploy and reconfigure enterprise DRE systems effectively that are not yet supported by LE-DAnCE:

- The capacity to accurately identify the actual or potential sources of QoS degradation in the system and make valid decisions about how to reconfigure the system to overcome said degradation.
- The availability of efficient and effective mechanisms provided by the component middleware, operating systems, and hardware to tune QoS parameters.

Each capability is orthogonal to the core functionality of a deployment and configuration engine like LE-DAnCE, which focuses on measuring application performance, identifying the presence of QoS degredations, and determining the cause of such degredations. Moreover, effecting changes in the domain or middleware configuration is application-, middleware, and domain-specific.

Fortunately, however, the architecture of LE-DAnCE—as embodied by its Locality Manager—is amenable to adapting to these application-, middleware-, and domain-specific properties. Our future work on LE-DAnCE will therefore investigate the applicability of customized *Instance Installation Handlers*, as well as another facility present in the Locality Manager called *Deployment Portable Interceptors*, that allow the injection of user-supplied

modifications into the deployment process at runtime. These mechanisms support the redeployment and reconfiguration capabilities mentioned above, as follows:

- Allow online planners (both human and software tools) to inject application- and domain-specific monitors into the deployment process and collect information from which to make intelligent planning decisions.
- Tweak the logic used to implement redeployment plans provided by the online planners. For example, additional QoS configuration steps can be injected into the deployment process and applied to configurations that are outside the scope of a particular component model.

Examples of other middleware that support some aspects of these capabilities include the *Resource Allocation and Control Engine* (RACE) [19] and Swap-CIAO [20] for DRE systems, as well as WS-DIAMOND [21] for self-healing web services.

4. Evaluation of LE-DAnCE Performance Optimizations

This section analyzes the results of experiments we conducted to empirically evaluate LE-DAnCE's ability to overcome the deployment latency bottlenecks we encountered in DAnCE, as described in Section 3. These experiments were conducted in two environments. The first is a highly controlled environment intended to emulate representative DRE systems, as described in Section 4.1.1. The second set of experiments were conducted in a high-performance computing cluster intended to evaluate the performance of LE-DAnCE with a large numbers of processors available and/or a large numbers of computing nodes, as described in Section 4.2.1.

The component applications deployed as part of these experiments include a single component type with one provided port ("facet") and one required port ("receptacle"). The component application itself is intentionally simple, *i.e.*, the component implementations contain minimal application logic to emphasize sources of latency in the deployment framework, rather than latencies arising from implementation details of the application components.

4.1. Experiments in a Controlled DRE System Environment

4.1.1. Overview of the Hardware and Software Testbed

These experiments were conducted in ISISLab (`www.isislab.vanderbilt.edu`), which consists of 4 IBM Blade centers consisting of 14 blades each. In-

dividual blades are equipped with dual 2.8 GHz Intel Xeon CPUs, 1GB of RAM, and 4 Gigabit network interface cards. Connectivity is provided by 6 Cisco 3750G-24TS switches and a single 3750G-48TS switch. ISISLab leverages the Emulab [22] configuration software to provide customized system configurations and virtual network topologies.

For this set of experiments, a deployment of 11 nodes was created with Fedora Core 8 with G++ 4.1.2 used to compile the 1.0 release of the LE-DAnCE and CIAO middleware. The default Linux kernel included with Fedora Core 8 was replaced with a vanilla Linux kernel version 2.6.23 patched with the latest Real-time Preemption patchset [23]. All results reported below are the average of 15 repetitions of the experiment.

4.1.2. Experiment 1: Measuring XML Processing Overhead

Experiment design. A python script was used to generate XML deployment descriptors for applications containing 500, 1,000, 5,000, 10,000, 50,000, and 100,000 component instances equally distributed over 10 nodes. Each component has a single connection to one other component. Each of these XML-based deployment plans was then converted to an in-memory IDL representation using the same methods used during a normal LE-DAnCE deployment.

Experiment results. Table 1 contains the results for the plans described at the beginning of this section and the timing results for the preprocessing described in Section 3.1. This table shows that the time taken to parse an

Table 1: CDP Sizes and Conversion Times

Components	XML Size	CDR Size	Conversion	CDR Read
500	112 KB	48 KB	0.196 Sec	.001982 Sec
1000	304 KB	120 KB	0.323 Sec	.003602 Sec
5000	1.4 MB	608 KB	3.974 Sec	.015747 Sec
10000	2.7 MB	1.2 MB	9.543 Sec	.030199 Sec
50000	13.1 MB	5.8 MB	540.003 Sec	.147542 Sec
100000	27 MB	12 MB	1038.288 Sec	.285286 Sec

XML deployment plan and convert it to IDL can be significant. The plans generated as part of this experiment contain the absolute minimum metadata necessary to successfully deploy the components. If additional configuration information is included—such as attribute initialization (especially involving user-defined complex data types), QoS configurations, or densely connected plans—the amount of XML that must be converted for a given component

count can increase quickly. Experiment 1 showcases the lower bound on the bottleneck—any additional meta-data included in a plan will always be larger than the test case exercised here.

While the on-disk sizes of the various deployment plan files are somewhat interesting, of particular interest are the conversion times from the on-disk format to the in-memory IDL format used by the deployment tools. The results in Table 1 demonstrate that the CDR encoding is an improvement of several orders of magnitude over runtime XML processing. Moreover, the approach described in Section 3.1 exhibits a linear increase in the plan processing time as a function of the number of instances, rather than the exponential behavior shown by runtime XML conversion.

The results outlined in the "Conversion" column are representative of the minimum latencies incurred in the original DAnCE implementation by this stage of the deployment process, as in both the conversion experiment and original DAnCE we are using the same XSC-based transformation logic. LE-DAnCE avoids this latency by enabling the use of binary encoded plans.

4.1.3. Experiment 2: Measuring Application Deployment Latency

Experiment design. To gauge the deployment latency incurred by LE-DAnCE across a wide range of deployment plan sizes, the component application deployments generated for the experiment in Section 4.1.2 were executed. Each plan was executed a total of 25 times, and the reported measurements are represented as the arithmetic mean of all executions.

Experiment results. Table 2 shows the results from experiment 2. These results demonstrate the substantial deployment latency savings ob-

Table 2: Deployment Times (Seconds) for Plans with No Delay

Components	Total Time	Prepare Plan	Start Launch	Finish Launch	Start
1000	1.925	1.761	0.1426	0.0135	0.0061
5000	41.163	40.130	0.2870	0.0255	0.0179
10000	165.623	165.092	0.4576	0.0409	0.0316

tained by the parallel deployment compared to serialized deployments. The plan analysis phase (which occurs during the "Prepare Plan" phase of deployment) in this experimental run still uses the single-threaded analysis process.

The nonlinear growth of time required for this phase makes extremely large deployments infeasible, which is why results for 50,000 and 100,000 components are not shown in Table 2). While deployments of this size were

infeasible, the improvements we later implemented in the runtime plan analysis portion of the deployment process (see Section 3.2) substantially reduce the deployment latency overhead and makes deployments of this size feasible. Results for this improved analysis process are shown in Section 4.2.2.

By combining the results in Section 4.2.2 with those in Table 3 that characterize the latency incurred as a result of this analysis process, we can estimate an upper bound on the total time needed for a completely serial DAnCE deployment. These results can then be compared against the results

Table 3: Split Times (Seconds) for Plans in a DRE Environment

Components	Total Time
1000	1.58
5000	37.195
10000	156.142

contained in Table 2 for parallel LE-DAnCE deployments.

The "Prepare Plan" column from Table 2 contains the time required to first analyze the plan, then dispatch an asynchronous request to each node to perform its local analysis. The difference between this and the times in Table 3 gives the amount of time required to fully analyze and prepare an individual node. It thus provides a reasonable estimate of latency for serial DAnCE execution (before optimizations described in this paper). These estimates appear in Table 4.

Table 4: Estimated Serial DAnCE Deployment for DRE Environments

Components	Single Node Preparation	Est. Total Time
1000	0.181	5.012
5000	2.968	70.179
10000	9.481	256.647

The timing results for the plan preparation phase reveal yet another source of deployment latency. The plan preparation phase includes two important steps, as discussed in Section 2.1.3. The first is a split plan operation to divide the global plan into locality-constrained plans for each node. Next, each node in the deployment performs its own local split to determine how many LocalityManager instances to start, as discussed in Section 3.3. Mitigating this overhead was discussed in Section 3.2 and experimental results appear in Section 4.2.2.

4.1.4. Experiment 3: Measuring the Determinism of Deployment Latency

Experiment design. This experiment characterizes the determinism of the deployment latency performance of LE-DAnCE. To measure this latency we repeatedly deployed the test application with 1,000 components and analyzed the performance metrics over 500 iterations. After each deployment, the testbed was reset and the LE-DAnCE daemons restarted on each node. For this experiment, all DAnCE executable were executed as root and placed in the round robin $SCHED_RR$ scheduling class with the highest possible priority.

Experiment results. The results for experiment 3 are shown in Figure 8. This figure represents the deployment latencies over the course of 500

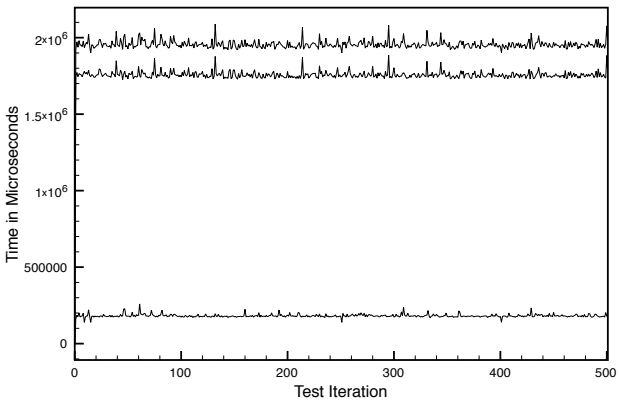

Figure 8: Latency Jitter for 1,000 Component Deployment

iterations for the total deployment latency and the two most time consuming phases: *plan preparation* and *start launch*. The top line of the figure represents the total latency, the middle line represents plan preparation, and the bottom represents the start launch phase (the remaining two phases of deployment took too little time to graph). This figure shows that the LE-DAnCE latency results are relatively stable, indicating the reproducibility of the timeliness in deployment latencies.

Figure 8 also pinpoints the source of most jitter in these results. Most spikes in the total deployment latency are accompanied by spikes in the plan preparation deployment phase. This behavior is likely due to jitter caused by network access, as control messages to individual nodes in this phase contain portions of a large deployment plan and are substantially larger than the

messages for other phases.

Table 5: Deployment Latency Results for 600 iterations of a 1000 component deployment.

	Total Time	Prepare Plan	Start Launch	Finish Launch	Start
Mean	1.9551	1.7569	0.18175	0.01145	0.00451
Maximum	2.0891	1.8871	0.25953	0.01791	0.00575
Minimum	1.8861	1.7261	0.13897	0.01058	0.00417
Std. Deviation	0.0248	0.0216	0.01061	0.00121	0.00017

4.2. Experiments in a Cluster Environment

4.2.1. Overview of the ACCRE Compute Cluster

The Vanderbilt University Advanced Computing Center for Research and Education (ACCRE) is a high performance computing collaboratory built by and for Vanderbilt faculty and research staff. The ACCRE High Performance Computing cluster has about 3,800 processor cores and is growing. Processor cores each have 3 to 6 GB or memory except the 48 GPU nodes which has 12 GB per core (or 96 GB per node). Compute nodes all run 64-bit Linux OS and have 250 GB to 1 TB hard drive and dual copper gigabit Ethernet ports. Resource management, scheduling of jobs, and usage tracking are handled by an integrated scheduling system by Moab/Torque.

4.2.2. Experiment 4: Measuring Plan Analysis Overhead

Experiment Design. This experiment is intended to evaluate the effectiveness of the node-first multi-threading strategy. We used the same technique from Experiment 1 (see Section 4.1.2) to generate plans containing 1,000, 5,000, 10,000, 50,000, and 100,000 component deployment. For the purposes of this experiment, two sets of plans were generated: one set with components distributed equally over 10 nodes, and other set with 100 nodes.

A standalone executable implementing the revised plan analysis module from Section 3.2 was created to time the analysis process. This binary was compiled using the Intel C++ Compiler 2011 with OpenMP enabled. Each plan was converted first to CDR format so that XML plan conversion times would not factor in to the results. For this experiment, one node with 8 processors was allocated for the task in the ACCRE cluster, ensuring that no jobs from other cluster users would interfere in the results. Both single- and multi-threaded results were obtained from the same binary, with the

maximum number of threads available to the OpenMP middleware set at 1 or 8, respectively.

Experiment results. Table 6 summarizes the timing results for experiment 4, showing the total time required, in seconds, to complete the plan analysis process. These values are the arithmetic mean of 25 experimental

Table 6: Plan Analysis Results.

Components	Single 10 Node	Multi 10 Node	Single 100 Node	Multi 100 Node
1000	0.522	0.324	0.322	0.325
5000	5.589	0.946	2.153	0.539
10000	21.366	2.578	8.304	0.968
50000	512.155	54.668	208.02	16.272
100000	2083.505	214.558	858.494	65.132

runs. These results show the substantial performance gains from a parallel implementation of the plan analysis functionality, with improvements of an order of magnitude for all but the smallest of cases. This result is unsurprising given the previously mentioned embarrassingly parallel nature of the analysis problem.

A more interesting aspect of the results shown in Table 6 are the performance differences between the case where we have 10 nodes vs. 100 nodes. If only the results for the single- and multi-threaded 100 node case are combined, it shows a slightly improved performance increase ($\tilde{1}$0x vs. $\tilde{1}$3x). This result is explained by better thread utilization due to the increased number of nodes.

Comparing the single-threaded 10 node case with the single-threaded 100 node case shows that the latter has substantially improved performance for the same number of components. We would expect, however, that these numbers would be similar in the single-threaded case. Since the only difference between these plans is the number of components allocated to each node, this discrepancy shows that some overhead remains from resizing of sub-plan sequences for storing instances and connections, despite the attempt to estimate final sub-plan size in LE-DAnCE.

4.2.3. Experiment 5: Measuring Application Deployment Latency

Experiment Design. The purpose of this experiment is to replicate Experiment 3 in the ACCRE environment. For this experiment, we used the same 1,000 component application used in Experiment 3, using the same methodology of deploying the application, tearing it down, then resetting

the deployment infrastructure on each node. Unlike the previous experiment, however, we were forced to run this experiment as a normal user and without a real-time scheduler in the kernel.

For this experiment, an allocation in ACCRE was made consisting of 11 nodes with two processors per node. Unlike the ISIS environment, this does not guarantee that we get 11 separate, physical hardware nodes—only that we have 22 processors available, allocated in multiples of 2 across several hardware nodes. This allocation model has two important implications: (1) we may be sharing a hardware node with another user and (2) we may have several virtual 'nodes' in the plan allocated to a single physical hardware node. Moreover, we do not know *a priori* what the names or locations of these nodes are.

To accommodate this property of ACCRE, we developed a Python script that interprets the allocation result provided by the ACCRE scheduler and generates a set of initialization scripts and a domain file used by the global deployment infrastructure to map 'virtual' node names in the plan to concrete addresses. The initialization scripts, one for each physical node, start one or more deployment daemons on each node (listening on distinct ports) depending on the number of processors allocated on that node.

Experiment results. The results for experiment 5 are shown in Figure 9. This figure shows the actual deployment latencies for this experiment over the course of 250 iterations. Like Figure 8, we only show the total time (the top line), and the two most time-consuming deployment phases: *prepare plan* (second line) and *start launch* (third line). A summary of our analysis of the results is shown in Table 7.

Table 7: Deployment Latency Results for 250 iterations of a 1000 component deployment.

	Total	PreparePlan	Start Launch	Finish Launch	Start
Mean	0.44856	0.29320	0.14796	0.00477	0.00234
Minimum	0.37429	0.22500	0.13358	0.00390	0.00176
Maximum	0.58585	0.43945	0.20036	0.00872	0.00325
Std. Deviation	0.03247	0.03016	0.00991	0.00061	0.00025

The results in Table 7 show substantially decreased deployment latency over those from Experiment 3 due to the faster hardware available in ACCRE compared to ISISLab. Moreover, comparing the ratio of time spent in the preparation phases vs. the total deployment time shows that the plan preparation phase takes less time than the of the deployment when the par-

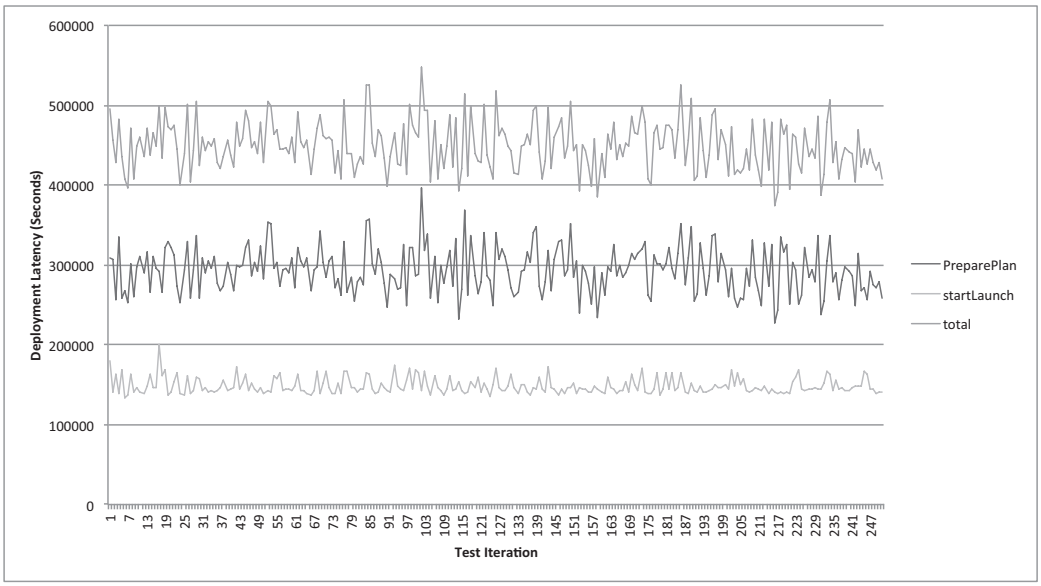

Figure 9: Application Deployment Latency Jitter

allel plan analysis module is used. In Experiment 3, plan preparation was roughly 90 percent of the total deployment time, whereas in Experiment 5 plan preparation only occupies about 65 percent of the total time.

The results also show substantially more jitter as a percentage of the total deployment time. As discussed in Section 4.2.1, this jitter likely stems from characteristics of ACCRE that are beyond our control. In particular, we are not guaranteed exclusive use of the physical nodes on which we are located, so we may compete for limited (1) memory bandwidth from other users on that node and (2) network bandwidth from other users on the cluster.

5. Related Work

This section compares our research on LE-DAnCE with related work in the area of deploying and configuring large-scale distributed applications.

GoDIET [24] is a deployment framework targeted at grid-based distributed applications. GoDIET uses XML metadata defined by a UML model to (1) describe applications and their requirements and (2) wrap applications they wish to deploy inside components based on the Fractal [25] component model. They propose a hierarchical approach to deployment that addresses

deployment latency challenges in grid-based distributed systems. Their approach first partitions nodes present in the domain into two or more segments and then spawns separate deployment processes for those domains. GoDIET is optimized for deployment of applications to grid domains with hundreds of nodes but an extremely limited number of components per node, and performs best when nodes have a mapped NFS mount point in the local file system.

In contrast, LE-DAnCE focuses on applications with high component density, *e.g.*, such deployments often have hundreds or thousands of components per node, which are deployed across tens or hundreds of processes within that node. In addition, applications in DRE system domains often cannot use a shared file system to distribute component implementations due to inherent complexities in the network topology, security concerns, or heterogeneity of the target domain. Moreover, LE-DAnCE automatically coordinates connections between components, whereas the connections must be performed programmatically via GoDIET.

DeployWare [26] is another framework for managing deployments in grid environments based on the Fractal [25] component model. It supports heterogeneous deployments and currently supports middleware intended for the grid environment, such as MPI [27] and GridCCM [28]. Like LE-DAnCE, DeployWare captures deployment metadata in a manner that is relatively agnostic to the eventual deployment target. Unlike LE-DAnCE, however, DeployWare does not capture more complex deployment metadata (such as connection information and QoS metadata) required for DRE systems. Like GoDIET, DeployWare is optimized for delivering relatively few instances/-components to a large number of nodes, and thus uses a similar approach to optimizing deployment latency by partitioning the node into subgroups. In contrast, LE-DAnCE provides a more generic D&C solution by supporting low deployment latencies across a large number of possible hardware and component application sizes and configurations.

The work that is closest to the goals of LE-DAnCE is described in [29], which uses hierarchical separation of concerns to provide concurrent—and hence faster—deployments. This work differs from LE-DAnCE since it does not focus on a standard (*e.g.*, the OMG D&C specification), but rather some general concepts of deployment and configuration. In contrast, LE-DAnCE is aimed at providing a standards-based solution to enhance broader applicability, while also optimizing performance and minimizing/bounding latency.

The work presented in [30] seeks to find deployment solutions in dynamic

environments. The focus is on deploying a hierarchical component (which is an assembly of components treated as a single unit), while ensuring the deployment of individual monolithic units do not violate architectural constraints of the platform and the network before deploying that component. While the goal of their deployment solution is similar to that of LE-DAnCE, their approach differs in its focus on the deployment of hierarchical components (*i.e.*, amalgamations of primitive components with other hierarchical components), which they represent at runtime via "membrane" components that act as proxies for internal primitive components. In contrast, the metadata present in the D&C specification supports such hierarchies at design time, but is flattened by LE-DAnCE for runtime deployment to avoid the overhead of additional component instances implemented as membranes at a per-process level.

CaDAnCE [31] was an earlier effort we conducted to reduce latency and increase deterministic behavior of DRE system D&C operations. It focused on simultaneous deployment of multiple applications from a single deployment plan in which certain components are shared among multiple sub-applications. CaDAnCE demonstrated that dependencies among these sub-applications can yield deployment-order priority inversions where low-priority applications may complete their deployments ahead of a mission-critical sub-application. CaDAnCE solved this problem using priority-inheritance to ensure predictable deployment for high-priority sub-applications that are deployed simultaneously with other low-priority sub-applications and with which they share components. The goals and approach of CaDAnCE are orthogonal to the goals of LE-DAnCE since CaDAnCE focuses on re-ordering component deployment and installation of particular components within the context of a single application, whereas LE-DAnCE focuses on reducing overall deployment latency for an entire application.

6. Concluding Remarks and Lessons Learned

This paper described enhancements to the OMG *Deployment and Configuration* (D&C) specification that support QoS needs of component-based enterprise DRE systems. We first described sources of deployment latency overhead that degraded the responsiveness of the *Deployment And Configuration Engine* (DAnCE), which was our original open-source implementation of the D&C specification. We then explained how our *Locality-Enhanced Deployment and Configuration Engine* (LE-DAnCE) improved DAnCE to

alleviate key sources of deployment latency overhead associated with XML pre-processing and *LocalityManager* architecture.

The effectiveness of LE-DAnCE's *LocalityManager* architecture was then evaluated empirically in two contexts: a highly controlled DRE environment and a high-performance computing cluster. The results showed that LE-DAnCE can manage large-scale deployments of component applications, scaling to both large numbers of components and nodes, as follows:

1. **Off-line processing of XML-based deployment plans**. LE-DAnCE minimizes deployment latency by pre-processing XML-based deployment plans and pre-serializing them into the in-memory format required by the deployment infrastructure and storing that to disk, which is both more compact and requires no runtime processing to initiate deployment.
2. **Parallel processing of deployment plans**. LE-DAnCE can take full advantage of multi-core platforms to significantly reduce latency stemming from deployment time processing and analysis of deployment plans.
3. **Asynchronous transport-level processing**. LE-DAnCE leverages transport-level asychrony and deployment scheduling to substantially reduce serialized phasing of deployment activities between application, node, and global-level deployment entities during distributed component application deployments.

We learned the following lessons from developing and evaluating LE-DAnCE:

• **Serialized phasing is a major source of deployment latency.** Serialized phasing, which occurs when the deployment infrastructure waits for an application or node to complete deployment before proceeding to the next, significantly increases the time required to completely deploy distributed component applications. The LE-DAnCE improvements as described in this paper substantially reduce the impact of serialized phasing and its contributions to deployment latency.

• **Split plan processing incurs significant deployment latency.** The results presented in Section 4 showed that the plan preparation phase of deployment is a large source of deployment latency, due in large part to inefficiency in the LE-DAnCE "split plan" algorithm. As part of the work conducted for this article, we alleviated this inefficiency by optimizing this algorithm and leveraging multi-core/processor architectures to reduce latency

stemming from this deployment phase. This process can be further optimized, as demonstrated by the results in Section 4.2.2. Our future work is investigating how to apply a similar pre-deployment analysis approach used to address latency due to XML processing and split the plan before deployment to reduce runtime deployment latency.

• **The startLaunch operation is a significant source of jitter.** The start launch phase of deployment produces the largest amount of jitter in the LE-DAnCE deployment process. Prior experiments [32] conducted on DAnCE showed this jitter stemmed from the dynamic loading of component implementations at runtime, which can be alleviated by directly compiling component implementations and plan metadata into the deployment infrastructure. While this approach reduces jitter and latency, it is also invasive to the D&C implementation, hard to maintain, and removes too much flexibility from the D&C toolchain. Our future work will reduce this jitter by applying static configuration optimizations [33] to both the component middleware (CIAO) and the plug-in architecture of LE-DAnCE.

CIAO and LE-DAnCE are available in open-source form from `download.dre.vanderbilt.edu`.

7. Acknowledgments

This work was supported in part by NSF CAREER 0845789 and CNS 0915976, and a contract from Northrop Grumman and AFRL GUTS. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, AFRL, or NGC.

This work was conducted in part using the resources of the Advanced Computing Center for Research and Education at Vanderbilt University, Nashville, TN.

References

[1] C. Esposito, D. Cotroneo, Resilient and timely event dissemination in publish/subscribe middleware., IJARAS 1 (2010) 1–20.

[2] P. Lardieri, J. Balasubramanian, D. C. Schmidt, G. Thaker, A. Gokhale, T. Damiano, A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems, Journal of

Systems and Software: Special Issue on Dynamic Resource Management in Distributed Real-time Systems 80 (2007) 984–996.

[3] D. Suri, A. Howell, D. C. Schmidt, G. Biswas, J. Kinnebrew, W. Otte, N. Shankaran, A Multi-agent Architecture for Smart Sensing in the NASA Sensor Web, in: Proceedings of the 2007 IEEE Aerospace Conference, Big Sky, Montana.

[4] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, D. C. Schmidt, Integrated adaptive qos management in middleware: A case study, Real-Time Syst. 29 (2005) 101–130.

[5] W. R. Otte, J. S. Kinnebrew, D. C. Schmidt, G. Biswas, D. Suri, Application of Middleware and Agent Technologies to a Representative Sensor Network, in: Proceedings of the Eighth Annual NASA Earth Science Technology Conference, University of Maryland.

[6] Deployment and Configuration of Component-based Distributed Applications, v4.0, OMG, Document formal/2006-04-02 edition, 2006.

[7] Object Management Group, The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model, Object Management Group, OMG Document formal/2008-01-08 edition, 2008.

[8] 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 (CD 2005), Grenoble, France, pp. 67–82.

[9] W. R. Otte, A. Gokhale, D. C. Schmidt, Predictable Deployment in Component-based Enterprise Distributed Real-time and Embedded Systems, in: Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering, CBSE '11, ACM, New York, NY, USA, 2011, pp. 21–30.

[10] A. Gokhale, B. Natarajan, D. C. Schmidt, A. Nechypurenko, J. Gray, N. Wang, S. Neema, T. Bapty, J. Parsons, CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications, in: Proceedings of the OOPSLA 2002 Workshop

on Generative Techniques in the Context of Model Driven Architecture, ACM, Seattle, WA.

[11] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, J. Parsons, The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging, in: Proceedings of the Middleware 2000 Conference, ACM/IFIP.

[12] ObjectWeb Consortium, CARDAMOM - An Enterprise Middleware for Building Mission and Safety Critical Applications, `cardamom.objectweb.org`, 2006.

[13] J. White, B. Kolpackov, B. Natarajan, D. C. Schmidt, Reducing Application Code Complexity with Vocabulary-specific XML language Bindings, in: ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference.

[14] Object Management Group, The Common Object Request Broker: Architecture and Specification Version 3.1, Part 2: CORBA Interoperability, Object Management Group, OMG Document formal/2008-01-07 edition, 2008.

[15] OpenMP, OpenMP Home Page, www.openmp.org, ????

[16] W. R. Otte, D. C. Schmidt, A. Gokhale, Towards an Adaptive Deployment and Configuration Framework for Component-based Distributed Systems, in: Proceedings of the 9th Workshop on Adaptive and Reflective Middleware (ARM '10), Bengaluru, India.

[17] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[18] 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.

[19] N. Shankaran, 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.

[20] J. Balasubramanian, B. Natarajan, D. C. Schmidt, A. Gokhale, G. Deng, J. Parsons, Middleware Support for Dynamic Component Updating, in: International Symposium on Distributed Objects and Applications (DOA 2005), Agia Napa, Cyprus.

[21] R. Ben Halima, K. Drira, M. Jmaiel, A qos-oriented reconfigurable middleware for self-healing web services, in: Web Services, 2008. ICWS '08. IEEE International Conference on, pp. 104 –111.

[22] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar, An integrated experimental environment for distributed systems and networks, in: Proc. of the Fifth Symposium on Operating Systems Design and Implementation, USENIX Association, Boston, MA, pp. 255–270.

[23] I. Molnar, Linux with Real-time Pre-emption Patches, `http://www.kernel.org/pub/linux/kernel/projects/rt/`, 2006.

[24] M. Toure, P. Stolf, D. Hagimont, L. Broto, Large scale deployment, in: Autonomic and Autonomous Systems (ICAS), 2010 Sixth International Conference on, pp. 78 –83.

[25] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, An open component model and its support in Java, in: Component-Based Software Engineering, pp. 7–22.

[26] A. Flissi, J. Dubus, N. Dolet, P. Merle, Deploying on the grid with deployware, in: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, USA, 2008, pp. 177–184.

[27] Argonne National Laboratory, The Message Passing Interface (MPI) standard, www-unix.mcs.anl.gov/mpi/, ????

[28] C. Pérez, T. Priol, A. Ribes, A Parallel CORBA Component Model for Numerical Code Coupling, in: M. Parashar (Ed.), Grid Computing-GRID 2002, Springer Berlin / Heidelberg, PARIS research group

IRISA/INRIA Campus de Beaulieu 35042 Rennes Cedex France, 2002, pp. 88–99. 10.1007/3-540-36133-2_9.

[29] V. Quéma, R. Balter, L. Bellissard, D. Féliot, A. Freyssinet, S. Lacourte, Asynchronous, hierarchical, and scalable deployment of component-based applications, in: Proceedings of Second International Working Conference on Component Deployment, Edinburgh, UK, pp. 50–64.

[30] D. Hoareau, Y. Mahéo, Middleware support for the deployment of ubiquitous software components, Personal and Ubiquitous Computing 12 (2008) 167–178.

[31] G. Deng, D. C. Schmidt, A. Gokhale, Cadance: A criticality-aware deployment and configuration engine, in: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, IEEE Computer Society, Washington, DC, USA, 2008, pp. 317–321.

[32] V. Subramonian, G. Deng, C. Gill, J. Balasubramanian, L.-J. Shen, W. Otte, D. C. Schmidt, A. Gokhale, N. Wang, The Design and Performance of Component Middleware for QoS-enabled Deployment and Conguration of DRE Systems, Elsevier Journal of Systems and Software, Special Issue Component-Based Software Engineering of Trustworthy Embedded Systems 80 (2007) 668–677.

[33] V. Subramonian, L.-J. Shen, C. Gill, N. Wang, The Design and Performance of Configurable Component Middleware for Distributed Real-Time and Embedded Systems, in: RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04), IEEE Computer Society, Lisbon, Portugal, 2004, pp. 252–261.