Predictable Deployment in Component-based Enterprise Distributed Real-time and Embedded Systems*

William R. Otte, Aniruddha Gokhale, and Douglas C. Schmidt
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235
{wotte, gokhale, schmidt}@dre.vanderbilt.edu

ABSTRACT

Component-based middleware, such as the Lightweight CORBA Component Model, are increasingly used to implement large-scale 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.

This paper makes three contributions to the study of predictable deployment latencies in DRE systems. First, we describe OMG's Deployment and Configuration (D&C) specification for component-based systems and discuss how conventional implementations of this standard 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. Finally, 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. Our results show LE-DAnCE's optimizations provide a bounded deployment latency of less than 2 seconds with 4 percent jitter.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*components,deployment*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Design, Experimentation, Algorithms

*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.

CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.

Keywords

component-based real-time systems, predictable deployment

1. INTRODUCTION

Component-based software engineering techniques are increasingly applied to develop large-scale *distributed real-time and embedded* (DRE) systems, such as air-traffic management [6], shipboard computing environments [12], and distributed sensor webs [25]. These domains are often characterized as "open" since applications in these domains 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 [9].

To adapt to changing environments and operational requirements, it may be necessary to change the deployment and configuration characteristics of these DRE systems at runtime. 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 these domains, it is important that any changes to DRE system deployment and configuration occur as quickly and predictably as possible, *i.e.*, DRE systems expect short and bounded deployment latencies.

Not only are timely and dependable runtime deployment and configuration changes essential in DRE systems; even initial application startup time can be an important metric. For example, in extremely 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 [18]. In such environments, deployments must be fast and time-bounded.

To support these requirements, the efficiency and QoS 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 [17] for component-based applications, have emerged in recent years.[1] The OMG D&C specification provides comprehensive development, packaging, and deployment frameworks for a wide range of component middleware.

[1]Although originally developed for the *CORBA Component Model* (CCM) [14], the OMG D&C specification is defined via a UML metamodel that is applicable to many other component models.

In the OMG D&C specification, deployment instructions are delivered to the deployment infrastructure via a *component deployment plan* (CDP), 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 predictable manner. In this paper, 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 CDP is provided to the deployment infrastructure to the time at which all deployment instructions have been executed and the system activated.

This paper motivates and describes architectural enhancements we made to the OMG D&C specification to achieve predictable deployment latencies for large-scale DRE systems. Our solution is called the *Locality-Enhanced Deployment and Configuration Engine* (LE-DAnCE), which extends our earlier *Deployment and Configuration Engine* (DAnCE) [4]. We developed DAnCE with the sole aim of cleanly separating concerns defined by the OMG D&C specification and demonstrating its feasibility. After applying DAnCE to a range of representative DRE systems [12, 18], 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. 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. This paper explains how LE-DAnCE overcomes these limitations.

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 DAnCE; Section 3 describes how we addressed these sources of overhead, 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 then describes how an implementation of this specification called the *Deployment and Configuration Engine* (DAnCE) [4] supports the separation of concerns espoused in the D&C specification. We expose key sources of overhead that impact deployment latencies in DRE systems and pinpoint 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 application development lifecycle, as well as runtime interfaces used for packaging and planning. Below we focus on the interfaces, metadata, and architecture used for runtime deployment and configuration.

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 a set of global entities used to coordinate deployment and a set of 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:

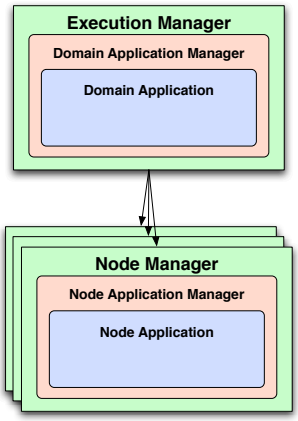

Figure 1: OMG D&C Architectural Overview and Separation of Concerns

- **Manager.** The Manager role, found at the global level as the *ExecutionManager* and at the node-level as the *NodeManager*, corresponds to a singleton daemon that manages all deployment entities in a single context. The Manager serves as the entry point for all deployment activity and serves as a factory for implementations of the *ApplicationManager* role.
- **ApplicationManager.** The ApplicationManager serves as a lifecycle manager for running instances of a component application. The global entity is known as the *DomainApplicationManager* and the node-level entity is known as the *NodeApplicationManager*. Each ApplicationManager represents exactly one component application and is used to initiate deployment and teardown of the application. This role serves as a factory for implementations of the *Application* role.
- **Application.** This role represents a deployed instance of a component application, and is used to finalize the configuration of the associated component instances and to begin execution of the deployed component application. At the global level, this entity is called the *DomainApplication*, while the node-level entity is called the *NodeApplication*.

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 created by the specification is intended for use as (1) 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 (2) directives that describe the configuration and deployment used by the runtime infrastructure. Most entities in the D&C metadata contains a section where arbitrary 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 is used to describe everything from basic configuration information (such as shared library entrypoints 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 is broadly 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 (*i.e.*, shared libraries), and create packages that contain both metadata and implementations that may be installed into the target environment. Domain descriptors are used by hardware administrators to describe the 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 into physical reality through the creation of the third type of metadata supported by the OMG D&C standard: the *component deployment plan* (CDP), which contains the following information:

- **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.* entrypoints 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.

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 that is codified in 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, all of which are described below:

1. **Plan preparation.** In this phase, a CDP 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 only information for each 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 CDP. 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 discusses the sources of overheads that impact deployment latencies in the context of the architecture defined by the OMG D&C specification. We use our existing DAnCE [4] OMG D&C implementation as a vehicle to demonstrate these sources of overhead. The major sources of latency overhead 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 sub-optimal implementations.

2.2.1 Challenge 1: 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 CDP is serialized on disk in a XML file whose structure is described by an XML Schema defined by the OMG D&C standard. This XML document format for CDP files presents significant advantages by providing a simple interchange format between modeling tools [10], is easy to generate and manipulate using widely available XML modules for popular programming languages, and enables simple modification and data mining by text processing tools, such as perl, grep, sed, and awk.

Processing these CDP files during deployment and even runtime, however, can lead to substantial deployment latency costs, as shown in Section 4.2. This increased latency stems from the following sources:

- XML CDP 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, so it must first be converted into the native OMG *Interface Definition Language* (IDL) format used by the runtime interfaces of the deployment framework.

In many enterprise DRE systems, component deployments that number in the thousands are not uncommon, and component instances in these domains will exhibit a high degree of connectivity. Given the structure of CDPs outlined in Section 2.1.2, both these factors contribute to large plans. While the above latency source is most immediately applicable to initial application deployment, it can also present a significant problem during potential re-deployment activities at application runtime that involve significant changes to the application configuration. While CDP 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 the application continues to meet its stringent QoS and end-to-end deadlines during online modifications.

Section 3.1 describes how LE-DAnCE resolves Challenge 1 by pre-processing large deployment plans offline into a portable binary representation.

2.2.2 Challenge 2: 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 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 [2].

To minimize initial implementation complexity, we used synchronous invocation in an (admittedly shortsighted) design choice in an earlier implementation of DAnCE. This global synchronicity did not cause problems for relatively small deployments (less than 100 components). As the number of both nodes and instances assigned to those nodes begin to scale up, however, this global/local serialization imposes a substantial cost in deployment latency.

This serialization problem, however, is not limited only to the global/local task dispatching and 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 (in this case, CCM), instead leaving such an interface as an implementation detail. Early versions of DAnCE directly instantiated the CCM container and components directly in the address space of the NodeApplication. To alleviate the resulting tedious and error-prone deployment logic, we later separated the CCM container into a separate process. In DAnCE, the D&C architecture was implemented using three processes, as shown in Figure 2.

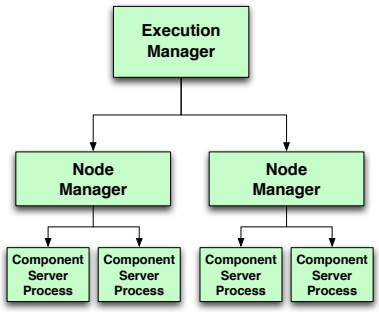

Figure 2: Simplified DAnCE Architecture

The ExecutionManager and NodeManager processes instantiate their associated ApplicationManager and Application instances in their address space. When the NodeApplication installs concrete component instances it spawns one (or more) separate component server processes as needed. The component server processes use an interface derived from an older version of the CCM specification that allows the NodeApplication to individually instantiate containers and component instances. This approach is similar to that taken by CARDAMOM [16], which is another CCM imple-

mentation tailored for enterprise DRE systems, such as air-traffic management systems.

While the DAnCE architecture shown in Figure 2 improved upon the original implementation that collocated all CCM entities in NodeApplication address space, it was still problematic with respect to parallelization. Rather than performing only some processing and delegating the remainder of the concrete deployment logic to the component server process, the DAnCE NodeApplication implementation instead integrates all logic necessary for installing, configuring, and connecting instances directly, as shown in Figure 3.

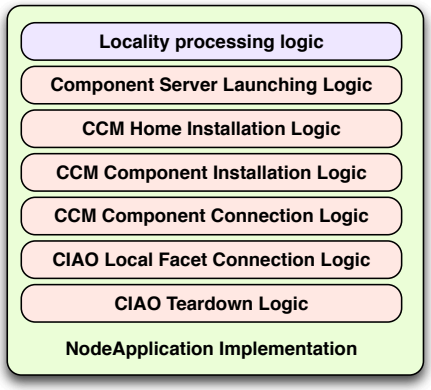

Figure 3: DAnCE NodeApplication Implementation

This tight integration made it hard to optimize the node-level installation process 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 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.
- Since groups of components installed to separate component servers can be considered separate deployment sub-tasks, these groupings could be also parallelized.

3. OVERCOMING DEPLOYMENT LATENCY BOTTLENECKS IN LE-DANCE

This section describes the enhancements we developed for *Locality Enhanced DAnCE* (LE-DAnCE), which is a new implementation of the OMG D&C standard that addresses the challenges outlined in Section 2.2. 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. Section 3.2 then introduces techniques LE-DAnCE uses to increase deployment and configuration parallelism to overcome the challenge of deployment latency bottlenecks in DAnCE outlined in Section 2.2.2.

3.1 Improving Runtime Plan Processing

There are two 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 [28] 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. In general, DOM is a time/space-intensive approach since the entire document must first be processed to fully construct a tree-like representation of the document before the XML-to-IDL translation process can occur.

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 may be too small to invest the substantial development time required to re-factor 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 runtime text-based processing. Moreover, CDP files have substantial amounts of internal cross-referencing, which would require the entire document be processed before any actual XML-to-IDL conversion could occur.

2. Pre-process the XML files for latency-critical deployments. This optimization approach (used by LE-DAnCE) is accomplished via a tool we developed that leverages the existing DOM-based XML-to-IDL conversion handlers in DAnCE to (1) convert the CDP into its runtime IDL representation and (2) serialize the result to disk using the *Common Data Representation* (CDR) [15] binary format defined by the CORBA specification. This platform-independent binary format used to store the CDP 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 (TAO) to create an in-memory representation of the CDP data structure from the on-disk binary stream.

3.2 Parallelizing Deployment Activity

To support parallelized dispatch of deployment activity at the node level, we enhanced the OMG D&C standard by adding a LocalityManager to LE-DAnCE. The LocalityManager unifies all three deployment roles outlined in Section 2.1.1, and functions as a replacement for the component server in Figure 2. An overview of LE-DAnCE's LocalityManager appears in [19].

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 causing the installation of concrete component instances, this responsibility is now entirely 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 component servers. 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 process in parallel.

Unlike the previous DAnCE NodeApplication implemen-

tation, the LE-DAnCE LocalityManager functions as a generic application server that maintains a strict separation of concerns between the general deployment logic required to analyze the plan and the specific deployment logic required to actually 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 4 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()`. During this process, 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 `install_instance()` method on the Installation Handler.

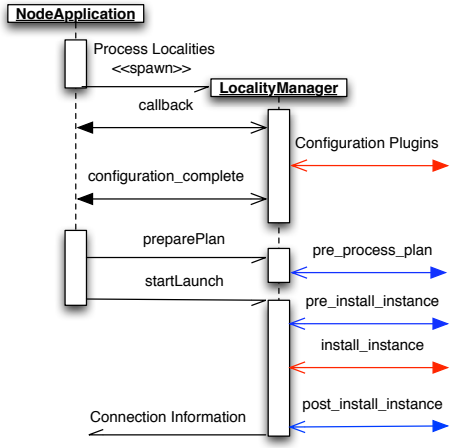

Figure 4: LocalityManager Startup Sequence

The new LE-DAnCE LocalityManager and Installation Handlers make it substantially easier to parallelize than in DAnCE. Parallelism in both the LocalityManager and NodeApplication is achieved using an entity called the *Deployment Scheduler*, which is shown in Figure 5. The Deployment Scheduler combines the Command pattern [8] and the Active Object pattern [22]. 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 re-written for each potential 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, which can provide user-selected single-threaded

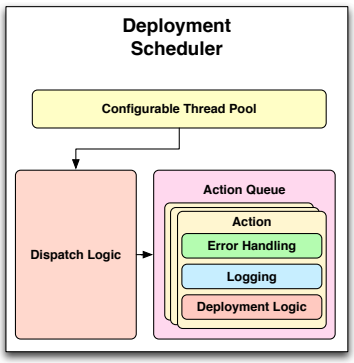

Figure 5: DAnCE Deployment Scheduler

or multi-threaded behavior, depending on the requirements of the application. This thread pool could also be used to implement more sophisticated scheduling behavior. For example, it might be desirable to implement a priority-based scheduling algorithm that dynamically reorders the installation of component instances based on metadata present in the plan.

During deployment, the LocalityManager determines which actions to perform during each particular phase 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 implementation of the NodeApplication, along with an Installation Handler for LocalityManager processes. Using the Deployment Scheduler at this level also helps to 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.

4. EXPERIMENTAL RESULTS

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.

4.1 Overview of 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. Individual 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 [27] configuration software to provide customized system configurations and virtual network topologies.

For the following 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 DAnCE and CIAO middleware frameworks. 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 Pre-Emption patchset [13]. The component application deployed as part of these tests included 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 due to the, rather than latencies due to implementation details of the application components.

All results reported below are the average of 15 repetitions of the experiment.

4.2 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 pre-processing described in Section 3.1.

This table shows that the time taken to parse an XML deployment plan and convert it to IDL can be significant. It is worth noting that 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. In this case, we are attempting to showcase 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 CDP 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.

4.3 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 4.2 were executed. Each plan was executed a total of 25 times, and the reported measurements represent the arithmetic mean of all executions.

Experiment results. Table 2 shows the results from this experiment. These results demonstrate the substantial deployment latency savings of parallel deployment compared to serialized deployments. If we disregard the plan preparation timings, the remaining phases of the deployment would require ten times the amount taken by the remaining phases (*e.g.,* the 1,000 component deployment would require at least 1.622 seconds additional time).

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 3.2. The nonlinear growth of the time required for this phase shown in Table 2 makes extremely large deployments infeasible, which is the reason why results for 50,000 and 100,000 components are not included.

4.4 Experiment 3: Measuring the Predictability of Deployment Latency

Experiment design. This experiment characterizes the predictability of the deployment latency performance of LE-DAnCE. To accomplish this, 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 this experiment are shown in Figure 6.

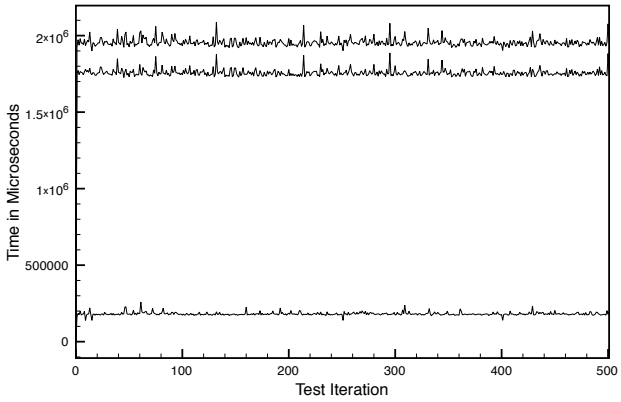

Figure 6: Latency Jitter for 1000 Component Deployment

This figure represents the deployment latencies over the course of 500 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.

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

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

Of particular interest in Figure 6 is identifying the source of most jitter in these results. Most spikes in the total deployment latency are also accompanied by spikes in the plan preparation deployment phase. This is likely due to jitter due to 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.

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 [26] is a deployment framework intended for 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 [3] 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 will often have hundreds or thousands of components per node, often deployed across tens or hundreds of processes within that node. In addition, applications in DRE 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 [7] is another framework for managing deployments in grid environments based on the Fractal [3] component model. It supports heterogeneous deployments and currently supports middleware intended for the grid environment, such as MPI [1] and GridCCM [20]. 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 comes close to the goals of LE-DAnCE is described in [21], 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 standardized solution to enhance applicability while also optimizing performance and minimizing/bounding latency.

The work presented in [11] 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 [5] was an earlier effort we conducted to reduce latency and increase predictability of DRE system D&C operations. It focused on simultaneous deployment of multi-

Table 3: 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

ple 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 the OMG *Deployment and Configuration* (D&C) specification for component-based applications and explored sources of deployment latency overhead that degraded the responsiveness of the *Deployment And Configuration Engine* (DAnCE), which is an 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 preprocessing and *LocalityManager* architecture. The effectiveness of the LE-DAnCE *LocalityManager* architecture was then empirically evaluated by (1) deploying a number of high component-density applications to demonstrate the performance of the toolchain as the number of components grows and (2) measuring the predictability of these performance results by repeatedly deploying the same setup on a 1,000 component deployment.

The following lessons were learned conducting this research:

Split Plan process 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. To alleviate this inefficiency our future work will determine if this algorithm can further be optimized or investigate ways that the plan can be split 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 [23] conducted on DAnCE showed this jitter stemmed from the dynamic loading of component implementations at runtime and can be alleviated by directly compiling component implementations and plan metadata into the deployment infrastructure [24]. While this approach reduces jitter and latency, it is also invasive to the D&C implementation, hard to maintain, and removes much of the flexibility from the D&C toolchain. Our future work is exploring more flexible ways to reduce this jitter via work that builds on these previous efforts at static configuration of not only the component middleware (CIAO), but also the plug-in architecture of LE-DAnCE.

CIAO and LE-DAnCE are open-source software and all work described in this paper is available in the latest version which can be obtained from `download.dre.vanderbilt.edu`.

7. REFERENCES

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

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

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in Java. In *Component-Based Software Engineering*, pages 7–22, 2004.

[4] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, pages 67–82, Grenoble, France, Nov. 2005.

[5] G. Deng, D. C. Schmidt, and A. Gokhale. CaDANCE: A Criticality-Aware Deployment And Configuration Engine. In *Proceedings of the 11th International Symposium on Object/Component/Service-oriented Real-time Distributed Computing)*, pages 317–321, Orlando, Florida, May 2008. IEEE.

[6] C. Esposito and D. Cotroneo. Resilient and timely event dissemination in publish/subscribe middleware. *International Journal of Adaptive, Resilient and Autonomic Systems*, 1:1 – 20, 2010.

[7] A. Flissi, J. Dubus, N. Dolet, and P. Merle. Deploying on the grid with deployware. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 177–184, Washington, DC, USA, 2008. IEEE Computer Society.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable*

Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[9] C. Gill, J. Gossett, J. Loyall, D. Schmidt, D. Corman, R. Schantz, and M. Atighetchi. Integrated Adaptive QoS Management in Middleware: A Case Study. In *Proceedings of the 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, Toronto, Canada, May 2004. IEEE.

[10] A. Gokhale, B. Natarajan, D. C. Schmidt, A. Nechypurenko, J. Gray, N. Wang, S. Neema, T. Bapty, and J. Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, Nov. 2002. ACM.

[11] D. Hoareau and Y. Mahéo. Middleware support for the deployment of ubiquitous software components. *Personal and Ubiquitous Computing*, 12(2):167–178, 2008.

[12] P. Lardieri, J. Balasubramanian, D. C. Schmidt, G. Thaker, A. Gokhale, and 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(7):984–996, July 2007.

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

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

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

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

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

[18] W. R. Otte, J. S. Kinnebrew, D. C. Schmidt, G. Biswas, and 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, June 2008.

[19] W. R. Otte, D. C. Schmidt, and 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)*, Bengarulu, India, Nov. 2010.

[20] C. Pérez, T. Priol, and A. Ribes. A Parallel CORBA Component Model for Numerical Code Coupling. In M. Parashar, editor, *Grid Computing-GRID 2002*, pages 88–99. Springer Berlin / Heidelberg, PARIS research group IRISA/INRIA Campus de Beaulieu 35042 Rennes Cedex France, 2002. 10.1007/3-540-36133-2_9.

[21] V. Quéma, R. Balter, L. Bellissard, D. Féliot, A. Freyssinet, and S. Lacourte. Asynchronous, hierarchical, and scalable deployment of component-based applications. In *Proceedings of Second International Working Conference on Component Deployment*, pages 50–64, Edinburgh, UK, May 2004.

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

[23] V. Subramonian, G. Deng, C. Gill, J. Balasubramanian, L.-J. Shen, W. Otte, D. C. Schmidt, A. Gokhale, and 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(5):668–677, Mar. 2007.

[24] V. Subramonian, L.-J. Shen, C. Gill, and 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)*, pages 252–261, Lisbon, Portugal, 2004. IEEE Computer Society.

[25] D. Suri, A. Howell, D. C. Schmidt, G. Biswas, J. Kinnebrew, W. Otte, and 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, Mar. 2007.

[26] M. Toure, P. Stolf, D. Hagimont, and L. Broto. Large scale deployment. In *Autonomic and Autonomous Systems (ICAS), 2010 Sixth International Conference on*, pages 78 –83, Mar. 2010.

[27] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[28] J. White, B. Kolpackov, B. Natarajan, and 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*, 2005.