ADAPTIVE DEPLOYMENT AND CONFIGURATION FRAMEWORKS FOR COMPONENT-BASED APPLICATIONS

By

William R. Otte

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December, 2011

Nashville, Tennessee

Approved:

Professor Douglas C. Schmidt

Professor Aniruddha Gokhale

Professor Janos Sztipanovits

Professor Gabor Karsai

Professor Jeff Gray

To my mother, father, and wife Lindsay, for their unconditional love, encouragement, and support

ACKNOWLEDGMENTS

First and foremost, I would like to extend my profound gratitude to my co-advisors through my tenure as a graduate student, Dr. Douglas C. Schmidt and Dr. Aniruddha Gokhale. I joined the DOC Group after an undergraduate class with Dr. Schmidt piqued my interest, and was immediately involved in several high impact and interesting projects. Under Dr. Schmidt's leadership, the DOC Group, in addition to mentoring and technical education, provided me with opportunities not available to many other graduate students to travel and meet and work directly with sponsors. His insight into emerging trends and belief in producing practical solutions to real-world problems has been essential in developing my research vision. When Dr. Schmidt took a leave of absence, Dr. Gokhale took over and worked very closely with me to develop publish my ideas, and finally to develop my proposal and dissertation. His tireless efforts to provide me guidance and insight into this process and to review these documents were essential and greatly appreciated.

I would like to thank the remainder of my committee members, Dr. Gabor Karsai, Dr. Janos Sztipanvits, and Dr. Gabor Karsai for serving on my qualifying examination and dissertation committees. In particular, the insightful questions asked by Dr. Karsai during both examinations helped me to refine my ideas and view them in a new light. I am especially grateful to Dr. Gray for the time he took to provide detailed feedback on my proposal and dissertation — his insightful comments helped me to significantly improve the quality of this dissertation.

Over the years, my work has been supported by a number of agencies and companies, who provided the funding necessary to do my work as well as challenging motivating scenarios. First, I would like to thank Patrick Lardieri, Gautam Thacker, and Tom Damiano of Lockheed-Martin Advanced Technology Laboratories. I worked with these three gentlemen on the DARPA ARMS project, and they gave me an opportunity to join them for a summer as an intern. Second, I am especially grateful to Dipa Suri of Lockheed-Martin

Advanced Technology Center with whom I had the opportunity to work for several years on what eventually became the MACRO project. Finally, I would like to thank my sponsors at the Northrup-Grumman corporation — in particular Mark Hayman, and Trent Nadeau who provided key technical and conceptual insights that eventually led to what I feel is the most important contribution of this dissertation, the Locality Manager.

I would like to thank Johnny Willemsen and Martin Corino of Remedy IT who, over the years, have provided valuable mentoring and technical insights. Their advice and support were instrumental in developing the ideas that eventually became the LocalityManager and DDS4CIAO.

Of course, none of this would have happened had I taken a different course more than a decade ago and pursued a different course of study. While as a child I was always fascinated by computers (mostly the computer games), I fell in love with computer science at the age of 17 while a senior at Jesuit College Preparatory School in Dallas, Texas. There, a teacher by the name of Peter Billingham convinced me that I should take his brand new AP Computer Science course. In this class, I quickly excelled and outpaced most of the other students. Peter did his best to keep up, and took the time to really challenge me, and eventually convinced me that I'd be a fool to go to law school.

While at ISIS, I've had the opportunity to work with and become friends with a number of incredibly talented individuals. First and foremost, I'd like to thank Jeff Parsons whose assistance and advice has been an integral part of my work during my tenure with the DOC Group. Balachandran Natarajan, a former ISIS engineer and DOC Group member provided much of my early technical education and mentored me during my first years with the group. Dr. Jaiganesh Balasubramanian, a graduate of the DOC Group, helped me to improve my research ideas and helped me understand how to be a successful graduate student. Finally, Dr. Gan Deng and Dr. Nanbor Wang, who developed the initial implementations of the deployment tools and component middleware that formed the basis of my research.

Last, but certainly under no circumstances least, my family for their unconditional love, encouragement, and support. My mother, Dr. Hollon Meaders and father, Calvin C. Otte always provided the means, opportunity, and encouragement for me to pursue my interests. My wife, Lindsay Gail Johnson, for putting up with a stressed out graduate student who didn’t always have time to clean or do laundry.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 New Demands on Distributed, Real-time and Embedded Systems

Component-Based Software Engineering (CBSE) [27] is increasingly used as a paradigm for developing applications in both the enterprise [1] and embedded, severely resource-constrained applications [76]. CBSE facilitates systematic software reuse by encouraging developers to create black box components that interact with each other and their environment through well-defined interfaces. This allows applications of greater complexity to be composed from smaller units of functionality, *e.g.*, commercial off-the-shelf components, and preexisting applications. These applications are packaged along with descriptive and configuration meta-data, and made available for deployment into a production environment.

Managing deployment and configuration of component-based applications in general is an extremely complex and challenging problem for the following reasons. First, there may be complex requirements and relationships amongst individual components. Components may depend on one another for proper operation, or specifically require or exclude particular versions. If these relationships are not described and enforced, component applications may fail to deploy properly; even worse, malfunction in subtle and pernicious ways. Second, a component might expose configuration hooks that change its behavior, and the deployment system must manage and apply any required configuration information. Furthermore, several components in a deployment may have related configuration properties, and the deployment infrastructure should ensure that these properties remain consistent across an entire application. Third, in the case of enterprise systems, components must be installed and have their connection and activation managed on remote hosts.

Distributed, real-time and embedded (DRE) systems are an emerging class of applications which share properties of both enterprise systems and severely resource-constrained

systems. DRE applications are similar to the enterprise kind in that they are distributed across a large domain. Like embedded systems, DRE applications are often mission-critical and carry stringent safety, reliability, and quality of service (QoS) requirements. Deployment of DRE systems, in addition to the complexities described above, carry their own set of unique challenges. First, applications in the DRE domains may have particular dependencies on the target environment, such as particular hardware/software (*e.g.*, GPS, sensors, actuators, particular operating systems). Second, the deployment infrastructure must contend with strict resource requirements in environments with finite resources (*e.g.*, CPU, RAM, network bandwidth).

The deployment infrastructure must ensure that these resources are present and available in an environment that is changing due to a number of factors including loss or damage to nodes, changing availability of resources such as network bandwidth, and contention from other applications. Third, these applications often have changing goals and QoS requirements in response to new situations in the environment, and the deployment infrastructure must be able to react and modify the deployed application accordingly. Finally, real-world applications do not live in a vacuum and in many cases be homogeneous with respect to their distribution middleware, and must often interface with either legacy systems or applications from different vendors. Such heterogeneity cannot be foreseen by the component middleware developers, so a mechanism to inject new communication mechanisms is desireable.

I.2 Overview of Research Needs

In this section we list the research needs that arise in the context of deployment and configuration of different varieties of DRE systems. These new directions of research are organized according to research focus areas that make up this dissertation.

I.2.1 Focus Area 1: Deployment and Configuration for Resource-Constrained Systems

Resource-constrained systems present unique challenges to both component middleware and deployment and configuration infrastructure not found in other DRE systems. An example of such a resource-constrained system includes the domain of sensor webs. Sensor webs are large-scale, networked systems often made up of heterogeneous computing platforms that include commodity servers and DRE systems. Unfortunately, the configuration and operation of individual sensor webs are often performed in an *ad hoc* manner, which impedes adding new sensors, updating and modifying their software, and reconfiguring them to accommodate evolving conditions and changing science needs. These challenges include standardized execution of low-level hardware-dependent actions and on-going data tasks, automated provisioning of agents for heterogeneous field hardware, and minimizing deployment infrastructure overhead.

Traditional heavyweight component middleware, while an attractive solution for the reasons outlined above, can be inappropriate for all elements of the sensor web software due to extremely limited resources. First, severely constrained available memory limits the number of components that may be deployed to each node. Second, heavyweight components may take a comparatively long time to deploy and could cause the sensor web to not react quickly enough to changes in the environment or mission and violate QoS guarantees.

While other approaches to component-based development for extremely embedded applications, such as Programming in the Many (PitM) [41], effectively meet the stringent footprint requirements, they lack the interoperability and rich ecosystem of CORBA services provided by CCM. Moreover, it limits the communication mechanisms to simple message passing, lacking the rich datatype descriptions and interface communication provided by a more expressive component model. Chapter II describes the integration of more advanced component middleware and deployment and configuration techniques to a representative sensor web called SEAMONSTER. This chapter examines in detail the

challenges inherent in applying component middleware to embedded sensor web platforms and presents the Action/Effector framework, an extremely lightweight component model for encapsulating low-level hardware dependent tasks.

I.2.2 Focus Area 2: Heterogeneous and Adaptable Deployment and Configuration Frameworks

Production-quality, large-scale distributed computing systems often cannot be limited to a single component model, particularly if they must integrate and interface with legacy systems. While it is possible to use multiple individual deployment frameworks to deploy and configure applications, this approach can complicate the planning process (*i.e.*, assigning instances to nodes, ensuring that sufficient resources exist, performing static verification, etc.), thereby leading to problems during system integration. These problems stem from potentially incompatible tooling, meta-data formats, and problems coordinating the activity of disparate deployment infrastructures.

In addition to the need to potentially support multiple deployment targets, our experience with the development of large-scale DRE systems described earlier in this section has demonstrated that applications and/or users may have wildly different expectations of the behavior of the D&C infrastructure based on (1) domain requirements (*e.g.*, safety criticality or domain requirements), (2) the stage of the development process (*e.g.*, development/-testing vs. deployment/operation). Examples of such behavior adaptation include customized error handling semantics, differing models of application liveness monitoring, or customized discovery services for connecting disparate portions of the application.

Some frameworks for heterogeneous deployment do already exist, but are inappropriate for DRE systems. One such tool, DeployWare [21], provides a way to create "personalities" that allow for the deployment of multiple component models. DeployWare, however, provides anemic support for meta-data that can be shared throughout the lifecycle of a component application, which can make it difficult to use in larger projects in which multiple,

independent teams must collaborate. Another tool, ADAGE [38] provides heterogeneous deployment for grid environments. It, however, is inappropriate because it is not possible to capture specific component/node pairings, which are necessary in DRE systems to ensure that components are properly allocated to domain hardware in order to provide QoS guarantees. Neither tool provides a standard mechanism to customize the behavior of the deployment toolchain.

Chapter III describes the LocalityManager, a novel framework for creating a heterogeneous and adaptable deployment infrastructure, based on the OMG Deployment and Configuration specification which provides robust meta-data facilities. The LocalityManager provides mechanisms whereby the *specific deployment logic, i.e.*, the target component model for deployment, may be augmented through plug-ins called Installation Handlers that are loaded at run-time. *Generic deployment logic*, portions of the deployment process that are component middleware agnostic, may also be customized through the use of Deployment Portable Interceptors.

I.2.3 Focus Area 3: Efficient and Deterministic Deployment Latency

Domains that feature DRE applications 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 [24]. 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 [64]. In such environments, deployments must be fast and time-bounded.

Other contemporary deployment infrastructure tooling for large-scale domains, such as GoDIET [77] or DeployWare [21], are optmized for computational grids, with relatively homogeneous hardware and networks, as well as relatively few component instances on large numbers of nodes. Some DRE domains such as shipboard computing environments, however, have very high component density on relatively few nodes. Chapter IV describes in detail sources of deployment latency in DRE component deployment infrastructure, and steps taken in the LocalityManager to overcome these difficulties.

I.2.4 Focus Area 4: Design Approaches to Extensible Component Middleware

Existing and planned enterprise DRE systems must increasingly support large data spaces generated by thousands of collaborating nodes, sensors, and actuators that must exchange information to detect changes in the operational environment, make sense of that information, and effect changes. These capabilities require scalable publish/subscribe (pub-/sub) semantics [19] that support a range of QoS properties, that control properties, such as liveliness, latency, deadlines, timing, and reliability. Unfortunately, the conventional component technologies used to develop enterprise DRE systems either do not provide first class support for pub/sub semantics or do so in an ineffective manner that is not scalable and does not support real-time QoS properties.

A standardized, QoS-enabled pub/sub technology called the OMG Data Distribution Service (DDS) [53] has emerged as a promising pub/sub technology to support the requirements of enterprise DRE systems. DDS includes standard QoS policies and mechanisms

to handle data (de)marshaling, node discovery and connection, and configuration. Middleware based on the DDS standard has been applied successfully in mission-critical domains, such as air traffic management systems [18] and tactical information systems [28].

Integration of new distribution middleware and features into component models is not a straightforward process. First, interfaces provided by the component container and standardized generated code may lack the expressivity necessary to fully take advantage of the new middleware. Second, proper integration often requires deep knowledge of both the new distribution middleware and the component container implementation — developers with such dual expertise may not exist. Chapter V describes DDS4CIAO, a framework that combines key advantages of the DDS middleware, such as low latency communication and extensive QoS policy support, with the strengths of a mature component model, such as simplified application composition and automatic deployment and configuration. This integration is accomplished via entities called connectors that live outside the container.

I.3 Dissertation Organization

The remainder of this dissertation describes each of the above focus areas and research challenges incurred within each area in more detail and describes how this dissertation resolves these challenges. Chapter II outlines the application of D&C frameworks to resource-constrained sensor webs. Next, Chapter III describes the LocalityManager, a framework for heterogeneous and adaptive deployment and configuration infrastructure. Chapter IV presents the design principles and substantial empirical evidence that illustrates performance optimizations to deployment latency in the LocalityManager. Chapter V describes DDS4CIAO, a connector-based integration of DDS into Lightweight CCM that makes it possible to realize the missing pub/sub capabilities within CCM without breaking the original component programming model. Finally, Chapter VII describes future research directions and concluding remarks.

CHAPTER II

DEPLOYMENT AND CONFIGURATION FOR RESOURCE CONSTRAINED SENSOR WEBS

A variety of sensor webs [13] can now provide data in near real-time to help scientists study and predict weather, natural disasters, and climate change. Modern sensor webs provide capabilities for information to be gathered from sensors around the globe and quickly transmitted to local or remote servers where significant computational resources are available for model building, data analysis, and prediction. With the appropriate infrastructure, these systems can facilitate the real-time collection and analysis of sensor data even under changing environmental conditions and multiple concurrent science objectives.

Sensor webs are large-scale, networked systems often made up of heterogeneous computing platforms that include commodity servers and distributed real-time embedded (DRE) systems. Unfortunately, the configuration and operation of individual sensor webs are often performed in an *ad hoc* manner, which impedes adding new sensors, updating and modifying their software, and reconfiguring them to accommodate evolving conditions and changing science needs.

Like other DRE systems, the field subsystems of sensor webs can benefit from recent advances in middleware infrastructures. The use of *quality-of-service (QoS)-enabled component middleware* helps automate remoting, life-cycle management, system resource management, deployment, and configuration in DRE systems. QoS-enabled component middleware supports explicit configuration of QoS aspects (*e.g.*, priority and threading models), and provides many desirable real-time features (*e.g.*, priority propagation, scheduling services, and explicit binding of network connections). In integrated, adaptive sensor webs,

QoS-enabled component middleware helps address the large, heterogeneous set of sensor assets and computational resources that must be coordinated and managed to address weather, climate change, and disaster prediction/management problems.

Sensor web hardware is also increasingly configurable and must operate in *open* environments where operating conditions, workload, resource availability, and connectivity cannot be accurately characterized *a priori*. Our previous work described the design of the *Multi-agent Architecture for Coordinated Responsive Observations* (MACRO) [75], which provides a QoS-enabled component middleware platform that automates many system configuration and management tasks for sensor web applications, including dynamic system management and autonomous operation of configurable sensor webs in open DRE system environments. This chapter addresses new distributed deployment challenges that result from applying the MACRO platform to the *South East Alaska MOnitoring Network for Science, Telecommunications, Education, and Research* (SEAMONSTER) [20], which is a representative sensor web for monitoring glacial change and watershed effects.

The remainder of this chapter is organized as follows: Section II.1 summarizes adaptive sensor web challenges in SEAMONSTER that include standardized execution of low-level hardware-dependent actions and on-going data tasks, automated provisioning of agents for heterogeneous field hardware, and minimizing deployment infrastructure overhead; Section II.2 describes how we addressed these challenges by extending MACRO to include an Action/Effector framework that standardizes the execution of lightweight actions, automates the provisioning of MACRO agents, and optimizes the footprint of the underlying QoS-enabled component middleware; Section II.3 empirically evaluates how these extensions address deployment challenges; Section II.4 compares MACRO with related work.

II.1 Motivation

II.1.1 Overview of SEAMONSTER

SEAMONSTER is a glacier and watershed sensor web at the University of Alaska Southeast (UAS) [20]. This sensor web monitors and collects data regarding glacier dynamics and mass balance, watershed hydrology, coastal marine ecology, and human impact/hazards in and around the Lemon Creek watershed and Lemon Glacier. The collected data is used to study the correlations between glacier velocity, glacial lake formation and drainage, watershed hydrology, and temperature variation.

The SEAMONSTER sensor web, as shown in Figure II.1, includes sensors and weatherized computer platforms that are deployed on the glacier and throughout the watershed to collect data of scientific interest. The data collected by the sensors is relayed via wireless networks to a cluster of servers that filter, correlate, and analyze the data. These data collection and processing applications are being transitioned to run on top of a QoS-enabled component middleware platform consisting of the *Component-Integrated ACE ORB* (CIAO) [80], which is open-source, QoS-enabled, component middleware that implements the OMG Lightweight CORBA Component Model (CCM) [50] and Deployment and Configuration [49] specifications.

Figure II.1: SEAMONSTER field sensors and UAS servers

II.1.2 Distributed Deployment and Adaptation Challenges in SEAMONSTER

Effective deployment of data collection and filtering applications on SEAMONSTER field hardware and dynamic adaptation to changing environmental conditions and resource availability present significant software challenges for efficient operation of SEAMONSTER. While SEAMONSTER servers provide significant computational resources, the field hardware is computationally constrained. The server-based MACRO agents perform extensive planning and scheduling to provide direction and coordination of tasks to be performed by the computationally limited field resources. In the field, the limited computational resources require software solutions with small footprints and low computational complexity.

Field nodes in a sensor web often have a large number of observable phenomena in their area of interest. The type, duration, and frequency of observation of these phenomena may change over time, based on changes in the environment, occurrence of transient events in the environment, and changing goals and objectives in the science mission of the sensor web. Moreover, limited power, processing capability, storage, and network bandwidth constrain the ability of these nodes to continually perform observations at the desired frequency and fidelity. Dynamic changes in environmental conditions coupled with limited resource availability requires individual nodes of the sensor web to rapidly revise current operations and future plans to make the best use of their resources.

To handle dynamic changes effectively, sensor web nodes must be capable of goal-driven, functional adaptation. Moreover, they must have the capability to adapt the local system in light of resource constraints and fluctuations throughout the sensor web to maintain efficient and correct operation of the overall system. Prior work [35] describes how MACRO addresses these challenges by combining the planning and resource management services of its server agents with the template plan schemas of its field agents. This chapter extends our prior work by focusing on the following unexplored challenges associated with

providing a flexible deployment infrastructure to support system management and dynamic adaptation of the SEAMONSTER field nodes.

II.1.3 Challenge 1: Standardized Execution of Planned Low-Level Actions and Data Tasks

Most tasks performed by MACRO agents on the SEAMONSTER server cluster involve on-going data processing and analysis that are implemented by components selected and configured during planning/scheduling. A scheduled plan for the deployment and operation of these configured components is passed to a resource management service, which allocates them to individual server nodes and adjusts configuration settings and operating system parameters to handle fluctuations in resource usage and availability. The resource management service employs the deployment infrastructure to coordinate the deployment, configuration, connection, and execution of the specified components. This provides a standardized, flexible system for implementing tasks as configured components.

Data collection and transmission tasks on field nodes are implemented as components for the same reasons as data processing tasks on the servers. However, many of the other activities that MACRO agents plan and perform on field nodes consist of low-level, hardware-dependent actions that execute only briefly to configure sensors or the power management hardware subsystem. Implementing these short-duration "actions" as components would incur disproportionate amounts of overhead for their deployment and execution than for data processing "tasks" that typically execute over longer periods of time and have to transmit data streams to other components. Given the limited computational resources available on field nodes, the overhead for implementing brief, low-level actions as components is unacceptable.

Lower levels of granularity are needed for efficient execution of many planned activities on field nodes. Agents could implement these actions directly, but this would require hard-coding of hardware-dependent actions into each field agent. Alternatively, grouping these

actions into larger pre-planned sets of actions executing as a component would proportionally reduce the overhead. However, this would negatively impact maintainability through duplication of action code segments and constrain the available options for planning. Instead, a standardized deployment and execution framework, such as that provided by a middleware infrastructure for components, but with lower overhead, would greatly enhance the maintainability of the system and simplify initial system development. Section II.2.3.1 describes how such a framework has been designed and incorporated in MACRO to address this challenge.

II.1.4 Challenge 2: Automated Agent Provisioning for a Variety of Field Hardware

Field nodes in a sensor web may have a large number of possible configurations, due to a variety of sensors, software, and situations that they may be tasked to observe and appropriately react. Consequently, the agents that manage these nodes must be as flexible as possible. Hard-coding available tasks into agent code requires that new versions of each agent be created as nodes add new responsibilities or hardware. The solution developed to address the previous challenge should include integration with the deployment infrastructure to download and load at runtime appropriate action implementations. Section II.2.3.2 describes how the deployment infrastructure may be leveraged to dynamically provision agents with available, context-specific actions at deployment time.

II.1.5 Challenge 3: Minimizing Deployment Infrastructure Overhead

The SEAMONSTER sensor web, described in Section II.1.1, includes many field nodes operating with extremely limited computational resources. SEAMONSTER includes two types of computational platforms for field nodes [64]:

• **Primary Microservers.** These units are weatherized single board computers (SBC) that are designed to have very limited power consumption and precise control over the

power consumption of the SBC and attached devices. The SBC is a commercial off-the--shelf (COTS) product that has a 200 MHz low-power ARM processor with 64 MB of built-in RAM.

• **Adjunct Microservers.** These units are re-purposed COTS Linksys NSLU-2 network attached storage devices that are essentially inexpensive SBCs. These computers consist of a 133 Mhz (with simple hardware modifications possible to reach 266 MHz) ARM processor with 32 MB of built-in RAM. These units provide a low-cost alternative to using Primary Microservers for some field nodes. However, they lack power control capabilities and have even more limited computational power primarily due to the minimal amount of RAM.

Each platform presents an environment where the resident footprint of the middleware infrastructure and component implementations is critically important. Excessive footprint will at best cause excessive memory swapping to occur, significantly degrading performance and shortening the life of attached flash drives, and at worst cause deployment failure due to exhaustion of memory, as happened occasionally during initial trials of MACRO software in the SEAMONSTER testbed. Section II.2.3.3 describes initial efforts to reduce the footprint of the middleware.

II.1.6 Challenge 4: Local power management with sleep/wake cycles

SEAMONSTER's need for power management is motivated by limited availability of power, due to variable weather conditions limiting the ability to recharge the batteries. The available power is often insufficient for continuous operation of the processor, requiring the system to periodically power down completely. Moreover, to protect against "wedging" (which is a situation where the operating system becomes unresponsive), it is useful to periodically hard-reset the microservers, which are difficult to physically access in the field. When a microserver returns from one of these sleep/wake cycles, *i.e.*, when the boot process

completes, local agents and applications must be correctly re-deployed and connections between nodes must be correctly re-established. Section II.2.3.4

II.2 Minimizing Infrastructure Overhead in MACRO

This section explains how MACRO addresses the challenges described in Section II.1. This chapter begins with an overview of the agent-based system developed in our previous work, along with a description of its middleware infrastructure. The new MACRO Action/Effector framework is introduced, which addresses the deployment infrastructure challenges encountered in the SEAMONSTER project.

II.2.1 Overview of MACRO

The MACRO platform provides a powerful computational infrastructure for enabling the deployment, configuration, and operation of large-scale sensor webs that are composed of many constituent sensor webs. Figure II.2 shows how MACRO supports intelligent autonomy via agents at the following two levels of abstraction:

- **Mission level**, where agents interact with users to allocate high-level science tasks to sensor webs and coordinate scheduled plans to achieve these goals, and
- **Resource level**, where local server and field agents achieve mission goals through functional adaptation of a sensor web in light of current environmental conditions and resource availability.

The work presented in this chapter focuses on the resource level of MACRO, which is applicable to individual sensor webs, such as SEAMONSTER.

System adaptation for current conditions and science goals, described as a set of desired data products and results, is directed by MACRO server-based agents with functional knowledge of the sensor web system and available software components and actions. MACRO server-based agents employ novel services, such as the *Spreading Activation Partial Order Planner* (SA-POP) [34] and the *Resource Allocation and Control Engine*

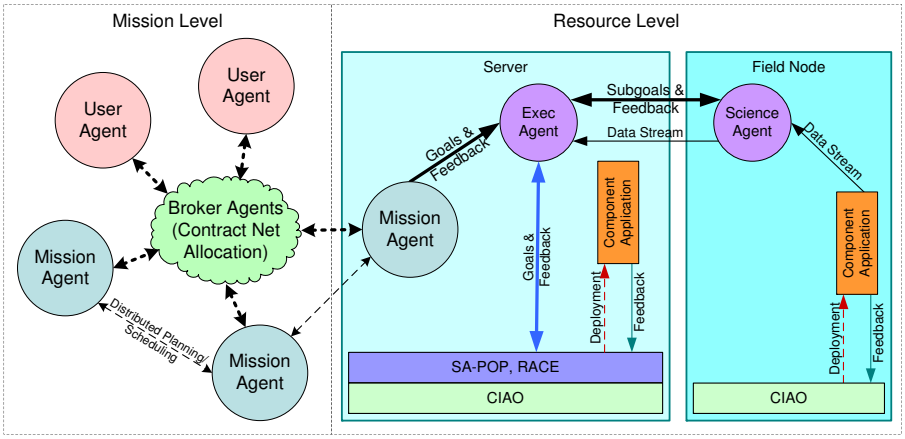

Figure II.2: MACRO Agent Architecture

(RACE) [70]. These agents use the SA-POP service to (1) decompose goals into subgoals that are achieved at the server or by individual field nodes and (2) plan/schedule for their achievement.

With information from field agents about current conditions and local activities, SA-POP produces scheduled, high expected utility plans to achieve an optimized set of current goals. These scheduled plans are also broken into subplans by SA-POP. These subplans describe (1) the selection/configuration of server-based software components, which are allocated and managed by the RACE service on the servers, and (2) hardware-dependent actions on individual field nodes, as well as additional component deployments.

Although the sub-plans generated by SA-POP on the servers can provide an important starting point for deployments and actions on the field nodes, changing local conditions may invalidate those plans or require modification to them for effective, rapid reaction to environmental phenomena and changing resource availability. Since local field agents have limited computational resources, extensive planning and scheduling, such as that provided by SA-POP, is not possible for rapid reaction to local changes. Instead, field agents use a set of template plan schemas that cover a range of conditions and local subgoals to which they are applicable.

Server-based agents provide the field agents with the current set of local subgoals to

pursue and suggested schema instantiations corresponding to the sub-plans produced by SA-POP. The task of the field agent is therefore the simpler choice of an appropriate set of schemas to instantiate as local conditions evolve. The extensive planning/scheduling performed by MACRO server agents using SA-POP—together with the choice of plan schemas to instantiate by MACRO field agents—provide effective system adaptation to achieve science goals in light of changing environmental conditions and resource availability.

The implementation of agents in MACRO is based on the CIAO [80] QoS-enabled component middleware (described in Section II.2.2 to ensure interoperability across heterogeneous computing platforms, reduce development costs, and improve overall robustness and scalability. The agents operate on the CIAO middleware to ensure that a diverse set of

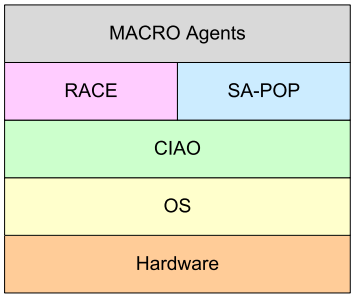

Figure II.3: The MACRO Architecture

science objectives can be met, as shown in Figure II.3. This architecture helps facilitate real-time, adaptive data acquisition, analysis, fusion, and distribution.

II.2.2 Overview of MACRO's QoS-enabled Component Middleware

The MACRO middleware infrastructure is based on the CORBA Component Model (CCM) [52], which is an extension to the Common Request Broker Architecture (CORBA) [48] that supports Component Based Software Engineering. CCM enhances re-usability by allowing developers to focus only on application business logic, abstracting away the details

of communication and configuration. Components interact with one another only through well-defined ports, which include *facets* (provided interfaces), *receptacles* (required interfaces), and *event sources and sinks* (asynchronous publish/subscribe transport).

The CCM middleware used in MACRO is the *Component Integrated ACE ORB* (CIAO) [78]. CIAO is a QoS-enabled implementation of the Lightweight CCM (LWCCM) [51] specification built on top of *The ACE ORB* (TAO). CIAO provides a clear separation of concerns between *configuration logic*, specified at deployment time via XML-based meta-data, and *business logic*.

CIAO's deployment and configuration capabilities are provided by the *Deployment and Configuration of Component Based Systems* (DnC) [59] specification, which was created by the OMG in response to the need for generic and standard mechanisms for deploying component-based applications. The DnC standard includes both a *data model* (*i.e.*, descriptions of components, component compositions, target domains, and associated configuration meta-data) and a *runtime model* (*i.e.*, a set of interfaces used to manage application life-cycles).

The DnC *runtime model* in CIAO is implemented by the *Deployment And Configuration Engine* (DAnCE) [14]. DAnCE is a set of daemons executing in the *domain*, which is the collection of nodes and communication methods that comprise the target environment. Important elements of the runtime model are shown in Figure II.4 and include:

• **Node Manager**, which is a daemon that runs on all nodes in the domain and is responsible for deploying, configuring, and managing all components deployed to that node. This daemon also supports the monitors necessary to report the resource status on the node to the MACRO agents. Each node in the sensor web will have a running Node Manager.

• **Execution Manager**, which is a daemon that coordinates the activities of all *Node Managers* in a given domain. This daemon is the primary point of control for the life-cycle of all component applications. Primary microservers with direct connections to the SEAMONSTER server cluster will have Execution Managers.

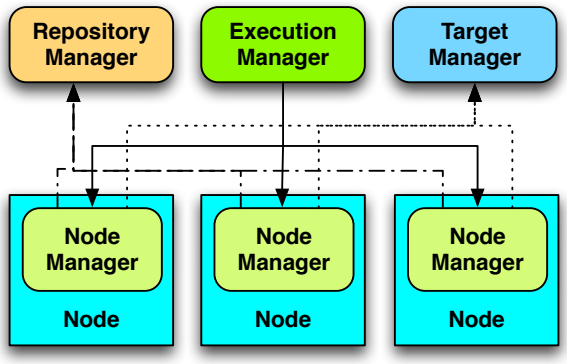

Figure II.4: DAnCE Daemons

• **Target Manager**, which is a daemon that collates and reports resource availability in a given domain. Information is collected from resource monitors installed in individual *Node Managers*. Like the *Execution Manager*, this daemon will run on primary microservers with direct connections to the servers.

• **Repository Manager**, which is a daemon that maintains a collection of component meta-data and binary implementations. Individual *Node Managers* may contact nearby repositories to download binaries for components they are tasked to deploy, and MACRO agents may query the repository for information about components available for deployment. An instance of the *Repository Manager* will run on the primary server for use by the MACRO server agents and server deployments. Another instance will reside on primary microservers with direct connections to the SEAMONSTER server cluster for use by nodes in the field.

II.2.3 Applying MACRO to Address SEAMONSTER Challenges

The remainder of this section explains how MACRO applies and enhances the CIAO and DAnCE middleware described above addresses the sensor web challenges identified in Section II.1.2.

```
1  struct Property {
2    string name;
3    any value;
4  };
5  typedef sequence<Property> Properties;
6  valuetype Action_Info {
7    public string id;
8    public Properties resource_requirements;
9    public Properties init_arguments;
10   public Properties exec_arguments;
11   public Properties reference_requirements;
12 }
```

Listing II.1: Action_Info Data Structure Example.

II.2.3.1 Addressing Challenge 1: MACRO's Action/Effector Framework

MACRO's Action/Effector framework has been developed to provide a standardized mechanism that has two primary benefits for implementing short-lived, lightweight "actions," as opposed to on-going "tasks" implemented as components. First, it allows the MACRO agents with their SA-POP planning service and plan schemas to use a common vocabulary for describing preconditions, dependencies, and effects of individual actions, as well as resource requirements of the associated action implementations. Second, it provides a clear separation of concerns between invoking the action and the business logic of the action, similar to that of components, *i.e.*, it provides a mechanism that agents can use to execute a set of actions without knowledge at compile or link time of the implementation of those actions.

Action meta-data.

Listing II.1 describes the `Action_Info` data structure which allows an action to provide meta-data about itself to the system/agents.

This meta-data describes properties (*e.g.*, a unique identifier, argument identifiers and

```
1 local interface Action {
2   readonly attribute Action_Info info;
3   void initialize (in ObjSeq references);
4   void execute (in any arguments,
5                 out any result);
6   void release ();
7 };
```

Listing II.2: Action Interface

types, return value identifier and type) and requirements (*e.g.* CPU and memory requirements, hardware/sensor resources, and component or object references). This data structure is implemented as a CORBA valuetype, which will leave open the possibility for derivation though inheritance should additional fields need to be added later without breaking backwards compatibility with the interfaces described below.

Action interface. Listing II.2 describes the interface for the Action itself.

This interface provides a vehicle for provision of meta-data, and operations to manage the full life-cycle of an Action. To provide lightweight actions with minimal overhead, this interface is specified as a *local* interface, which instructs the CORBA IDL compiler to omit generation of code that allows for remote invocation of the object, creating a locality constrained object. This design substantially reduces overhead, as shown in Section II.3.3. While this locality constraint prevents MACRO agents from directly accessing Action objects, the framework provides a mechanism which does not constrain their use by those agents. This framework allows MACRO agents to access and execute actions while hiding the complexities of action deployment and execution through the Effector interface described in Section II.2.3.1.

The Action attribute `info` allows the Action implementation to self-describe its meta-data, ultimately providing information to the agents about its requirements and capabilities. This information is also used by an implementation of the Effector interface to determine

```
1 extern ''C'' {
2   Action_ptr create_action (void);
3 }
```

Listing II.3: Action Factory

which object references and arguments are to be passed to the operations contained in this Action interface.

These operations allow the Effector to manage the life-cycle of Actions. The `initialize` operation is invoked upon creation of the Action, providing it with object references to deployed components and objects that the business logic may need in order to successfully execute. The `execute` operation implements the business logic of the Action. This operation accepts two parameters, both of type CORBA Any, which is a generic container which may contain any valid CORBA data type, allowing the Actions to accept arguments or provide results in a flexible, but standardized, manner. Finally, the `release` operation informs the Action that it is about to be deallocated so that it may release any resources that it holds.

Each Action implementation provides a factory method (an example of which is found in Listing II.3) that is used by the Effector to construct instances of the action at runtime. Similar to the method used by the DnC specification [59] to construct component instances, this factory method is declared as `extern "C"`, which will allow the Effector interface to load actions at runtime using methods similar to `dlopen` and `dlsym`, which are POSIX APIs for dynamically loading shared libraries.

Effector interface. Listing II.4 describes the Effector interface, which is used by the MACRO agents to load and execute actions. This interface is provided as either a facet or a supported interface on a component. It is used by MACRO agents to execute plans or schemas and interact with the components providing abstractions of the available hardware, as shown in Figure II.5. For example, the `load_action` method may be used by an agent or other Effector client to load a new action from a named shared library that contains a

```
1  interface Effector {
2    Action_Info load_action (in string library_name,
3                             in string factory_name);
4    void unload_action (in string id);
5    Action_Info query_action (in string id);
6    StringSeq list_actions ();
7    void execute_action (in string id,
8                         in any arguments,
9                         out any result);
10 };
```

Listing II.4: Effector Interface

provided factory symbol. The operations on the Effector interface allow MACRO agents to (1) manage the life-cycle of Actions installed in the Effector, (2) determine which Actions have been loaded and query their meta-data, and (3) instruct the Effector to execute an Action.

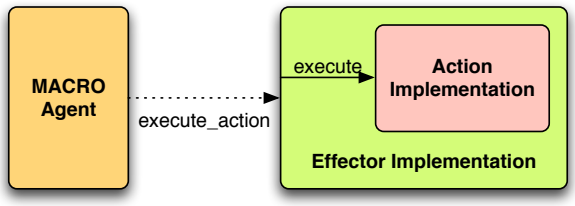

Figure II.5: The Action/Effector Framework

II.2.3.2 Addressing Challenge 2: Providing Flexible Agent Provisioning

The Action/Effector framework described in Section II.2.3.2 provides a mechanism through which MACRO agent implementations may be dynamically provisioned at deployment time with Action objects apropos to the particular hardware configuration, including its suite of available sensors, on which the agent resides. Component interface

```
1  component EffectorProvider {
2    provides Effector effect;
3    attribute Action_Factories factories;
4  };
```

Listing II.5: Example Component with Effector

descriptions, similar to standard CORBA object descriptions, may have attributes of arbitrary types. As seen in Listing II.5, the example component has an attribute of type `Action_Factories`, which is a sequence of structures containing a pair of string member variables indicating a library name and factory symbol name.

Component deployments are described via XML files that capture information about component configuration, topology, and connections. These XML descriptors may be used to populate the value of this attribute with desired library name/factory name pairs at deployment time. Moreover, through the mechanism used to describe the implementation dependencies of components (*i.e.*, shared libraries implementing a component), it is possible to indicate to the Node Manager that shared libraries implementing Actions also be downloaded from the RepositoryManager, as described in Section II.2.2. This approach allows the component providing the Effector interface to invoke the `load_action` operation for each library/entrypoint pair provided during activation.

II.2.3.3 Addressing Challenge 3: Reducing Middleware Footprint

Initial efforts to run MACRO (and the associated middleware infrastructure) presented difficulties and, in some cases, failures due to the large footprint of the default configuration of CIAO and the limited memory capacity of the SEAMONSTER nodes. To reduce memory footprint, the initial application of the deployment infrastructure to SEAMONSTER field hardware included two straightforward modifications:

• **Leverage compiler optimizations.** Most compilers have the ability to provide space-saving optimizations to most code, which an experienced programmer can easily leverage to provide footprint reduction.

• **Leverage mechanisms present in underlying middleware.** The build system of the middleware underpinning of CIAO provides configuration settings that allow one to strip unneeded features from compiled binaries, such as unused portions of the ACE and TAO libraries, which can provide also provide substantial footprint savings in resource-constrained environments.

While these steps are relatively straightforward and not particularly novel, Section II.3.2 shows that they were sufficient to reduce the static footprint of the middleware stack to a level that allowed successful use of the MACRO platform on SEAMONSTER hardware.

II.2.3.4 Addressing Challenge 4: Ensuring Correct Re-Deployment After Reboot

The MACRO approach to resolving this challenge involves creating all deployments as locality-constrained deployments. Locality-constrained deployments describe only components that reside on a single node and refer to connections with components on other nodes using external references. This approach is in contrast to the use of a global deployment plan, which can include components deployed to several nodes, describing connections across nodes by referring to the connected components directly. With locality-constrained deployments each node must execute both the global *and* local deployment entities, rather than only the local ones. Although this increases local node overhead, it allows the middleware to correctly reconstitute its agent and other software deployments upon reboot.

Correctly executing these locality constrained plans on each node requires that connections external to each individual node be correctly re-established. By default, DAnCE only supports connections between components within the context of a single global deployment plan. Since we are using multiple "global" deployment plans that have been locality constrained and deployed using a DAnCE stack unique to each node, correct re-connection

cannot be achieved using only inter-plan connections. This was resolved by enhancing DAnCE to be able to make use of external directory services, such as the CORBA Naming Service, to resolve these external connections at deployment time.

Future work includes extensions to the deployment infrastructure to allow reconstitution of local deployments from global deployment plans, thereby reducing middleware overhead on the field nodes.

II.3 Experimental Results

This section presents the results of experiments that evaluate (1) the effectiveness of MACRO's Action/Effector framework for lightweight, hardware-dependent actions and (2) the reduction of middleware footprint described in Section II.2.3.4. These results show that the efforts described reduced the total static footprint of MACRO and its underlying middleware stack. They also show the reduction in overhead achieved by implementing short-lived actions in the Action/Effector framework discussed in Section II.2.3.2, rather than using heavier-weight components.

II.3.1 Hardware/Software Testbed and Experiment Methodology

The static footprint results were obtained via a cross-compiler tool-chain used to build software for the SEAMONSTER hardware. This tool-chain consists of `g++` 4.1.2 and `ld` 2.17, which are hosted on Debian Linux 4.0 and target `arm-linux-gnu`. The CIAO middleware platform was version 0.6.6.

For the initial baseline results, this platform was compiled using default options, with debugging symbols disabled and the compiler optimization level at `O3`, which instructs the `g++` compiler to optimize for speed. For the results based on our optimization efforts, the middleware was compiled using built-in methods for reducing footprint and the compiler was instructed to optimize using `Os`, which instructs the `g++` compiler to optimize for space. In all cases, we used the GNU `strip` utility to remove any debugging symbols

from the compiled binaries to ensure the footprint metrics just measured the size of the executables.

Executable footprint sizes were determined by statically linking all required symbols from the underlying middleware into the final binary, ensuring that all necessary symbols from the underlying middleware are present, while not including any unnecessary symbols. For the purposes of calculating the size of a component, we assume that any symbols necessary from the underlying middleware were already present in the component server, and thus the calculation of the component footprint sizes was obtained by summing the size of the *shared* libraries that implement the component. This size includes CORBA stubs and skeletons, the servant (the component specific portions of the container), and the executor (business logic) implementation.

Run-time results were obtained using a primary microserver described in Section II.1.6. This microserver consists of a 266 Mhz ARM processor with 64 MB of built-in RAM. The operating system is a derivative of the Debian Sarge running GNU/Linux kernel 2.4.26, which was provided by the manufacturer of the microserver (Technologic Systems).

II.3.2 Initial Footprint Reduction

The results of the efforts described in Section II.2.3.4 are summarized in Table II.1. The `ExecutionManager` and `NodeManager` (which were described in Section II.2.2)

Table II.1: Results of Initial Footprint Optimization

Entity	Default	Optimized	Savings
ExecutionManager	12,203 KB	11,136 KB	1,067 KB
NodeManager	13,865 KB	12,623 KB	1,242 KB
NodeApplication	12,710 KB	11,460 KB	1,250 KB
Null Component	670 KB	605 KB	65 KB

and the `NodeApplication` (which is a component server spawned during the deployment process) each experienced a reduction in footprint of $\sim$1 megabyte. The combined savings reduced the footprint of node-local infrastructure (*i.e.*, the `NodeManager` and `NodeApplication`) from 26.5 MB to 24 MB. Although this reduction allowed us to deploy and operate a prototype MACRO-based application on the SEAMONSTER hardware, this deployment consumed nearly all available physical memory on the primary microservers, and resulted in frequent thrashing on the memory-constrained adjunct microservers.

These results show that largest consumers of memory in the middleware stack are the DAnCE daemons, in particular the `ExecutionManager` and `NodeManager`. The footprint of the newer deployment and configuration aspects of the middleware has been largely overlooked until now and needs to be addressed. Perhaps more importantly is the latency experienced during deployment, which has been observed to take as long as several minutes on SEAMONSTER hardware. Moreover, the DAnCE implementation used in MACRO tangles concerns of deployment and configuration with the runtime elements of the component server in the `NodeApplication`. This entanglement increases footprint by replicating large swathes of deployment logic in each component server. Careful analysis and re-factoring of the deployment infrastructure is therefore needed to substantially decrease footprint and deployment latency.

This serves in part as motivation for our work on the LocalityManager framework described in Chapter III, which outlines our efforts to re-factor the DAnCE framework and address deployment latency issues.

II.3.3 Impact of Action/Effector Framework on MACRO Execution Overhead

MACRO's Action/Effector framework (described in II.2.3.2) substantially reduces footprint overhead compared to using CIAO's complete component implementations to encapsulate SEAMONSTER tasks and actions. Table II.2 summarizes the differences in footprint size between these two approaches.

Table II.2: Action/Effector Footprint

Implementation Type	Size
Component	623 KB
Action Implementation	23 KB
Effector	123 KB

When implemented as a component, the action has a footprint of over half a megabyte, substantially limiting the number of action implementations that could simultaneously be deployed to a single resource-limited field node.

When the action was implemented in MACRO's new Action/Effector framework, however, its footprint was only 23 KB, which is a fraction of the memory required by an executing component. Moreover, an implementation of the Effector framework as a component facet adds only 123 KB to the footprint of an existing MACRO agent component, one of which is required per node.

Table II.3: Action/Effector Footprint

Deployment Latency	Average Time (Seconds)
Component	218.96
Action/Effector	3.23

A more important result, moreover, is the deployment latency experienced by a component compared against the latency experienced by an Action implementation. In this case, deployment latency refers to the amount of time from the moment deployment is started

(*e.g.*, `load_action` in the Action/Effector framework) until deployment is completed and the component or Action is ready for invocation. As shown in Table II.3, which documents the average of twenty runs of each, the difference in deployment latency is dramatic, with component deployment requiring over three minutes while an Action is deployed in only three seconds. These results do not include the time required to download the component implementation from the `RepositoryManager`, which could be substantial over a bandwidth-limited wireless connection, but is only required the first time a component is used on the microserver.

II.4 Related Work

This section compares the work on MACRO with related work.

Resource-Constrained Component Models. Programming in the Many (PitM) [41] is an *architectural style* aimed at the domain of distributed, highly mobile, severely resource constrained embedded systems. While this component model meets the stringent footprint requirements of SEAMONSTER, it lacks the interoperability and rich ecosystem of services offered by CORBA and CCM. PitM also limits communication between components to message-passing, lacking the rich interface-based communication possible with CIAO. The SOFtware Architectures (SOFA) component model [32] based on *Architecture Definition Languages*, which view applications as hierarchies of connected components. This component model provides capability for runtime modifications that may be lighter weight than CIAO components, but which must be described at *design time*[29], thereby limiting flexibility compared with MACRO.

Decision-theoretic planning and scheduling. The planning service used by MACRO server-based agents – SA-POP – is a decision-theoretic planner allowing uncertainty both in environmental conditions and action outcome, like C-SHOP [7] that does so with hierarchical planning and Drips [26] that produces conditional plans. However, to enable planning with resource constraints, such as those of sensor webs, many have chosen to

separate the planning and scheduling/resource aspects of the problem (*e.g.*, [71] and [17]). This approach works well when the resource/time constraints are relatively loose or there are relatively few alternatives in the planning process that could use fewer or different resources. However, with tight resource constraints, as are often present in sensor webs, others have chosen to integrate planning and scheduling as SA-POP does. For example, IxTeT [37] uses partial-order planning like SA-POP and allows interleaving resource conflict resolution with the planning process, but does not perform decision-theoretic planning and incorporates scheduling/timing information directly into the action representation.

Plan schemas for resource-constrained planning and scheduling. The MACRO field agents use plan schemas (also called template plans or skeletal plans) [22], which have also been used in other situations where complete planning was too time consuming for appropriate responses. MACRO's plan schemas have been enhanced with scheduling information, such as in [42], and generated through partial order planning techniques, like [31]. The combination of MACRO server-based agents using the SA-POP planning/scheduling service with generated schemas used by MACRO field agents provides a uniquely flexible solution for autonomy in sensor webs with a server cluster connected to DRE field systems.

II.5 Summary and Lessons Learned

The lessons learned from our extensions to the MACRO distributed deployment infrastructure include:

- **Feasible integration of non-component entities.** The Action/Effector framework has demonstrated the feasibility of integrating non-component entities into component assemblies where footprint, latency, or lifetime rules out the use of a full component. In fact, the Action/Effector framework could be seen as a simple component framework. In this case, Actions are themselves components and Effectors are a simplified container that provides only lifecycle services and no built-in distribution middleware.

• **Unitary Effector may limit framework flexibility.** A unitary Effector (*i.e.*, one which is incapable of operating in a hierarchical manner with other effectors) may limit flexibility in dynamic sensor web environments. Extending the Effector interface to support hierarchical and peer behavior with other Effectors deployed to the same node(s) potentially has two advantages: (1) it allows Effectors to expand their vocabulary as nearby nodes and devices power up/down in response to changing power availability and (2) it allows the creation of "meta-Actions," which are ordered compositions of one or more concrete actions across one or more Effectors.

• **A synchronous Effector interface may cause unacceptable delays.** If an Action hangs or takes longer to complete than expected, the present synchronous interface will also cause the agent plan execution code to hang. This behavior is undesirable, however, since it may cause the agent to miss other important deadlines in its current plan of execution. Asynchronous Effector and Action interfaces can alleviate this concern.

• **CIAO footprint is too large for resource constrained systems.** The stringent resource constraints (*i.e.*, 32-64 MB RAM and processors operating at 266 MHz or less) of SEAMONSTER field hardware were a significant hurdle due to the overhead (especially memory footprint) of CIAO components and deployment infrastructure. Previous CIAO developments focused on environments with significantly greater resources, *e.g.*, more than a gigabyte of RAM and processors faster than two gigahertz. While CIAO is operational on the SEAMONSTER hardware, as indicated in Section II.3, further work is needed to make the middleware efficient under tight resource constraints.

• **DAnCE footprint and deployment latency is too high for resource constrained systems.** As shown in Section II.3, the largest consumers of memory in the middleware stack are the DAnCE daemons, in particular the `ExecutionManager` and `NodeManager`. The footprint of the newer deployment and configuration aspects of the middleware has been largely overlooked until now and needs to be addressed. Perhaps more importantly is the latency experienced during deployment, which has been observed to take

as long as several minutes on SEAMONSTER hardware. Improvements in the deployment latency for DAnCE are further discussed in Chapter IV.

To further reduce the overhead of CIAO components and the DAnCE deployment infrastructure, we are working on multiple approaches, including context-aware generative techniques to prune unnecessary code/features:

• **Generative component specialization.** The CCM specification includes several features and capabilities in the component definition that may not be necessary in all situations, such as generic navigation, introspection, and security features, which contribute to footprint bloat. Generative techniques could be used to prune these features on a case-by-case basis.

• **Generative container specialization.** The CIAO container is intended to be a generic solution providing a large feature set to satisfy user needs in most situations. As such, it contains features and services that may not be necessary in specific deployments, and could be pruned by generating scenario-specific container implementations.

An avenue for simplication of both the component logic and container implementation are discussed in Chapter V and Chapter VI.

• **Improve separation of concerns in DAnCE.** The current DAnCE implementation tangles concerns of deployment and configuration with the run-time elements of the component server in the `NodeApplication`. This entanglement increases footprint by replicating large swathes of deployment logic in each component server. Careful analysis and re-factoring is therefore needed to substantially decrease footprint and deployment latency.

Work that significantly improves the separation of concerns in DAnCE and largely addresses this item is discussed in Chapter III.

CHAPTER III

HETEROGENEOUS AND ADAPTIVE DEPLOYMENT AND CONFIGURATION FRAMEWORKS

Large-scale distributed real-time and embedded (DRE) computing systems, such as shipboard computing environments [39] and air-traffic management [18] systems, are increasingly being developed with the use of component-based software technologies. Component-based development not only offers useful abstractions for developing large systems [27] by encouraging systematic reuse and composition, they also simplify the deployment and configuration process at runtime. The CORBA Component Model (CCM) [52] along with the Deployment and Configuration Specification (D&C) from the Object Management Group [59], and the SOFA component model [10] assist in the deployment and configuration of component-based applications.

Production large-scale distributed computing systems often cannot be limited to a single component model, particularly if they must integrate and interface with legacy systems. While it is possible to use multiple individual deployment frameworks to deploy and configure applications, this approach can complicate the planning process (*i.e.*, assigning instances to nodes, ensuring that sufficient resources exist, performing static verification, etc.), thereby leading to problems during system integration. These problems stem from potentially incompatible tooling, meta-data formats, and problems coordinating the activity of disparate deployment infrastructures.

The original *Deployment and Configuration Engine* (DAnCE) framework provides an offline deployment and configuration for the *C*omponent Integrated ACE ORB (CIAO) [80] CCM implementation. The Locality-Enhanced (LE-DAnCE) version described in this paper provides a deployment tool-chain that can handle heterogeneous deployments and adapt its behavior dynamically to meet changes in the requirements of the applications it deploys.

The remainder of this chapter will be organized as follows. Section III.1 provides an overview of the Deployment and Configuration Specification for Component Based Applications. Next, Section III.2 discusses the research challenges addressed herein. Section III.3 describes solutions to these challenges. Finally, Section VI.2 will describe future enhancements to the LE-DAnCE framework and Section III.4 will describe related work.

III.1 D&C Standard Overview

The OMG D&C specification provides standard interchange formats for meta-data used throughout the component application development lifecycle, as well as runtime interfaces used for packaging and planning. Below we focus on the interfaces, meta-data, and architecture used for runtime deployment and configuration.

III.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 III.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:

- **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

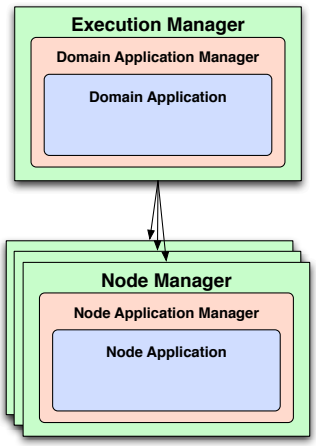

Figure III.1: OMG D&C Architectural Overview and Separation of Concerns

DomainApplicationManager and the node-level entity is known as the *NodeApplication-Manager*. 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 *Node-Application*.

III.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 meta-data 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 meta-data 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 meta-data 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 meta-data is further used to group individual components into assemblies, describe pairings with implementation artifacts (*i.e.*, shared libraries), and create packages that contain both meta-data 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 meta-data 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 meta-data 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.* 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 meta-data 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 meta-data 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.

III.2 Adaptive D&C Challenges in Component-based DRE Systems

The LE-DAnCE deployment framework is motivated by a desire to have a deployment framework that is able to both deploy heterogeneous applications (consisting of potentially multiple component frameworks in DRE systems) and adapt its behavior to meet changing requirement and expectations.

This section describes the key challenges of creating a heterogeneous and adaptive

D&C tool that have motivated the key features of LE-DAnCE that are described in Section III.3.

III.2.1 Challenge 1: Support for Heterogeneous Deployments

The process of applying the PSM to the OMG D&C specification specializes the PIM using a particular component model (such as CCM or EJB) as the target for deployment. Transforming the model with the deployment target, *i*.e. the component model we wish to deploy, as the object of the transformation has the following two categories of important categories specialization of the UML model and semantics found in the PIM:

(1) Data Model and runtime model transformation. The data and runtime model that results from the PIM to PSM transformation is mapped to a format suited to the deployment of the target component model. Transforming an OMG D&C-based model to CCM, for example, results in the creation of a data model and runtime interfaces that are specified in the OMG interface definition language (IDL). This transformation itself does not pose an inherent problem for supporting heterogeneous deployments (*i*.e., deployments consisting of more than one deployment target). This is due to the fact that almost all of the the IDL data structures that are created are agnostic to the deployment target in that they can easily represent non-CCM entities. However, some of the IDL data structures contain concrete data elements that are specific to CCM. For example, the data structures used to communicate connection meta-data contains CORBA Object references. If an attempt was made to reuse the same transformation (including the IDL and the data structures) for other non-CCM component models these data structures might not be semantically meaningful.

(2) Configuration property language. The transformation defines a particular *property language* that communicates target-specific meta-data (such as shared library names, entry points, and component model specific configuration data) in the D&C deployment plan. This property language consists of standard-defined name/value pairs that are encoded in property fields that decorate most entries in a deployment plan. These fields are

used by a D&C framework to describe meta-data specific to a component model that is needed to deploy and configure instances.

Section III.3.1 describes how LE-DAnCE addresses the challenge of supporting heterogeneous deployments by introducing an Installation Handler, a well-defined interface used by D&C infrastructure to manage instance life-cycles.

III.2.2 Challenge 2: Customized Behavior During Deployment

Our experience with the DRE system domains described in Section III has demonstrated that applications may have different expectations of the behavior of the D&C infrastructure based on (1) the domain requirements (*e.g.*, safety-criticality, QoS requirements), or (2) the stage in the development process (*e.g.*, development/testing vs. deployment/operation). These differences in behavior include the following:

Customized error handling semantics. The deployment process for an application may result in many types errors, ranging from incorrect configuration data that may cause components to initialize improperly to application faults that cause runtime entities to crash. While some applications (*e.g.*, in safety-critical domains) should only be activated if and only if they have error-free deployments, other applications (*i.e.*, applications that are fault tolerant) may want their applications activated with "best-effort" deployment semantics, whereby deployment errors may be suppressed so as to not inhibit successful deployment and activation. Moreover, some end-users may want to ignore certain classes of errors (*e.g.*, an invalid CPU affinity setting) or errors from individual instances in a deployment.

Application liveness/status monitoring. End-users may want to leverage customized mechanisms to monitor the liveness/status of particular instances in their applications, particularly in "best-effort" deployment scenarios. Such mechanisms may be constrained by the types of information end-users want to capture, or the format and/or transport used to deliver system events. This information may be useful at the application layer (*e.g.*, to

ensure that certain services are available and to glean information about their configuration) or to a runtime planner/system management service *e.g.*, to enable automatic failure detection/recovery).

Customized discovery. Proper deployment and functioning of applications often depends on discovery services that can locate elements of the deployment infrastructure or to accomplish connections between instances in a deployment plan. Certain domains (*e*.g., security critical) many have stringent requirements as to how these discovery services must secure and manage access to these services that cannot be foreseen by the D&C implementer

Section III.3.2 describes how LE-DAnCE addresses the challenge of providing easily customized behavior during deployment by creating a well-defined interfaces users can leverage to provide customizations invoked during deployment.

III.2.3 Challenge 3: Customization of Behavior at Run-time

D&C infrastructure intended for long-running systems—or intended to provide deployment services to a variety of applications in DRE systems—must often adapt to changing conditions and requirements at runtime.

One use-case for adaptive behavior in D&C framework is the ability to select deployment-time behavior customizations (see Section III.2.2) since not all customizations may be appropriate for a particular deployment. Moreover, it may not be possible to know *a priori*, *i*.e. before the deployment tools are distributed to a target computing environment, which component models a D&C infrastructure may need to deploy. For example, an application may be assembled from components implemented with several different CCM implementations, which while compatible at runtime, have differing interfaces for deployment. Ideally, the D&C infrastructure should be able to upgrade at runtime its capability to deploy different versions of component models without requiring recompiling or restarting the infrastructure.

Section III.3.3 describes how LE-DAnCE addresses the challenge customizing behavior at runtime by providing a facility to deploy installation handlers and interceptors at runtime.

III.3 Decoupling the D&C Specification from Target Component Model

To address the challenges described in Section III.2, the existing DAnCE D&C framework was enhanced with a novel infrastructure entity called the *LocalityManager*. The LocalityManager represents a key change in how the OMG D&C specification transforms platform-independent D&C models to target specific component models. Rather than mapping the entire specification to a particular *component model*, we map the data and runtime model to a particular *distribution middleware* that is used only to represent and communicate deployment meta-data and deployment directives at runtime. Using such an approach for mapping the D&C PIM to concrete language elements allows us to reuse much of the data model which, as outlined in Section III.2.1 is largely agnostic to the deployment target.

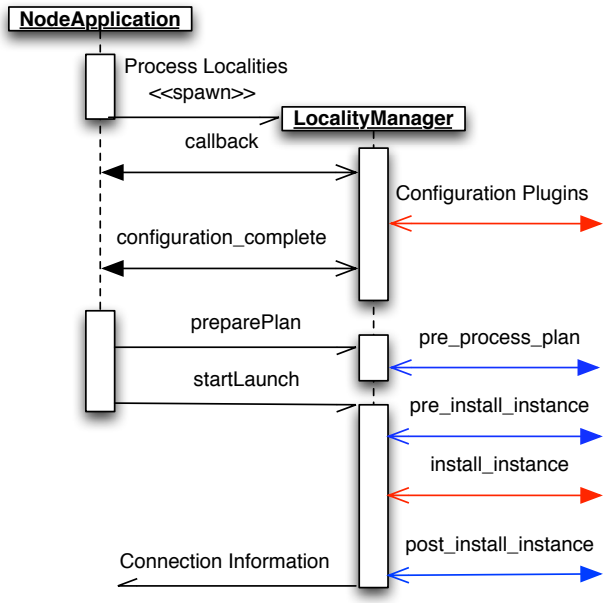

Figure III.2: Locality Manager

The LocalityManager, a key feature in our Locality-Enhanced DAnCE (LE-DAnCE), is an entity spawned by the NodeApplication entity described in Section III.1.1. The LocalityManager entity is intended to be a generic application server, maintaining a strict separation of concerns between *generic* deployment logic and the *specific* runtime logic necessary to deploy a particular deployment target. To provide a well-defined interface between the NodeApplication and the LocalityManager, we have reused elements from the D&C specification by including operations from the *Manager* interface and inheriting from the *ApplicationManager* and *Application* interfaces.

Figure III.2 shows the initial start up sequence of the LocalityManager.

The remainder of this section describes the structure and functionality of the LocalityManager.

III.3.1 Instance Installation Handlers

To address the challenge described in Section III.2.1, the implementation of the LocalityManager is entirely agnostic to the particular component model it is attempting to deploy, delegating all *component model specific* life-cycle management operations to pluggable Instance Installation Handlers, which we describe below.

Instance installation handlers represent a well-defined interface that is used by the LocalityManager to manage the life-cycle of all entities that are installed during deployment. The operations that are included in this interface were heavily influenced by the typical CCM Component life-cycle, which is shown in Figure III.3. The included operations allow the LocalityManager to install/remove an instance, create/remove a connection, indicate that configuration is complete, and to activate/passivate an instance. The operations in this interface are currently used by the locality manager to perform all initial deployment actions, and can be used in the future to provide for application re-deployment and re-configuration in the future.

It is important to note that not all instance types will require every lifecycle operation

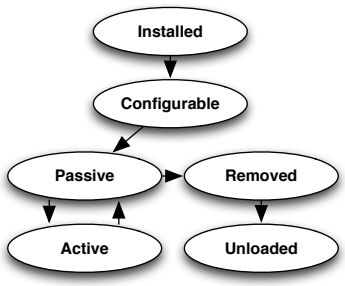

Figure III.3: Typical CCM Component Lifecycle

in the installation handlers to be implemented. For example, a total of four installation handlers were created to support the installation and management of CIAO components. First, it is necessary to instantiate a CIAO container to host any components hence an installation handler was created that initializes the CIAO runtime and is capable of instantiating containers.

Second, an installation handler was created to support the installation of CCM Homes. Neither of these first two entities have the same number of lifecycle states as a CCM component. For example, neither have connections nor distinct active/passive states, so the relevant operations in the handler remain unimplemented. Finally, handlers were created that are able to load components directly from a dynamically loaded shared library or from an appropriate factory operation on a CCM Home. These handlers implement all of the lifecycle operations in the installation handler.

Despite the differences in how each of these entities is installed and behaves at runtime, the common interface for managing their lifecycle allows the LocalityManager to treat each as an abstract instance. More importantly, it allows the LocalityManager to easily be configured to deploy entirely new instance types provided appropriate installation handlers are loaded.

For example, assume an application is made of only CIAO components. To accomplish this deployment, the LocalityManager would require two installed installation handlers - one for the containers that will host the components, a second installation handler that

manages the life-cycle of the components. In this case, the deployment plan would contain a instance that represents a container, and other instances that represent the components to be installed. The LocalityManager would first select (based on meta-data in the plan) the installation handler for the container and invoke the "install" operation, which causes the container handler to bootstrap the CIAO infrastructure. Next, the LocalityManager will select the handler for CIAO Components, and invoke "install" operations for each component instance, which will cause the handler to interact with the already installed container to create a component.

As a further example, lets assume that we now wish to introduce heterogeneity into this deployment example by also including non-CCM component instances. This can be accomplished by annotating the instances with an appropriate identification string and providing appropriate installation handlers for the new component model.

III.3.2 Deployment Portable Interceptors

Addressing the challenge described in Section III.2.2, by providing a mechanism for end-users to customize the behavior of the middleware, the LocalityManager also implements a mechanism which can be used to modify the elements of the deployment plan both before and after invocation of each life-cycle management operation. This mechanism, which we call "Deployment Portable Interceptors", was inspired by CORBA Portable Interceptors [46], and is described below.

The Deployment Portable Interceptor (DPI) facility in the LocalityManager allows end-users to supplement or modify behavior during deployment. The operations in the DPI interface derived from the operations present in the Installation Handler interface. Each operation in the Installation Handler interface resulted in two operations added to the DPI interface – one which is invoked *before* the lifecycle operation, and another which is invoked after.

In Figure III.2, for example, the LocalityManager invokes a DPI hook before (a *pre*

hook) and after (a *post* hook) the *install_instance* lifecycle operation. All of the "pre" interceptors receive the same parameters as their associated Instance Handler operation, and are allowed to manipulate those parameters to change the behavior of the operation. For example, an alternative discovery service for connections may be implemented by overriding the *pre_connect* interception point with logic that would retrieve the appropriate connection reference and modify the parameters passed to the *connect_instance* operation.

The "post" interceptors generally receive the same parameters of the lifecycle operation that preceded them, in addition to an additional parameter that contains any error result (*i.e.*, exception) that may have arisen during execution. Unlike the "pre" interceptor, the "post" event is only allowed to manipulate the error parameter, if present. This parameter allows the interceptor to, for example, log success or failure of the event (*i.e.*, for a system health and status service), or to clear the error status, causing that error to be overlooked by the LocalityManager implementation (*i.e.*, for implementation of best-effort deployment semantics).

III.3.3 Configuration of Handlers and Interceptors

Finally, to address the challenge outlined in Section III.2.3 and provide a mechanism to provision both Installation Handlers and Deployment Interceptors at runtime, the LocalityManager is capable of installing these entities during deployment as they would any other instance as described below.

Allowing runtime adaptation of the deployment framework requires the ability to dynamically add or remove instance installation handlers and deployment interceptors on a per-deployment basis. In the LocalityManager, we have added a facility which invokes user-supplied configuration plug-ins during start-up through a well-defined interface. In Figure III.2, this process takes place after the LocalityManager initially calls back to the NodeApplication to receive configuration meta-data that is present in the deployment plan.

Meta-data provided to the LocalityManager consists of a series of name/value pairs.

The name of each property is used to select an appropriate configuration plug-in to which the value is provided. By including both a property on the LocalityManager instance in the plan that describes the desired Instance Handlers and Deployment Interceptors for the plan, and a configuration plug-in that is able to interpret that property, it is possible to load them before the LocalityManager attempts to install any instances in the plan.

This facility has utility outside the configuration of Handlers and Interceptors. For example, we used these plug-ins to change QoS parameters (such as priority or CPU affinity) of the LocalityManager instance at deployment time without introducing platform-specific code into the LocalityManager.

III.4 Related Work

DeployWare [21] is a framework for managing heterogeneous software deployments in grid environments. Deployments in this system are described using a domain-specific modeling language that captures deployment meta-data in a manner agnostic to the eventual deployment target. Heterogeneous deployments are then accomplished by using appropriate "personalities", which are hierarchies of Fractal components that implement parts of the deployment process. Unlike the OMG D&C specification, DeployWare does not provide a well-defined set of meta-data that can be used throughout the application development lifecycle nor does it provide a way to model hardware resources in the computational domain. Such meta-data is desirable for fostering both reuse and a library of COTS component applications. As a result, DeployWare can be harder to use in larger projects in which multiple, independent teams must collaborate.

ADAGE [38] is another grid deployment tool that is capable of heterogeneous deployment capable of deploying both CCM and MPI applications. In this system, applications are described in a middleware-specific description language which is provided to a "translator" that converts that description into a middleware agnostic format called the Generic Application Description (GADe) model. Like the OMG D&C specification, it provides a

description language for hardware resources, but does not provide an expressive vehicle for component meta-data. For example, it is not possible to capture specific component/node pairings, which are decided by the deployment tool, or to capture QoS attributes, such as Processor/Core affinity or process priority. While this automatic planner included in ADAGE makes the planning process easier for the grid environments for which this tool is intended, it is not desirable for DRE systems in which specific control over the application topology may be required to provide sufficient quality of service for the application.

SOFA [10] is a component model with its own D&C framework that provides many advanced features for component-based software, including behavior specification and verification, software connectors for supporting many communication middleware platforms, and a robust redeployment mechanism. While SOFA's component model and D&C framework have many advanced and interesting features, it supports neither heterogeneous deployment nor adaptation of the behavior of the D&C framework found in DAnCE.

The work that comes close to the goals of LE-DAnCE is described in [67]. The authors also use hierarchical separation of concerns to provide concurrent, and hence faster deployments. A major difference of this work with that of LE-DAnCE is that the former does not consider the OMG D&C specification but rather some general concepts of deployment and configuration. One of the primary goals of LE-DAnCE is to provide solutions to standardized technologies for wide applicability. Naturally, in LE-DAnCE we seek solutions that will not break the standards, yet enable us to provide elegant performance optimizations.

The work presented in [30] seeks to find deployment solutions in dynamic environments. While the goal of dynamic environments is similar to that of LE-DAnCE, the focus of this related research is mostly on deploying a hierarchical component – essentially an assembly of components treated as a single unit – while ensuring that the deployment of individual monolithic units do not violate architectural constraints of the platform and the network before deploying that component.

III.5 Summary and Lessons Learned

This chapter described the LocalityManager, which is an extension to the OMG D&C specification and key feature of LE-DAnCE, that adds three important capabilities to the original standardized deployment framework to support heterogeneous deployment and adaptation. First, the LocalityManager uses *Instance Installation Handlers* to deploy applications that use heterogeneous component models by encapsulating middleware-specific deployment logic in a well-defined interface that handles all lifecycle events. Second, it can adapt the behavior of the deployment tool-chain at runtime through the use of *Deployment Portable Interceptors*. Third, the D&C tool-chain can adapt more readily to changing requirements by having the ability to load both installation handlers and interceptors at runtime.

The implementation of heterogeneous deployment and interceptors found in the LocalityManager described in this chapter is complicated by the fact that the deployment plan meta-data defined by the D&C specification is poorly suited to capture deployment ordering or dependencies. It is therefore hard to determine the order in which instances should be installed when there are implicit dependencies, *e.g.*, CIAO containers must be installed prior to the components they host. This challenge was addressed in the LocalityManager by following a FIFO approach to select the order of installing instance types. While sufficient for current end-users, this approach will not scale as the number of installed interceptors and/or installation handlers increase.

This this issue could be addressed by adapting the hierarchical deployment specification techniques in the DeployWare and SOFA component models. In particular, this prior work could be leveraged to build robust redeployment and reconfiguration capabilities into DAnCE to support adaptive deployment behavior in applications managed by this framework. Moreover, as we gain a complete understanding of the shortcomings of the OMG D&C specification and the associated PIM to PSM mapping process, we will work within the OMG to produce an updated specification.

CHAPTER IV

DETERMINISTIC AND EFFICIENT DEPLOYMENT IN COMPONENT-BASED ENTERPRISE DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

Component-based software engineering techniques are increasingly applied to develop large-scale *distributed real-time and embedded* (DRE) systems, such as air-traffic management [18], shipboard computing environments [39], and distributed sensor webs [75]. 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 [24].

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 [64]. 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 [59] 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.

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 deterministic manner. In this chapter, the timeliness of the deployment infrastructure to execute the deployment plan is referred to 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 chapter motivates and describes architectural enhancements made to the OMG D&C specification to achieve determinstic deployment latencies for large-scale DRE systems. The solution is called the *Locality-Enhanced Deployment and Configuration Engine* (LE-DAnCE), which extends the earlier *Deployment and Configuration Engine* (DAnCE) [14]. LE-DAnCE was developed 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 [39, 64], however, 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

[1] Although originally developed for the *CORBA Component Model* (CCM) [56], the OMG D&C specification is defined via a UML metamodel that is applicable to many other component models.

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 chapter is organized as follows: Section IV.1 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 IV.2 describes how these sources of overhead were addressed, focusing on deployment latency; Section IV.3 analyzes the results of experiments conducted to compare LE-DAnCE with DAnCE; Section IV.4 compares this research with related work on deploying and configuring large-scale distributed applications; and Section IV.5 presents concluding remarks and lessons learned.

IV.1 Impediments to Efficient and Deterministic Deployment Latency

This section presents an overview of the process used by 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) [14] supports the separation of concerns espoused in the D&C specification. Key sources of overhead are exposed that impact deployment latencies in DRE systems and pinpoint the architectural limitations in the D&C specification that exacerbate these overheads. An overview of the meta-data and interfaces defined by the standard may be found in Section III.1.

IV.1.1 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.

IV.1.2 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. The existing DAnCE [14] OMG D&C implementation is used 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 meta-data from disk in XML format and an architectural ambiguity in the runtime infrastructure that encourages sub-optimal implementations.

IV.1.3 Challenge 1: Parsing Deployment Plans

Component application deployments for OMG D&C are described by a data structure that contains all the relevant configuration meta-data 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 [25], 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 IV.3.2. 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 III.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 IV.2.1 describes how LE-DAnCE resolves Challenge 1 by pre-processing large deployment plans offline into a portable binary representation.

IV.1.4 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 IV.1.1 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 [6].

To minimize initial implementation complexity, synchronous invocation was used 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 IV.1.

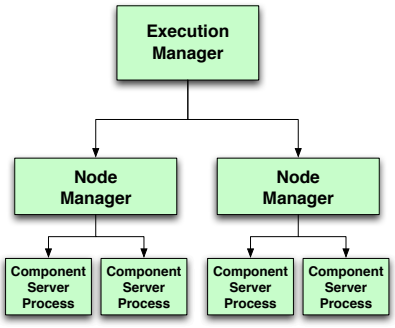

Figure IV.1: Simplified DAnCE Architecture

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 [58], which is another CCM implementation tailored for enterprise DRE systems, such as air-traffic management systems.

While the DAnCE architecture shown in Figure IV.1 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 IV.2.

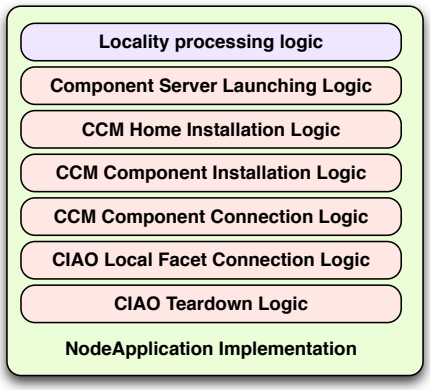

Figure IV.2: DAnCE NodeApplication Implementation

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.

Section IV.2.2 describes how LE-DAnCE resolves Challenge 2 by leveraging asynchronous features of the underlying CORBA middleware to paralellize at the global level.

IV.2 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 IV.1.2. Section IV.2.1 describes how we reduced deployment latency arising from the challenge of processing the XML-based deployment descriptors outlined in Section IV.1.3. Section IV.2.2 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 IV.1.4.

IV.2.1 Improving Runtime Plan Processing

There are two approaches to resolving the challenge of XML parsing outlined in Section IV.1.3.

1. Optimize the XML to IDL processing capability. DAnCE uses a vocabulary-specific XML data binding [83] 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 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) [55] 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.

IV.2.2 Parallelizing Deployment Activity

To support parallelized dispatch of deployment activity at the node level, OMG D&C standard was enhanced by adding a LocalityManager to LE-DAnCE. The LocalityManager unifies all three deployment roles outlined in Section III.1.1, and functions as a replacement for the component server in Figure IV.1. An overview of LE-DAnCE's LocalityManager appears in [65].

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 implementation, 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 IV.3 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.

The new LE-DAnCE LocalityManager and Installation Handlers make it substantially

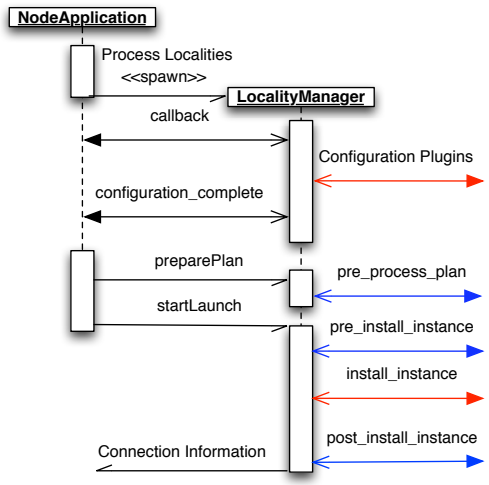

Figure IV.3: LocalityManager Startup Sequence

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 IV.4. The Deployment Scheduler combines the Command pattern [23] and the Active Object pattern [69]. Individual deployment actions (*e.g.*, instance installation, instance connection, *etc.*) are encased inside an Action object, along with any required meta-data. 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 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 meta-data 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

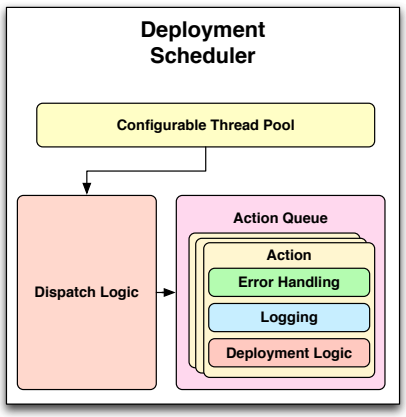

Figure IV.4: DAnCE Deployment Scheduler

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.

IV.3 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 IV.2.

IV.3.1 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 [82] 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 [43]. 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.

IV.3.2 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 IV.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 IV.2.1.

Table IV.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

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 meta-data 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 excercised 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 IV.1 demonstrate that the CDR encoding is an improvement of several orders of magnitude over runtime XML processing. Moreover, the approach described in Section IV.2.1 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.

IV.3.3 Experiment 2: Measuring Application Deployment Latency

Table IV.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

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 IV.3.2 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 IV.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 IV.1.1. 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 IV.2.2. The nonlinear growth of the time required for this phase shown in Table IV.2 makes extremely large deployments infeasible, which is the reason why results for 50,000 and 100,000 components are not included.

IV.3.4 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 IV.5.

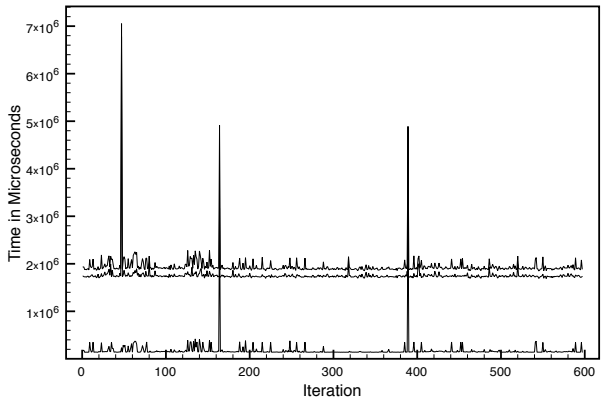

Figure IV.5: Latency Jitter for 1000 Component Deployment

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.

Of particular interest in Figure IV.5 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.

Table IV.3: Deployment Latency Results for 600 iterations of a 1000 component deployment.

	Total Time	Prepare Plan	Start Launch	Finish Launch	Start
Mean	1.9311	1.7476	0.16548	0.01275	0.0049
Max	2.2874	2.0043	0.41976	0.05526	0.0127
Min	1.8503	1.6890	0.13752	0.01072	0.0045
Std. Dev.	0.0780	0.0402	0.05581	0.00559	0.00098

IV.4 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 [77] is a deployment framework intended for grid-based distributed applications. GoDIET uses XML meta-data 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 [8] 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 [21] is another framework for managing deployments in grid environments based on the Fractal [8] component model. It supports heterogeneous deployments and currently supports middleware intended for the grid environment, such as MPI [5] and GridCCM [66]. Like LE-DAnCE, DeployWare captures deployment meta-data in a manner that is relatively agnostic to the eventual deployment target. Unlike LE-DAnCE, however, DeployWare does not capture more complex deployment meta-data (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 [67], 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 [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 meta-data 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 [15] was an earlier effort we conducted to reduce latency and increase predictability 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 determinstic 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 reordering 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.

IV.5 Summary and Lessons Learned

This chapter 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. Key features of the *Locality-Enhanced Deployment and Configuration Engine* (LE-DAnCE) were described that improved DAnCE to alleviate key sources of deployment latency overhead asssociated with XML pre-processing 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 IV.3 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 [73] 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 meta-data into the deployment infrastructure [74]. 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.

CHAPTER V

EXTENDING MIDDLEWARE CAPABILITIES USING CONNECTORS

The trend towards realizing enterprise distributed real-time and embedded (DRE) systems motivates the use of component-based middleware, such as the OMG's Lightweight CORBA Component Model (LwCCM) [47]. Component-based middleware offers DRE system developers significant flexibility in modularizing their system functionalities into reusable units, simplifies the deployment and configuration of the systems, and supports dynamic adaptation of system capabilities. Deployment and configuration standards, such as the OMG's Deployment and Configuration (D&C) specification [59], play a major role in realizing these capabilities.

Existing and planned enterprise DRE systems must increasingly support large data spaces generated by thousands of collaborating nodes, sensors, and actuators that must exchange information to detect changes in the operational environment, make sense of that information, and effect changes. These capabilities require scalable publish/subscribe (pub-/sub) semantics [19] that support a range of QoS properties, that control properties, such as liveliness, latency, deadlines, timing, and reliability. Unfortunately, the conventional component technologies used to develop enterprise DRE systems either do not provide first class support for pub/sub semantics or do so in an ineffective manner that is not scalable and does not support real-time QoS properties.

A standardized, QoS-enabled pub/sub technology called the OMG Data Distribution Service (DDS) [53] has emerged as a promising pub/sub technology to support the requirements of enterprise DRE systems. DDS includes standard QoS policies and mechanisms to handle data (de)marshaling, node discovery and connection, and configuration. Middleware based on the DDS standard has been applied successfully in mission-critical domains, such as air traffic management systems [18] and tactical information systems [28].

While the DDS specification simplifies key implementation aspects of pub/sub application, these benefits come at price of increased complexity of configuration glue code that must be written and maintained. Moreover, this configuration boilerplate code tightly couples the QoS configuration of a DDS application at compile-time, unless application developers create ad hoc methods of specifying the middleware configuration at runtime. Analysis [2] has shown that as 80% percent of DDS-related code in a typical applications is associated with configuring the middleware. Likewise, over half of the DDS API that developers must learn is configuration-related.

Addressing these deployment and configuration requirements of modern DRE systems calls for component-based middleware, such as LwCCM, to provide first-class support for QoS-enabled, pub/sub technologies, such as DDS. This need has been recognized and documented through the efforts of industry and academic collaborators in the OMG *DDS for Lightweight CCM* (DDS4CCM) [57] specification. Implementing this specification is hard, however, due to inherent and accidental complexities in integrating LwCCM and DDS. The inherent complexities stem from (1) differences in the language bindings and memory management strategies of the two middleware technologies, (2) incompatibilities between the various specifications, (3) deployment and configuration challenges to recognize DDS abstractions within LwCCM, and supporting variants of DDS in a single LwCCM implementation. The accidental complexities stem from (1) manual approaches to creating the deployment and configuration meta-data for DDS elements within LwCCM, and (2) the need to minimize runtime overhead imposed by both the deployment and configuration meta-data, and the additional abstraction atop native DDS.

This chapter describes how LwCCM and DDS have been integrated to address the inherent and accidental complexities described above as follows:

1. Systematic use of the extensible interface pattern in the form of mixins to extend existing interfaces as well as the deployment and configuration meta-data to bridge the incompatibilities between the two technologies.

2. A template-driven code generation approach that maximizes the potential for portability between various DDS implementations and maximizes maintainability.

3. Options to customize the integration are provided, which ensures that the runtime footprint of the resulting system does not pay unwanted memory footprint penalties.

4. Improvements to the D&C approach mandated by the DDS4CCM specification.

These contributions enable the realization of a product-line of DDS4CCM systems where it is possible to vary the implementations of the DDS technology used as well as support a wide range of port types for the LwCCM component technology. Empirical evaluations of our approach demonstrate that our implementation of the DDS4CCM specification, which is called DDS4CIAO, substantially eases the development of DDS-based applications while providing performance almost identical to native DDS applications.

The remainder of this chapter is organized as follows. Section V.1 summarizes key challenges encountered when integrating DDS within LwCCM; Section V.2 describes the design of DDS4CIAO that resolves the challenges described in Section V.1.3; Section V.3 examines the code generation of DDS4CIAO and analyzes the results of experiments that evaluate the performance of DDS4CIAO; and Section V.4 compares DDS4CIAO with related work.

V.1 Impediments to Integrating LwCCM and DDS

In this section both the inherent and accidental challenges in providing first class support for Data Distribution Service (DDS) within the Lightweight CORBA Component Model (LwCCM) is presented.[1] To better appreciate these challenges, first provide an overview is provided of LwCCM and DDS, and the deployment and configuration standard. Subsequently these challenges are elaborated.

[1]The LwCCM is a subset of the OMG CORBA Component Model. In the rest of this paper LwCCM is references because of the focus on DRE systems but the issues apply equally well to CCM.

V.1.1 Overview of the OMG Data Distribution Service (DDS)

The OMG DDS specification [53] defines a standard architecture for exchanging data in pub/sub systems. DDS provides a global data store in which publishers and subscribers write and read data, respectively. DDS provides flexibility and modular structure by decoupling: (1) *location*, via anonymous publish/subscribe, (2) *redundancy*, by allowing any numbers of readers and writers, (3) *time*, by providing asynchronous, time-independent data distribution, and (4) *platform*, by supporting a platform-independent model that can be mapped to different platform-specific models, such as C++ running on VxWorks or Java running on Real-time Linux.

DDS entities include *topics*, which describe the type of data to be written or read, *data readers*, which subscribe to the values or instances of particular topics, and *data writers*, which publish values or instances for particular topics. Moreover, *publishers* manage groups of data writers and *subscribers* manage groups of data readers.

Properties of these entities can be configured using combinations of DDS-supported QoS policies. Each QoS policy has $\sim$2 parameters, with the bulk of the parameters having a large number of possible values, *e.g.*, a parameter of type long or character string. DDS provides a wide range of QoS capabilities that can be configured to meet the needs of topic-based distributed systems with diverse QoS requirements. DDS' flexible configurability, however, requires careful management of interactions between various QoS policies so that the system behaves as expected. It is incumbent upon the developer to use the QoS policies appropriately and judiciously.

V.1.2 Addressing Limitations in the LwCCM Port System via DDS4CCM

The OMG's DDS4CCM [57] specification was developed to overcome the following limitations in LwCCM and DDS while still preserving the inherent advantages of each technology.

Limitation 1: Support for event-based pub/sub communication in LwCCM is extremely limited. LwCCM does not specify a particular distribution middleware that must be used inside the container for communicating events. While this approach allows a substantial amount of flexibility on the part of implementation authors, allowing them to choose to implement this support using, for example, the CORBA Event Service or CORBA Notification Service, has two important drawbacks. First, the integration of new pub/sub middleware requires modification of not only the core container implementation, but potentially also the deployment and configuration infrastructure in order to properly operate. As a result, this is an extremely complex task, often requiring that the integrator be an expert in both the LwCCM implementation and the desired distribution middleware.

Second, in order to remain completely generic, the interface available to component developers for event-based communication consists of only two operations: 1) a single method per port that allows for a single event to be published at a time, and 2) a single callback operation that provides an event to the component as it arrives. This prevents the component from taking advantage of many features of pub/sub messaging middleware that provide for status notifications and per-message QoS adjustment.

Limitation 2: Grouping of related services must be done in an ad-hoc manner. In many cases, services offered by a component require more than one interface in order to provide correct operation. As a simple example, consider a scenario in which two components expect to cooperate via mutually connected interfaces. In this scenario, one component provides an interface “A” and requires an interface “B”, while another component provides complementary ports (*i.e.*, provides “B” but requires “A”). In order for semantically correct operation, the connections for both “A” and “B” must go to the same component, but there exists no way in LwCCM to indicate this constraint on an interface level. To accomplish this goal, developers must rely on ad-hoc naming conventions and

documentation. This approach has the unfortunate side effect of complicating the planning process and potentially causing subtle and pernicious runtime errors if connections are mis-configured.

The DDS4CCM specification addresses these limitations by enabling LwCCM to leverage the powerful pub/sub mechanisms of DDS. First, it provides a substantially simplified API to the application developer that completely removes the configuration of the DDS middleware from the scope of the application developer. Second, it provides a set of ready-to-use ports that hide the complexity and groups data writing/access API with the appropriate callback and status interfaces. Third, by providing integration with the LwCCM container, DDS applications are now able to take advantage of robust and mature deployment and configuration technologies that obviates the need to write boilerplate application startup code, runtime configuration of QoS policies, and coordinated startup and teardown of applications across multiple nodes.

In particular, DDS4CCM proposes two new constructs — *extended ports*, which allow for the grouping of related services, and *connectors*, which allow for flexible integration of new distribution middleware. These new entities are defined using an extension of the IDL language for components (IDL3) called IDL3+. It is possible to map each of these new IDL3+ language constructs back to basic IDL3 using simple mapping rules to enable inseparability with older CCM implementations. Next, a brief overview of these enhancements is provided.

Extended Ports: Extended ports provide a mechanism whereby component designers can group semantically related ports to create coherent services offered by a component. These extended ports, defined using a new IDL keyword `porttype`, are defined outside the scope of components. Extended ports are allowed to contain any number of standard LwCCM ports in either direction. While these ports are allowed in terms of the specification to contain standard LwCCM event ports, in practice this is highly unlikely due to the limitations outlined earlier. Moreover, in combination with connectors (described

```
1
2  interface Data_Source {
3    Data pull (in long uuid);
4  };
5  interface Notifier {
6    void data_ready (in long uuid);
7  };
8  porttype NotifiedData {
9    provides Data_Source data_source;
10   uses Notifier data_ready;
11 };
12 component Sender {
13   port NotifiedData data_out;
14 };
15 component Receiver {
16   mirrorport NotifiedData data_in;
17 };
```

Listing V.1: Extended Port IDL

next), these extended port definitions could be used to recreate the behavior of the existing standard CCM event infrastructure.

Listing V.1 shows IDL for an example extended port. In this example, a service is created whereby one component may notify another of data that is ready to be sent, and the destination component may optionally choose to pull that data from the source component. Since each of the interfaces `Data_Source` and `Notifier` are semantically linked, *i.e.*, operation of the component application would be fundamentally broken if these ports are not pairwise connected, they are grouped into a single `porttype`. This is an indication to both high level modeling tools and the component runtime that these ports must be connected as a pair, and can generate appropriate deployment plan meta-data to connect them at runtime. Extended ports are assigned to components using two new IDL3+ keywords. The `port` keyword indicates that the component supports the extended port *as described*. The `mirrorport` keyword indicates that the component *inverts* the direction of the extended port, *i.e.*, facets become receptacles.

Some extended ports may vary only in the data type used as parameters. In order to avoid the necessity of re-defining an extended port for each new data type, IDL3+ offers a new template syntax that may be used to define services that are generic with respect to data type.

Connectors: While the extended port feature described above is quite useful, their power is most suited to providing novel communications mechanisms to components that provide/use those interfaces. In order for the extended ports to provide a coherent interface to a new distribution middleware, such as DDS or the CORBA Event Service, the business logic that supports that abstraction must be contained in some entity. This unit of business logic is called a *connector*. Connectors combine one or more extended ports to provide well-defined interfaces to new distribution middleware or communication techniques between components. In many cases, a single connector will support at least two extended ports, one intended for each "side" of the communications channel. By separating the core communications business logic, these connectors can then be used as COTS components across several applications without requiring modification of the core container code.

Connectors are defined in similar fashion to a component, using the new IDL3+ keyword `connector`. Connectors may contain, of course, one or more extended ports. In addition, they may also support attributes which are intended to be used to assist in runtime configuration, *i.e.* topic names, port numbers, QoS parameters, *etc.*. Finally, connectors also support inheritance which can be used to extend existing connectors with new capabilities. At runtime, instead of creating a new IDL type structure for the connector infrastructure, they are defined as components, deriving their interface from the same `CCMObject` used by regular components. Indeed, in the IDL3+ to IDL3 mapping, the `connector` keyword becomes `component`. This approach is much desirable in that no additional work is necessary in the D&C toolchain to support the deployment and configuration of

connectors. Moreover, connector implementations can take advantage of the same Component Implementation Framework that is available to standard LwCCM components and thus can take advantage of advances in services offered by the container.

V.1.3 Challenges in Integrating LwCCM and DDS

Although the DDS4CCM specification attempts to address the limitations of individual technologies, realizing an implementation of the DDS4CCM specification is fraught with multiple inherent and accidental complexities explained below:

Challenge 1: Indicating that a connector implementation has been fully configured, and should be made ready for execution. After a connector implementation has received all necessary configuration information, it must proceed to create the underlying low-level DDS entities (*e.g.*, `DomainParticipant`, `DataWriter` and/or `DataReader`) that are necessary for correct operation. To accomplish this task, the specification mandates the use of an operation called `configuration_complete` on the external connector interface. This operation, however, is not delegated to the connector business logic and thus is insufficient to fully inform the connector implementation of completed configuration. Section V.2.1 discusses our approach to resolve this challenge.

Challenge 2: Reducing D&C-related runtime memory footprint. The DDS4CCM specification mandates the use of LwCCM Homes (which nominally act as factories for component instances) as the primary vehicle for passing configuration information from the deployment plan to individual connector implementation during deployment. While this approach is certainly functional and sound (and in keeping with the spirit of the LwCCM specification), our experience developing component applications with LwCCM reveals that the home entity often adds very little value to the configuration of individual component, or in this case connector, instances. In most cases, the home implementation is little more than a simple factory that directly instantiates the component and nothing else. Meanwhile, the home instance carries a non-negligible amount of runtime footprint due

to the CORBA interface and accompanying home-specific generated container code that is necessary. Section V.2.2 discusses our approach to resolve this challenge.

Challenge 3: Reducing Connector-related runtime memory footprint. The decision to treat connectors for all intents and purposes as full LwCCM components greatly simplifies the implementation by substantially reducing the number of changes in the core container necessary to support the specification. A consequence of this decision, however, is that the runtime footprint of a LwCCM application using connectors could substantially increase. For example, assuming a deployment where each component instance has an associated connector instance, the number of actual "components" in the deployment is doubled. In memory-constrained DRE systems, this can be a significant impediment. Section V.2.3 discusses our approach to resolve this challenge.

Challenge 4: Supporting Local Interfaces as Facets All of the extended ports contained in the DDS4CCM specification are defined as "local interfaces". Local interfaces are significantly different from standard CORBA interfaces due to the fact that they are not generated with any of the infrastructure necessary to support remote invocation. As a result, any invocation on these interfaces does not travel through the CORBA internal infrastructure and as such only incurs overhead nominally involved in a virtual method invocation. The problem this strategy causes with the deployment and configuration aspect of LwCCM is very subtle: since these local interfaces lack the necessary remoting code, it is impossible to pass references to these local objects through a standard CORBA interface. Indeed, this behavior is undefined; any attempt to do so will fail and cause an exception to be propagated to the caller. Unfortunately, all of the standard-defined connection methods, including the Component Navigation interfaces used by the D&C tooling to make connections between components rely on being able to retrieve object references to Facets over a standard CORBA interface and pass these references to the receptacle component over a similar interface. Not having an object reference for the extended port implies that

the existing D&C tooling cannot be leveraged in a straightforward manner. Section V.2.4 discusses our approach to resolve this challenge.

Challenge 5: Supporting Multiple DDS Implementations One significant benefit of writing DDS applications using the DDS4CCM API is that it potentially makes it substantially easier to switch between various DDS implementations. Prior work [84] has shown that differences in the architecture between these different implementations cause them to have different strengths depending on the architecture of the application and hardware environment. Moreover, due to the proprietary nature of most DDS implementations and the different licensing requirements of each implementation, the ability to quickly and easily switch the targeted implementation would greatly facilitate the development of COTS DDS components. While it is currently possible to target multiple DDS implementations *at compile time* due to the presence of a standard API, subtle differences in the implementations of these APIs can make this difficult to accomplish. Ideally, any implementation of the DDS-4CCM specification would be architected in such a way that the core business logic of the connector is shielded from the differences between DDS implementations. In addition, the connector architecture could make it possible to delay the choice of DDS implementation from compile time to deployment time. Section V.2.5 discusses our approach to resolve this challenge.

Challenge 6: Making it easy for users to define their own connectors The DDS-4CCM specification provides for two connector types that correspond to common DDS usage patterns. The first provides for a state transfer pattern, and is intended to connect "Observable" components that publish state to other "Observer" components that consume that state. The second provides for event transfer connecting supplier components to consumer components. These two connectors, however, are not intended to be the only ones that are supported in the context of the specification. To that end, two "base" connectors are provided that collect the various configuration meta-data as attributes. It is intended that users be able to define their own connectors that are better suited to their usage cases. To

support this capability, the code generation techniques should be extensible such that it is easy for users to create their own connectors without having to modify the code generators. Section V.2.6 discusses our approach to resolve this challenge.

V.2 Resolving LwCCM and DDS Integration Challenges in DDS4CIAO

This section describes how the challenges in integrating LwCCM with DDS described in Section V.1.3 by are resolved presenting the architectural and design choices made for DDS4CIAO, which is our implementation of the DDS for Lightweight CCM specification outlined in Section V.1.2.

V.2.1 Accurate Indication of Successful Connector Configuration

The central difficulty outlined in **Challenge 1** from Section V.1.3 revolves around the final configuration stage of the D&C process. In this case, there lies a crucial phase before the application is "activated", but after it is fully configured. In this portion of the D&C process, the connector business logic must make themselves ready for execution by, for example, instantiating various DDS entities. In Figure V.1, which shows the lifecycle stages that connectors and components go through, this is represented by the "Passive" state. Unfortunately, the LwCCM specification currently provides no mechanism to communicate to the connector that it has entered this state; the only notification that is received when the component/connector becomes passive is when the prior state was "Active". To understand the reason for this, it is best to have a grasp of the layout of connectors and components at runtime.

Instantiated connectors consist of two primary pieces. First, there is a "Servant", which consists of the external CORBA interface and connector-specific container code. The Servant has two primary parts to its interface: (1) operations common to all connectors which come from the LwCCM specification (called the `CCMObject` interface), and (2) operations that result from the ports specified in the IDL declaration of the connector. Second

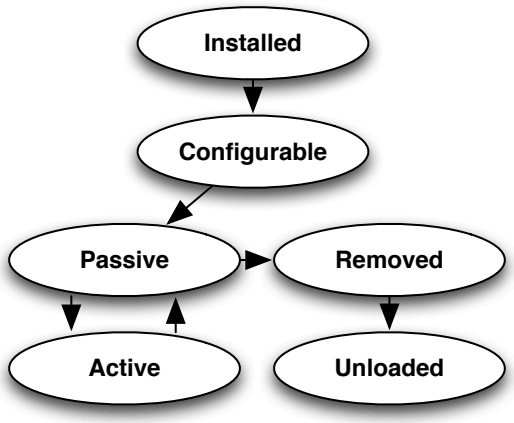

Figure V.1: LwCCM Component and Connector Lifecycle Stages

is the "Executor", which contains the actual business logic that implements the connector. Operations on this interface result from two sources: (1) specification-defined lifecycle operations (called the `SessionComponent` interface), and (2) operations that result from the ports defined for the connector.

The `configuration_complete` operation mentioned in Section V.1.3 is part of the `CCMObject` interface but is not, however, present on the `SessionComponent` interface so it cannot be directly delegated.[2] Unfortunately, the first lifecycle operation that is invoked on the Executor interface after its construction as defined by the LwCCM specification is `ccm_activate`. This lifecycle operation, however, must be disjoint from and occur later than `configuration_complete`.

One approach to work around this problem is to delay the creation of the DDS entities until the activation phase of the application lifecycle. This is problematic, however, because there exists no guarantee that a connector fragment will be activated *before* its connected component. If a component is activated before its connector and attempts to initiate outbound communication, that communication would naturally fail, potentially causing pernicious and difficult to reproduce errors. The ability for component business logic to

[2]This artifact results from the standards specification.

receive a notification upon configuration completion but before activation has proven to be useful for components as well as connectors because connectors are anyway treated as components.

As a result, a new interface that may be optionally used to extend the behavior of component executors to be able to receive these notifications has been created. This interface, which the `ConfigurableComponent` is called, uses a variation of the extension interface pattern to avoid changing the standard-defined `SessionComponent` interface. This new interface is intended to act as a mixin so that the component implementations wishing to receive `configuration_complete` will inherit from this in addition to the standard `SessionComponent` interface. The container, then, when it receives `configuration_complete` from the D&C tooling, will attempt a dynamic cast on the component implementation to determine if the operation should be delegated on a per-component basis.

V.2.2 Avoiding D&C-related Memory Footprint

Challenge 2, described in Section V.1.3, deals with eliminating unnecessary footprint from the specification-defined deployment and configuration requirements of connectors. DDS4CCM connectors are configured via attributes present in the IDL interfaces defined by the specification, which allow for the fragment to be associated with a particular DDS domain and topic as well as the QoS policies.

Many hardware platforms commonly used for DRE systems remain extremely memory-constrained, so the additional runtime memory footprint imposed by the CCM home is at best undesirable. To avoid this additional overhead, DDS4CIAO provides the capability to install “un-homed” components and connectors. These un-homed components are allocated from simple factory functions exported from their implementation libraries in much the same manner that Homes are already constructed. Component-specific container code,

which is generated automatically from IDL, is then able to interpret the D&C plan meta-data and individually invoke the attribute setter methods on the component.

V.2.3 Reducing Connector-Related Memory Footprint

The solution **Challenge 3**, described in Section V.1.3, attempts to reduce the runtime footprint of connector implementations. In order to accomplish this goal, it must be determined which, if any services that a component requires that are not necessary for connector implementations. Given the limitations of the standard LwCCM event ports described in Section V.1.2, it is highly unlikely that these inflexible port types would be used in the context of a connector — indeed, the extended port/connector infrastructure could be used to fabricate replacement infrastructure. Moreover, the DDS4CCM specification makes no use of the existing event infrastructure, making it an apt candidate for removal.

As a result, the event infrastructure was removed from the connector infrastructure in such a way that it would still be present for standard components that may need to interface with legacy systems. In this case, there are two pieces to the event support in DDS4CIAO: (1) the base classes that provide support to the component-specific generated container code, and (2) the component-specific generated container code itself, which includes a component-specific context that provides services to the component business logic. The first portion of the event support — the base classes described above were split into two pieces — a *connector* base and a *component* base. The container base contains all necessary functionality for component and connectors minus the LwCCM event support. The necessary plumbing LwCCM event support is contained in the component base, which derives from the connector base. This way the code generation infrastructure can choose to omit support for the event infrastructure if desired by selecting a different base class for the generated code. Our approach makes this artifact configurable.

V.2.4 Supporting Local Facets

The solution to **Challenge 4** outlined in Section V.1.3 is threefold. First, and most obviously, the Navigation and Introspection implementations generated for components with local facets and receptacles had to be modified to suppress any knowledge of these local ports. While this approach solves the issue of undefined behavior from trying to marshal one of these local object references, it also completely removes any standards-based mechanism by which a connection can be made by either the D&C tooling or any user attempting to use the Navigation interfaces. To address this undesired effect, a new connection API was created in the private interface to the CIAO container (which is our LwCCM implementation) that is used directly by the D&C tooling. This API accepts as arguments the string identifiers of two component endpoints as well as port names, and is able to use these to obtain references to the local Executor objects directly and create a connection without needing to marshal any local references over standard interfaces.

In order to make use of this new API, however, the D&C tooling needs an annotation on the connection meta-data so that it can be made aware that it should not attempt to use the standard Navigation API to make the connection. The data structure in the deployment plan that contains connection information encodes the type of connection (*e.g.*, Facet vs. Receptacle) as an enumerated value. While this enumeration could be extended to identify a new connection type (*i.e.*, LocalFacet), the changes to specification-defined types were minimized. The connection data structure does contain a section where requirements for deployments can be described using name/value pairs. This section would ordinarily be used to enumerate hardware capabilities or resources required by the connection. In this case, it is required that any local facet connected be annotated with a requirement on the container, namely that it provide support for local facets — when the D&C tooling encounters this annotation it assumes the connection to be local.

V.2.5 Ensuring Portability of DDS4CIAO Implementation

As described in **Challenge 5** from Section V.1.3, it is important to ensure that the design of the infrastructure is maximally portable in order to easily support implementations from multiple DDS vendors. This goal is complicated by the fact that despite the presence of a standard C++ language mapping, there are subtle and pernicious differences between the actual implementations of these mappings. Moreover, there exist also subtle behavioral differences between implementations that complicate source-level compatibility, *i.e.*, generated type-specific constructs such as `DataWriters` and `DataReaders` may have different namespaces and naming conventions, and indeed the same may be true of the entire API.

These challenges were addressed by using three approaches. The first approach targets the API that is implemented in the DDS4CCM basic ports against. The DDS specification, in addition to the widely supported C/C++ language binding, also has a language binding that maps the API into IDL interface definitions. This language binding is not widely implemented, but provides a promising vehicle for implementing portable DDS business logic in the context of the DDS4CCM basic ports. Since the same IDL code generator is used as with the rest of the CIAO infrastructure, that the APIs used to implement these ports are consistent.

Much of the work for supporting different DDS implementations then can be accomplished by providing an implementation of this IDL language binding. At first glance, this may seem a daunting proposition — however, this binding consists of only about 36 interfaces, many of whose functions may be directly delegated to the native implementation. The remaining problem with using this IDL-based approach is reconciling the differences between the CORBA types that are part of the IDL language mapping and the data types used natively by the DDS implementation. While this conversion could be handled inside the vendor-specific implementation of the IDL language binding, this approach would incur potentially expensive data copies. Fortunately, many DDS implementations provide a

CORBA compatibility layer that allows them to directly use types generated by the IDL compiler.

V.2.6 Connector Code Generation

Generating code for user-defined connectors is the focus of **Challenge 6** from Section V.1.3. Our experience developing code generators for our CORBA and LwCCM implementations has shown us that it is eminently undesirable to embed large amounts of business logic in generated code. This is largely due to the difficulty of maintaining and extending the code generators themselves. If there is a bug, modification, or extension to be made, this effort often involves at least two engineers — one who is familiar with the middleware or problem at hand, and another who is familiar with the process of extending and modifying the code generator. In addition to the extra personnel requirements, it often substantially increases the amount of time to test these changes, as not only does the initial proposed modification needs to be be tested (typically supplied to the code generation engineer as a handcrafted generated file), but also the final changes to the code generator and resulting modified output. For the same reason, this accidental complexity of the code generation process impedes the ability of users to create their own DDS4CCM connectors.

In order to avoid these accidental complexities, the design of the code generation infrastructure from the outset to contain zero DDS4CCM business logic and to be extensible **without** the need to modify the code generator to add new connector implementations. The first, and most obvious step given the presence of parameterized modules from Section V.1.2, was to leverage C++ templates for the implementations of the basic and extended DDS4CCM port types. Using C++ templates in this case allowed us to make generic two very important parts of the implementation — first, the core DDS4CCM business logic contained in the basic and extended DDS4CCM ports, but also the IDL wrapper (described

in Section V.2.5) around our target DDS implementation. These IDL wrappers require access to type-specific DDS entities (*e.g.* `DataWriters` and `Data Readers`) that are created by the code generation infrastructure that is part of the DDS implementation itself.

Connector implementations, then, are really a collection of template instantiations for the various basic and extended ports that are contained in their interface definition along with some configuration glue code. While source code for these connector implementations could be genreated, that would still represent an obstacle to novel connector creation. Connectors themselves may contain a nontrivial amount of configuration business logic that interprets the values of attributes on the connector interface. As a result, if a user were to define a new connector with new configuration attributes, they would be required to modify the code generator to be able to use their new connector.

To address this concern, the connector implementations template was made into classes as well. This allows the code generator for DDS4CCM to be extremely simple. In effect, the result of the code generation process is a header file that contains a set of C++ traits [44] which specify the properties necessary to use a particular IDL data type. These properties largely consist of the names of type-specific entities that are generated from the DDS infrastructure. These traits are then used to create concrete template instantiations of any required connector implementations. By default, instantiations of the standard DDS4CCM connectors are generated — the State and Event connectors described in Section V.1.3. If a user defines their own connector in IDL, the code generator emits an include of a header file whose name derives from the name of the connector in IDL, and a concrete instantiation of a template class whose name is similarly derived. While the user must then provide an implementation of this template class, this is substantially less effort than would be required to modify the code generator.

V.3 Experimental Results

This section outlines two key empirical observations of the DDS4CIAO implementation described in Section V.2 which cover two important goals outlined in Section V. First, in Section V.3.2, the impact that the code generation capabilities of DDS4CIAO have on the development and maintenance of DDS-enabled applications is quantified. Second, in Section V.3.3, we characterize the overhead that DDS-enabled applications must pay in terms of latency when using the DDS4CIAO abstraction versus using the DDS API directly.

V.3.1 Experimental Scenario

All results described below were obtained using a simple "ping-pong" application. A simple example was chosen since the business logic of the application is not important to evaluate the qualities of DDS4CIAO. Rather, understanding the overhead associated is interesting, if any, of the integration of LwCCM with DDS. In this application, an instance struct containing an octet sequence of a configured length and a sequence number would be written to the DDS data space by a "Sender". The instance would arrive at a "Receiver" entity, after which a new instance of the struct would be published on a separate topic with an identical sequence number but a zero length octet sequence. The "Sender", upon receipt of the second message, repeats the process with a new sequence number up to a specified number of iterations.

Two versions of this application were produced. The first uses the native C++ DDS API, with all customary error checking included. In the second version, the "Sender" and "Receiver" were each implemented as CIAO components and used DDS4CIAO to interface with the DDS middleware.

V.3.2 Evaluation of Code Generation

To evaluate the effectiveness of the code generation techniques described in Section V.2.6, the implementation source files from the experimental scenario outlined in Section V.3.1

were analyzed with the SLOCCount [81] tool. This is a program which counts physical Source Lines of Code (SLOC), and uses a number of heuristics to discard any whitespace and commenting present. For the purposes of this evaluation, only implementation source files were counted, discarding header files containing only class definitions. The reason for this is that header files for the DDS4CIAO implementation are largely generated automatically based on the class interfaces.

The results from this tool are summarized in Table V.1. If only the total SLOC for the native programs and the component implementations are compared, DDS4CIAO shows only a nominal improvement over that of the native implementation. It is important to consider, however, that the DDS4CIAO implementation contains a large amount of generated class skeletons which are created from the IDL interface descriptions from the component automatically (SLOC for which is shown in the "DDS4CIAO Generated" column of the table). When these lines of code are subtracted from the total for the DDS4CIAO implementation, the improvement becomes substantially more dramatic. In the case of the Sender component, the improvement is on the order of 50%, and for the receiver the difference is an order of magnitude. The reason for this discrepancy is the Sender programs — both native and DDS4CIAO — contains a substantial amount of code in common to measure latencies and calculate/display results.

Table V.1: Comparison of Source Lines of Code

Component	Native Lines	DDS4CIAO Total	DDS4CIAO Generated	DDS4CIAO Actual
Sender	643	560	211	349
Receiver	293	128	118	10

V.3.3 Evaluation of the Overhead of DDS4CIAO

To evaluate the overhead due to abstraction over the native DDS API introduced by the DDS4CIAO implementation, the experimental scenario described earlier in Section V.3.1

was used to evaluate the latency performance using a recent commercial DDS implementation and DDS4CIAO 0.8.3. Each configuration was executed for 1,000 iterations each with payload sizes along powers 2, from 16 to 8192 bytes. Each experimental run was executed in two transport configurations: once using UDP and again using Shared Memory transport. The experimental testbed consisted of Dell Optiplex 755 computers, with an Intel E4400 CPU, 2GB of RAM, and gigabit network connections.

The results for the experimental runs with the UDP transport protocol are shown in Figure V.2, which compares the average latency for each payload size, and Figure V.3, which compares the minimum latency results for each payload size. These results show that for this transport protocol, the average latencies are nearly identical. Figure V.4 shows the results from the experimental runs configured with the shared memory transport. This average latency result shows that the DDS4CIAO abstraction introduces approximately a four percent overhead over the native implementation for the shared memory transport. The best case results for the shared memory experiment are shown in Figure V.5.

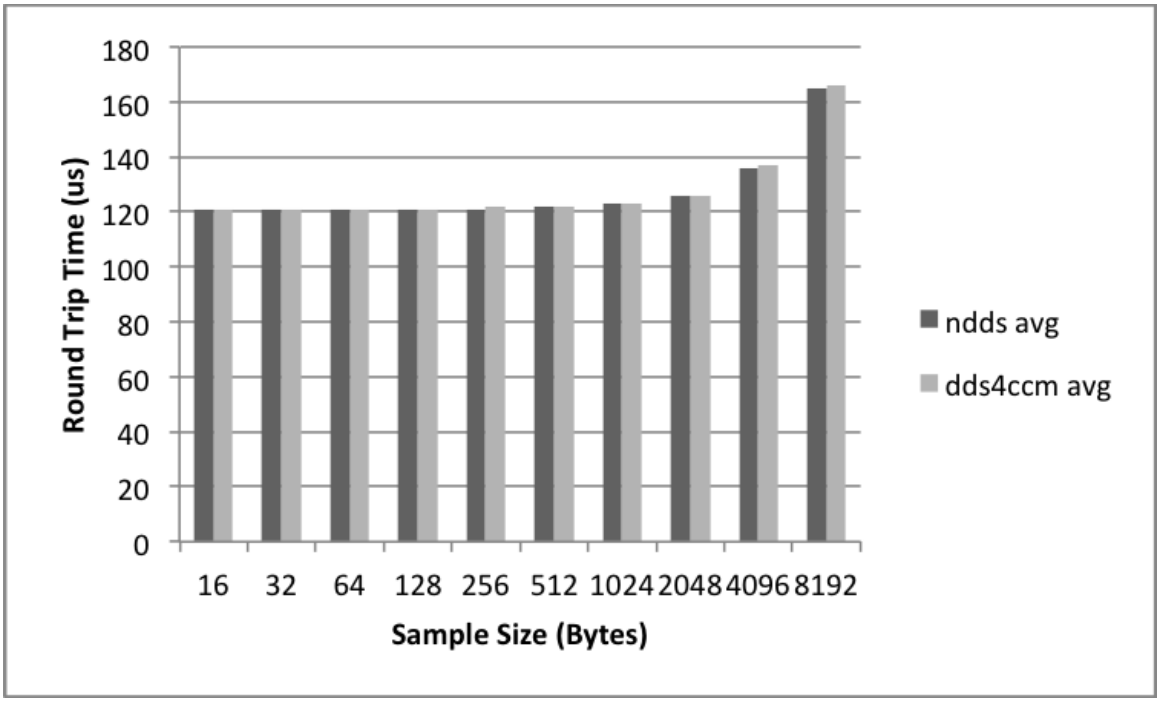

Figure V.2: Ping Latency Average with UDP

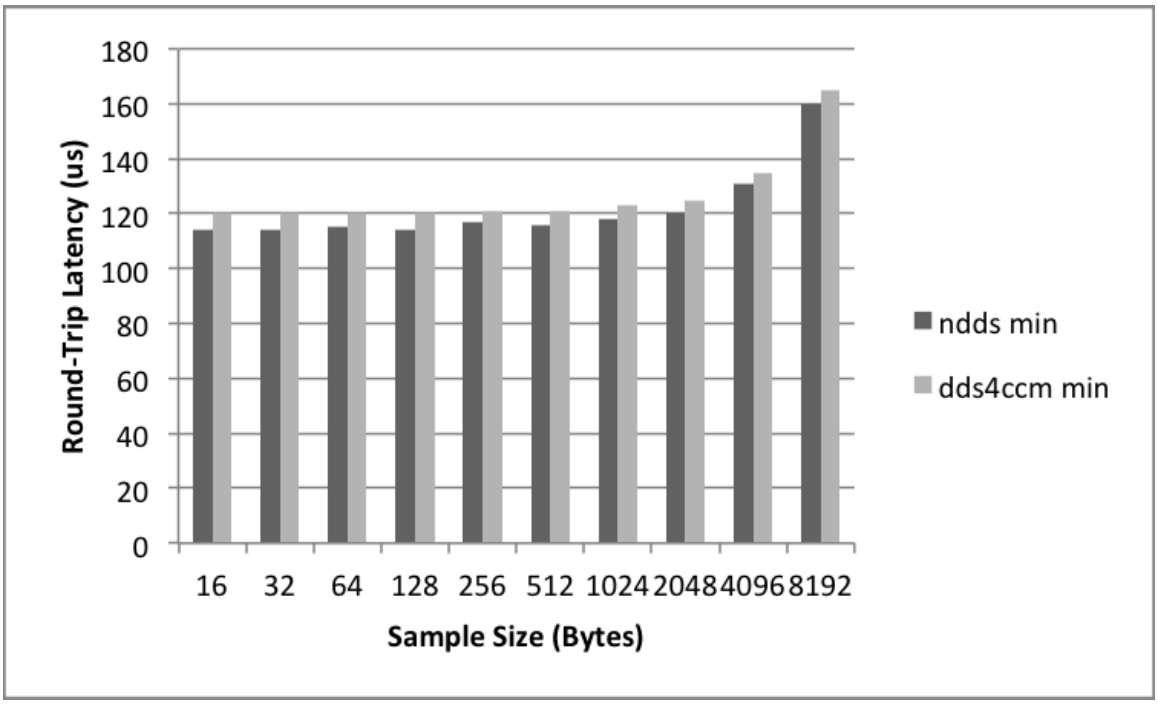

Figure V.3: Ping Latency Minimum with UDP

Table V.2 summarizes the standard deviation of the experimental runs for both UDP and shared memory. These results show that the DDS4CIAO abstraction does not introduce additional jitter over the native implementation.

Table V.2: Standard Deviation For All Experiments

Size	UDP	CIAO UDP	Shared	CIAO Shared
16	11.3	12.4	17.7	18.4
32	12.4	9.4	15	14.2
64	12.5	12.6	15.5	9.9
128	13.3	9.3	16	10.4
256	6.2	13.1	15.9	12.6
512	12.3	11.2	11.6	8.8
1024	14.7	8.1	15.7	12.1
2048	12.7	4.3	15.5	14.8
4096	7.1	13.7	15.3	10.8
8192	12.1	17.7	15.1	14.4

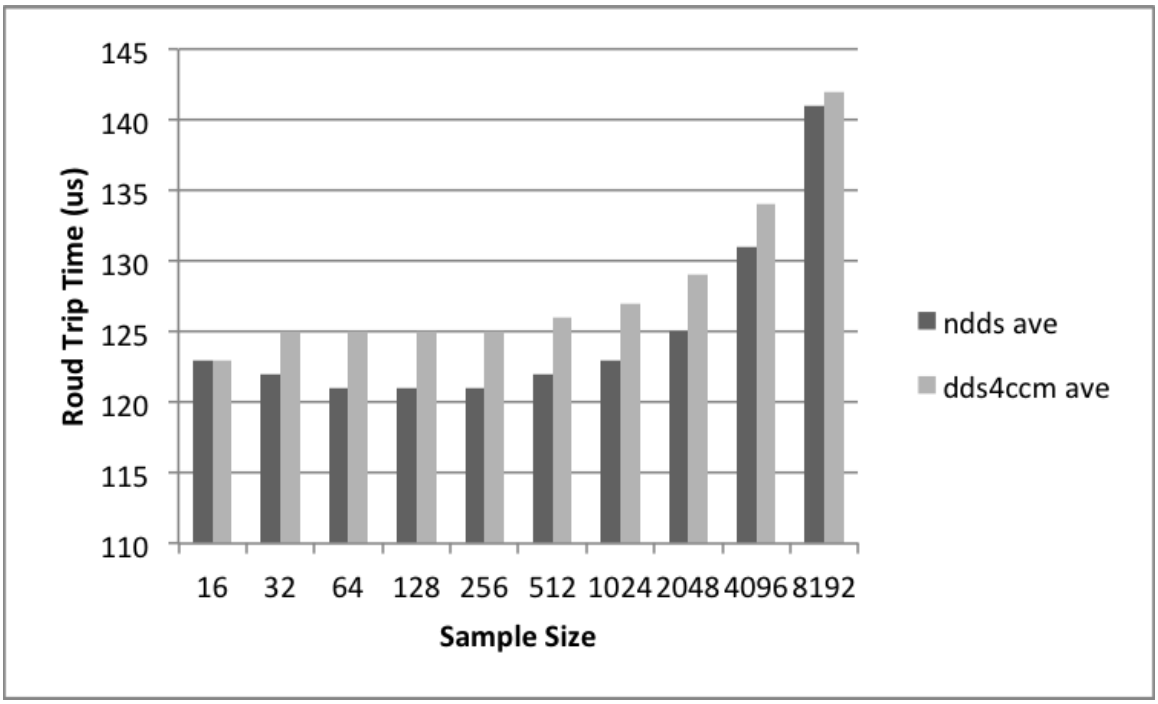

Figure V.4: Ping Latency Average with Shared Memory

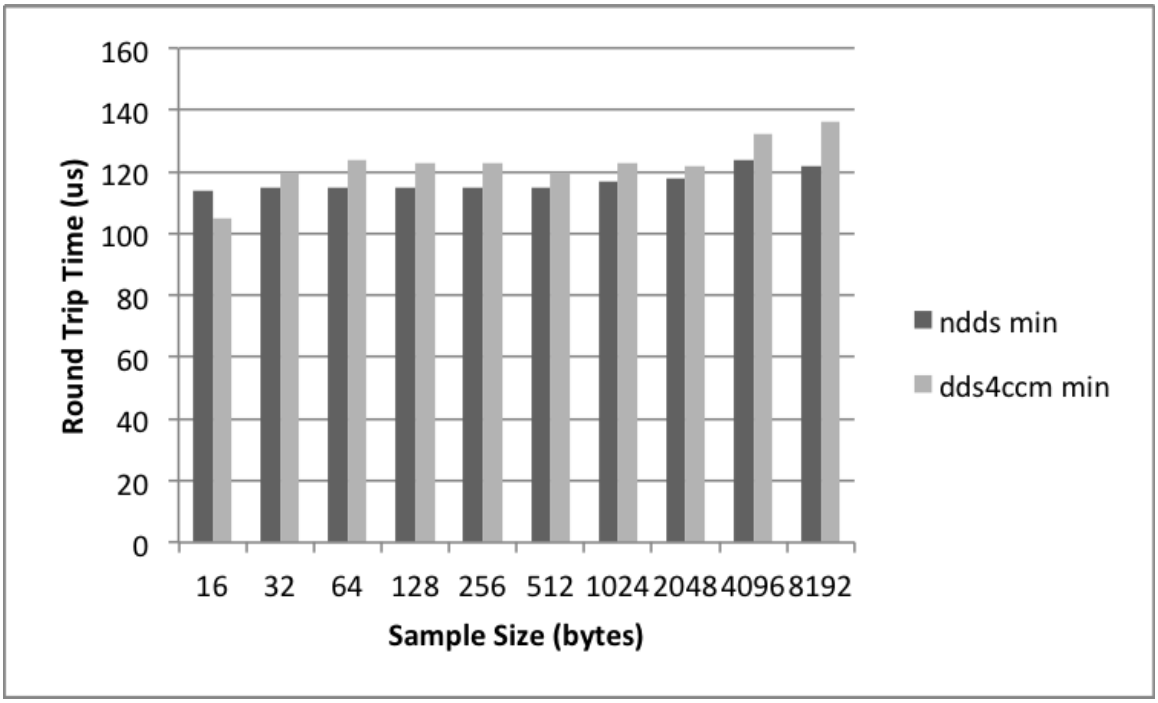

Figure V.5: Ping Latency Minimum with Shared Memory

V.4 Related Work

This section compares our research on component-based DDS with related work.

PocoCapsule [33] is an Inversion of Control container based on the Dependency Injection (DI) design pattern. This component framework allows developers to use "Plain Old C++ Objects" (POCO) that have been decorated with PocoCapsule macros that allow the loading of these C++ classes into a PocoCapsule container. DDS4CCM and DDS4CIAO differ in several important aspects from PocoCapsule. First, DDS4CCM—and LwCCM in general—are industry standards that have language bindings defined for many programming languages. Second, PocoCapsule still requires some amount of low-level glue code in the component business logic. Third, the DDS for PocoCapsule implementation currently only uses CORBA local interfaces to simulate small parts of the DDS API, and hence is not operable with standard-compliant DDS implementations.

Simple API for DDS (SimD) [4] uses C++ templates and template meta-programming to provide a simpler API for DDS that reduces the amount of infrastructure-related code required for DDS applications by an order of magnitude. Using SimD, a simple DDS application can be written in only 4 source hand-written lines of code, instead of dozens lines of code using the native API. While SimD reduces the complexity of the boilerplate code required for DDS applications, it differs substantially from DDS4CIAO in that it does not address runtime deployment and configuration capabilities provided by DDS4CIAO. Moreover, it has not yet been proposed as a standard.

Researchers at Real-Time Innovations, Inc [2] propose extensions to the DDS API to allow declarative configuration of DDS entities via an XML file that is interpreted at runtime. The application then queries the DDS middleware to obtain a particular `DataReader` or `DataWriter` that has been configured already with a domain and topic binding and QoS settings. While their work improves the state-of-the-practice in standards-based DDS application configuration, its capabilities are not as extensive as DDS4CCM and DDS4CIAO.

First, our existing D&C tooling provides coordinated installation of application implementations and startup across multiple nodes. Second, the connector infrastructure developed for DDS4CIAO allows integration with other distribution middleware, such as CORBA, TENA, JMS, or even socket based network programs. Third, the decoupling provided by the DDS4CIAO implementation enables the selection of DDS implementation at deployment time.

SOFA [9, 10] is a component model with an integrated D&C framework that provides remote communication capabilities via a connector infrastructure similar in spirit to that which is part of the DDS4CCM specification. SOFA, however, only provides connectors for CORBA and RMI distribution middleware. Our approach differs from that taken by SOFA in that the connectors implemented by DDS4CIAO are themselves lightweight components. The advantage of our approach is that any improvements to the QoS capabilities of the CIAO container can be automatically applied not only to all components deployed, but also connectors as well.

V.5 Summary and Lessons Learned

The experience developing applications with DDS4CIAO provided the basis for the following lessons learned:

1. **Substantially reduced DDS application complexity.** Tests and example applications developed with DDS4CIAO have shown that the simplified interface to the underlying DDS middleware, provided by the DDS4CCM specification, provides a platform that easier to write and develop DDS applications.

2. **Automatic configuration of DDS middleware.** By providing a strict separation of concerns between configuration-based aspects of DDS application development

and configuration aspects, users can automatically configure the underlying middleware at deployment time using standards-based deployment plan descriptors already available with LwCCM.

3. **Deployment-time binding of DDS implementation may ease application benchmarking.** It is also possible that the DDS implementation used by the component application could be chosen at deployment time, rather than compile time. This enhancement will allow developers to evaluate the merits and performance characteristics of different DDS implementations rapidly.

4. **Increased reliance on tooling.** A consequence of developing with DDS4CIAO is the increased reliance on tooling, especially modeling tools. While writing the IDL and business logic for DDS4CIAO components is straightforward, writing the deployment descriptors by hand is a difficult task that requires expert knowledge of the D&C specification. While the use of modeling tools — such as our CoSMIC toolsuite [40] or commercial tools that have emerged — can substantially ameliorate this concern, their use may not always be practical (CoSMIC, for example requires Windows while the commercial tools may be costly). A domain specific language (DSL) for describing deployments, configuration, and component packaging would substantially reduce the modeling requirement.

5. **Applying connectors to the CCM CORBA infrastructure**. The connector-based approach to integrating the DDS distribution middleware into CIAO has shown substantial promise. Unfortunately, however, the CORBA infrastructure that underlies CIAO/CCM still remains tightly integrated into the container implementation. There are many users and applications who find this situation undesirable for political, security, and runtime footprint of the middleware.

A similar connector based approach could be used to convert LwCCM into a “Common Component Model”, which is completely agnostic to the underlying communications middleware, by moving all of the extant CORBA communications functions to connectors themselves. This approach has the advantage of not only being able to remove the CORBA infrastructure currently used for synchronous two-way communication, but also makes it possible to, for example, swap in an alternative non-CORBA based connector implementation, if desired.

CHAPTER VI

FUTURE RESEARCH DIRECTIONS

VI.1 Deployment and Configuration of Cloud-based Applications

Cloud computing, a paradigm whereby computing resources (CPU, RAM, disk space) are provisioned on-the-fly and on-demand, and in a potentially elastic manner, is an increasingly popular deployment environment for large-scale applications [11, 45]. An attractive feature of cloud computing is that it both increases flexibility by allowing the virtualization of the hardware resources and enabling on-demand scaling of application performance while at the same time relieving the administrative overhead associated with managing and administering the associated physical hardware and software resources that are required for an application. These qualities provide a compelling way to expand the capabilities of DRE systems and applications where it may be practically or cost prohibitive to provision physical resources for intermittently needed resource-intensive computing.

Deployment and configuration solutions for the Cloud environment are an active area of research and development, however, many existing solutions are inappropriate for DRE systems. Commercial cloud providers such as Amazon Web Services (AWS) [3] and Rackspace Cloud Hosting [68] provide one of three options for deploying applications into cloud environments. The first option, such as the AWS Elastic Beanstalk, require applications to use either special APIs or application containers provided by the service to achieve automatic deployment. Moreover, these APIs and application containers tend to be specialized for web services and not DRE-type applications. Finally, the second option is to use purpose built grid/cloud computing APIs such as MPI [5] or Map-Reduce [12], which are not appropriate for all applications and require that the applications themselves be written specifically to be used in cloud environments.

The final option provided for application configuration is to provision a virtual machine instance with the application software required and transfer that image to the cloud provider. The application then, is required to perform any configuration on its own, potentially relying on a proprietary, ad-hoc mechanism. This can be a problematic approach for two important reasons. First, provisioning a virtual machine can be problematic, at the moment often requiring some level of human intervention and large amounts of data transfer to the cloud provider. Second, proprietary ad-hoc deployment and configuration systems must constantly re-invent the wheel and are not easily useable for other applications.

VI.1.1 Unresolved Challenges

This section will outline two unresolved challenges that arise when applying a generic deployment and configuration toolchain to a Cloud-based environment.

VI.1.1.1 Hierarchical Domains

In the context of deployment and configuration systems, a critical element of the framework is the characterization of the "domain" in which it operates. "Domain" in this case refers to the collection of hardware resources in which applications are deployed — this includes especially the physical hardware on which applications run, but also includes elements such as interconnects and bridges that make up the connectivity resources used by applications. Many D&C toolchains, including the OMG Deployment and Configuration specification, the SOFA component model [10], and deployment solutions for the Enterprise Java Beans [1] maintain only a flat representation in which the global infrastructure not only has full knowledge of all nodes in the domain, but is responsible for coordinating all deployment activity amongst them.

This flattened representation causes two significant problems when these D&C frameworks are applied to Cloud domains. First, due to the nature of Cloud infrastructure, the

requested hardware resources will not be collocated with the pieces of the global infrastructure used to initiate deployment, and such requests may have to traverse a wide area network such as the commodity Internet. This can cause problematic spikes in deployment latency due to communications latency and bandwidth limitations that may be present over a WAN. Second, since it may be difficult to ascertain *a priori* the hardware resources that are required or even available to the application, it may be impossible to properly configure the global infrastructure to discover these resources.

VI.1.1.2 Deployment Toolchain Installation

As discussed in Section VI.1, the current best practice for managing deployment of software into Cloud environments is to create a virtual machine image with all required software. This virtual machine is then transferred to the Cloud provider which uses this image to provision all allocated nodes. Managing such virtual machine images can be a challenging task, not only from the standpoint of ensuring that all required component implementations are available in the image, but also ensuring that the required version of the deployment infrastructure is present, along with any needed plugin functionality (*e.g.*, Installation Handlers and Deployment Portable Interceptors).

While LE-DAnCE currently has functionality that allows it to download implementation artifacts (*i.e.*, shared libraries that implement components) that are required at deployment time, it is currently not possible to use LE-DAnCE to bootstrap itself by copying its binaries and invoking the appropriate daemon processes.

VI.1.2 Solution Approach

The LocalityManager framework provides a different architectural way to visualize the deployment process from the one outlined in Section III.1.1. Instead of viewing the process of application deployment as the concrete establishment of component instances on

individual nodes as the primary goal for application deployment, we can view the deployment process as the establishment of various *localities* in different *contexts*. This view of deployment, shown in Figure VI.1, represents a novel way of viewing the deployment process.

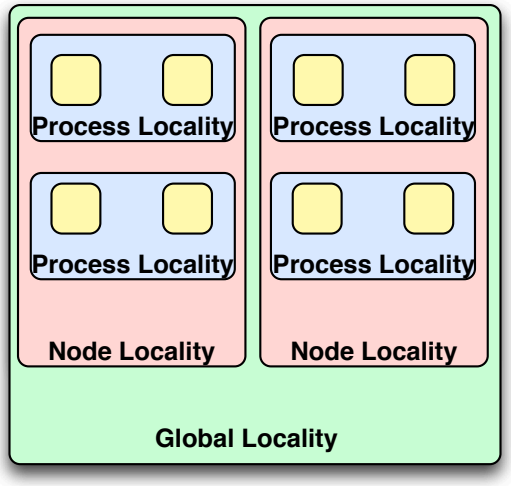

Figure VI.1: Locality-Based View of Deployment

While the architecture illustrated by this figure appears to be similar to the current LEDAnCE architecture, it contains an important distinction. Instead of viewing the ultimate object of deployment action as a "component instance," at each level we view the establishment of the next level of localities in the hierarchy. To put it another way, under the current methodology the only instances that appear in the plan are concrete component instances. Under the proposed view of the deployment process, *everything is represented as an instance*. Instead of having only eight instances in the plan for Figure VI.1 for each of the four yellow boxes that represent components, we would have instead 14 described instances in the plan.

Eight of the instances, as before, would represent components. These component instances would then be grouped (based on meta-data tagging) with an appropriate instance

representing the *process locality*, which could have its own configuration directives to load appropriate installation handlers or QoS configuration. Each of these *process locality* instances would be grouped with one of two *node locality* instances, which again could be individually configured. This approach allows more flexibility in how domains are assembled by allowing to not have any knowledge of how the remainder of the application is deployed (or how the domain is structured) beyond how the next locality level needs to be established.

A natural implementation approach for this new domain view would be to use the LocalityManager at all levels of the deployment hierarchy. Rather than have purpose built daemons that implement the roles of *ExecutionManager* and *NodeManager*, a *LocalityManager* instance could be configured with an appropriate Installation Handler implementation that provided the appropriate knowledge of how to establish the next level of the domain hierarchy. In the case of the *ExecutionManager*, this would include knowledge of how to contact a *LocalityManager* daemon running on a particular node to pass off the appropriate locality-constrained plan. In the case of the *NodeManager*, this would include the process of spawning new *LocalityManager* instances that would manage the process localities.

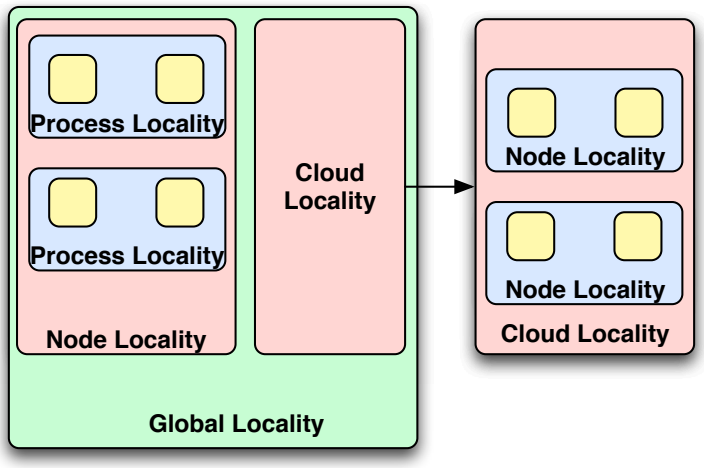

Figure VI.2: Locality-Based View of a Cloud Deployment

Figure VI.2 shows how this new domain hierarchy might be used in the context of a mixed standard/cloud deployment. In this case, there would be two localities considered by the global deployment infrastructure: (1) a standard node locality instance, and (2) a cloud locality with the configuration information necessary to dynamically request Cloud infrastructure be provisioned. An installation handler could then be loaded that would be able to use this configuration meta-data along with web service APIs exposed by many commercial cloud providers to automatically provision the necessary infrastructure.

Moreover, this approach can also be used in concert with the Instance Installation Handlers (IIH) and the LocalityManager architecture described in Section III.3.1 to address the challenge outlined in Section VI.1.1.2. By treating individual nodes required for a deployment as instances in the deployment plan, we are then able to use an IIH to perform any installation measures required. For example, an IIH could be created that would first check to see if any DAnCE infrastructure was present on the target node. If no infrastructure is detected, it would be able to then use readily available and mature remote management interfaces, *e.g.* SSH, to transfer the appropriate binary implementation of DAnCE and start up the necessary infrastructure.

VI.2 Real-Time Extension for CCM

Developing complex component applications for the domains described in the introduction is a very complex task due to the stringent Quality of Service requirements found in Open DRE environments. Prior work conducted by our research group [78, 79, 80] has demonstrated the need for configurable QoS properties to be integrated into both the component middleware used to implement the applications and the deployment infrastructure used to deploy and configure the application. Ideally, such configurable QoS properties should be configured in such a way that (1) systemic aspects of using these QoS properties

are explicitly separated from application business logic, and (2) such properties are explicitly configurable outside the application at runtime in order to allow tuning and adaptation of the application.

Prior implementations of RT-CCM produced in the context of CIAO and DAnCE were very tightly integrated into both the DAnCE tooling and the instantiation and installation process of CIAO containers and components. This approach is problematic for three reasons. First, such tight integration in the critical path complicates the implementation of the D&C toolchain, markedly increasing the difficulty of maintaining the functionality — *e.g.*, due to this tight integration and complex maintenance, RT-CCM functionality was unable to be transitioned to the LE-DAnCE infrastructure. This integration also increases the difficulty of applying the paralellization optimizations discussed earlier in this chapter. Second, this approach makes it markedly more difficult to integrate configurability for other middleware services into the D&C toolchain, *i.e.,* infrastructure level load balancing or fault tolerance services. Finally, its presence in the critical path makes it impossible to remove this functionality during deploy applications for which RT functionality is neither required nor appropriate, *i.e.,* deployments of non-CCM based applications.

Ideally, such QoS configuration should be accomplished using a pluggable architecture that uses standard interfaces to extend the functionality of both the component middleware implementation and the deployment infrastructure to support QoS configuration. While the Quality of Service for CCM specification [54] (QoS4CCM) has emerged to provide some extensibility to the CCM container in order to support some level of QoS configuration and could be used to also implement other extensions such as security controls and fault tolerance, the specification falls short in at least one important area. The pluggable infrastructure provided by the QoS4CCM specification is entirely focused on modifying the behavior of the container and components at runtime, *i.e.,* after the application has been activated, and makes no provision for influencing the initial configuration during installation and configuration by the D&C framework.

CHAPTER VII

CONCLUDING REMARKS

This dissertation has presented deployment and configuration research challenges in three areas: 1) resource constrained sensor webs, 2) adaptive and heterogeneous deployment in distributed real-time and embedded systems, 3) deterministic and efficient deployment and configuration of large-scale systems, and 4) simplified integration of distribution middleware into component middleware.

The challenges in the area of resource constrained sensor webs revolved around the deployment and configuration of the *Multi-agent Architecture for Coordinated Responsive Observations* (MACRO), an agent-based middleware platform implemented with component middleware. In particular, these challenges involved the ability of the MACRO framework to 1) be able to execute low-level hardware dependent tasks, and 2) be provisioned with the necessary business logic to be able to effect these actions at runtime. The solution to these two challenges, the Action/Effector framework, was described. Moreover, the severe resource limitations in this domain — especially CPU, memory, and power — caused challenges in the context of DAnCE, particularly the ability of DAnCE to deal with power saving measures that involved completely shutting down hardware and the ability to correctly recover a correct deployment upon reboot. Finally, the effort to resolve the CPU and memory constraints in this domain in part motivated the creation of the LocalityManager framework in an effort to address footprint and deployment latency challenges.

The LocalityManager framework is part of an improved version of the DAnCE framework called Locality-Enhanced DAnCE (LE-DAnCE). LE-DAnCE contains a number of compelling contributions to the state of the art for deployment and configuration toolchains for DRE systems. First, it allows for deployment of heterogeneous applications via the Instance Installation Handler (IIH) facility, *i.e.*, allowing for multiple component models

to be part of a single deployment. Second, it allows user customization of the deployment toolchain through a well-defined interface via the Deployment Portable Interceptor (DPI) facility. Third, it allows for these extensions (IIH and DPI implementations) to be loaded dynamically at runtime as needed. Finally, due to the strict separation of concerns enabled by the IIH and DPI infrastructure, it contains a Deployment Scheduler that allows for maintainable and extensible parallel scheduling of deployment events. The strict separation of concerns provided by the LocalityManager also allows for a number of important performance optimizations that allow it to scale to very large-scale deployments.

Finally, DDS4CIAO is a novel generative approach for developing DDS-based component-oriented DRE systems. This approach combines key advantages of the DDS middleware, such as low latency communication and extensive QoS policy support, with the strengths of a mature component model, such as simplified application composition and automatic deployment and configuration. The approach has been prototyped and evaluated via the DDS4CIAO middleware platform, which implements the Lightweight CCM (DDS-4CCM) specification, while addressing a number of inherent and accidental complexities in integrating the DDS and LwCCM technologies. In particular, extensive use of variants of the extensible interface pattern have been made to extend the existing standard-defined LwCCM interface and deployment meta-data to overcome incompatibilities between DDS and LwCCM and overcome oversights in the DDS4CCM specification. Additionally, a template driven code generation technique has been described that maximizes portability amongst DDS implementations while allowing users to extend DDS4CCM by defining their own connector types without having to modify the code generator.

CIAO, LE-DAnCE, and DDS4CIAO are all open source software and are available from `download.dre.vanderbilt.edu`

APPENDIX A

LIST OF PUBLICATIONS

Refereed Journal Publications

J2 M. Stal, D. C. Schmidt, and W. R. Otte. Efficiently and transparently automating scalable on-demand activation and deactivation of services with the activator pattern. *Software Practice and Experience, Special Issue on Pattern Languages: Addressing Challenges*, 41(10), Oct. 2011

J1 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

Refereed Conference Publications

C9 A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. R. Otte, J. Parsons, C. Szabó A. Coglio, E. Smith, and P. Bose. A software platform for fractionated spacecraft. In *Aerospace Conference, 2012 IEEE*, march 2012. to appear

C8 W. R. Otte, A. Gokhale, and 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, pages 21–30, New York, NY, USA, 2011. ACM

C7 W. R. Otte, A. Gokhale, D. C. Schmidt, and J. Willemsen. Infrastructure for component-based dds application development. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 53–62, New York, NY, USA, 2011. ACM

C6 J. S. Kinnebrew, W. R. Otte, N. Shankaran, G. Biswas, and D. C. Schmidt. Intelligent resource management and dynamic adaptation in a distributed real-time and embedded sensor web system. In *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '09, pages 135–142, Washington, DC, USA, 2009. IEEE Computer Society

C5 W. Otte, J. Kinnebrew, D. Schmidt, and G. Biswas. A flexible infrastructure for distributed deployment in adaptive sensor webs. In *Aerospace conference, 2009 IEEE*, pages 1 –12, march 2009

C4 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

C3 D. Suri, A. Howell, N. Shankaran, J. Kinnebrew, W. Otte, D. C. Schmidt, and G. Biswas. Onboard Processing using the Adaptive Network Architecture. In *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, College Park, MD, June 2006

C2 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

C1 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

Refereed Workshop Publications

W1 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)*, Bengaluru, India, Nov. 2010

Book Chapters

BC1 W. Otte and D. C. Schmidt. Labor-Saving Architcture: an Object-Oriented Framework for Networked Software. In *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. O'Reilly Media, 1st edition, July 2007

Submitted for Publication

S1 William R. Otte, Aniruddha Gokhale, and Douglbas C. Schmidt. Efficient and Deterministic Application Deployment in Component-based, Enterprise Distributed, Real-time, and Embedded Systems. Submitted to the Elsevier Information and Software Technology journal.

APPENDIX B

IDL LISTINGS

B.1 LocalityManager IDL

```
1  module DAnCE
2  {
3    /**
4     * @interface InstanceDeploymentHandler
5     * @brief Interface used to manage the lifecycle of instances.
6     *
7     * This interface is used by the LocalityManager to manage the lifecycle
8     * of various instance types.  Each instance type requires a seperate IDH.
9     */
10   local interface InstanceDeploymentHandler
11   {
12     readonly attribute string instance_type;
13
14     readonly attribute ::CORBA::StringSeq dependencies;
15
16     void configure (in ::Deployment::Properties config);
17
18     void install_instance (in ::Deployment::DeploymentPlan plan,
19                            in unsigned long instanceRef,
20                            out any instance_reference)
21       raises (Deployment::StartError,
22               Deployment::InvalidProperty,
23               Deployment::InvalidNodeExecParameter,
24               Deployment::InvalidComponentExecParameter);
25
26     void provide_endpoint_reference (in ::Deployment::DeploymentPlan plan,
27                                      in unsigned long connectionRef,
28                                      out any endpoint_reference)
29       raises (Deployment::StartError,
30               Deployment::InvalidProperty);
31
32     void connect_instance (in ::Deployment::DeploymentPlan plan,
33                            in unsigned long connectionRef,
34                            in any provided_reference)
35       raises (Deployment::StartError,
36               Deployment::InvalidConnection);
37
38     void disconnect_instance (in ::Deployment::DeploymentPlan plan,
39                               in unsigned long connectionRef)
40       raises (::Deployment::StopError);
41
42     void instance_configured (in ::Deployment::DeploymentPlan plan,
43                               in unsigned long instanceRef)
44       raises (Deployment::StartError);
45
```

```
46      void activate_instance (in ::Deployment::DeploymentPlan plan,
47                              in unsigned long instanceRef,
48                              in any instance_reference)
49        raises (Deployment::StartError);
50
51      void passivate_instance (in ::Deployment::DeploymentPlan plan,
52                               in unsigned long instanceRef,
53                               in any instance_reference)
54        raises (Deployment::StopError);
55
56      void remove_instance (in ::Deployment::DeploymentPlan plan,
57                            in unsigned long instanceRef,
58                            in any instance_reference)
59        raises (::Deployment::StopError);
60
61      /// Instruct the handler to release any resources prior to deallocation.
62      void close ();
63    };
64
65    interface LocalityManager :
66      Deployment::Application,
67      Deployment::ApplicationManager
68    {
69      readonly attribute ::Deployment::Properties configuration;
70
71      Deployment::ApplicationManager
72        preparePlan (in Deployment::DeploymentPlan plan,
73                     in Deployment::ResourceCommitmentManager resourceCommitment)
74        raises (Deployment::StartError,
75                Deployment::PlanError);
76
77      void destroyManager (in ::Deployment::ApplicationManager manager)
78        raises (Deployment::StopError);
79
80      oneway void shutdown ();
81    };
82
83    local interface LocalityConfiguration
84    {
85      readonly attribute string type;
86
87      void configure (in ::Deployment::Property prop);
88    };
89
90    interface LocalityManagerActivator
91    {
92      void locality_manager_callback (in LocalityManager ref,
93                                      in string uuid,
94                                      out Deployment::Properties config);
95
96      void configuration_complete (in string uuid);
97    };
```

REFERENCES

[1] A. Akkerman, A. Totok, and V. Karamcheti. Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments. In *3rd International Working Conference on Component Deployment (CD 2005)*, pages 17–32, Grenoble, France, Nov. 2005.

[2] Alejandro de Campos Ruiz and Gerardo Pardo-Castellote and GianPiero Napoli and Fernando Crespo-Sanchez and Javier Sanchez Monedero. High-level Programming of DDS Systems. In *Proceedings of the OMG Annual Real-time and Embedded Systems Workshop (RTWS)*, Arlington, VA, Mar. 2011.

[3] Amazon.com. Amazon Web Services. `aws.amazon.com`, 2010.

[4] Angelo Corsaro. Simple API for DDS. `http://code.google.com/p/simd-cxx/`.

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

[6] 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.

[7] A. Bouguerra and L. Karlsson. Hierarchical Task Planning Under Uncertainty. *3rd Italian Workshop on Planning and Scheduling (AI* IA 2004). Perugia, Italy*, 2004.

[8] 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.

[9] L. Bulej and T. Bures. A connector model suitable for automatic generation of connectors. Technical report, 2003.

[10] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[11] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. `http://labs.google.com/papers/mapreduce.html`, 2004.

[13] K. Delin and S. Jackson. Sensor web for in situ exploration of gaseous biosignatures. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 7, pages 465 –472 vol.7, 2000.

[14] 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.

[15] G. Deng, D. C. Schmidt, and 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*, pages 317–321, Washington, DC, USA, 2008. IEEE Computer Society.

[16] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. R. Otte, J. Parsons, C. Szabó A. Coglio, E. Smith, and P. Bose. A software platform for fractionated spacecraft. In

Aerospace Conference, 2012 IEEE, march 2012. to appear.

[17] A. El-Kholy and B. Richards. Temporal and Resource Reasoning in Planning: The parcPLAN Approach. pages 614–618. Wiley\ & Sons, 1996.

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

[19] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computer Survey*, 35:114–131, June 2003.

[20] D. R. Fatland, M. J. Heavner, E. Hood, and C. Connor. The SEAMONSTER Sensor Web: Lessons and Opportunities after One Year. *AGU Fall Meeting Abstracts*, pages A3+, Dec. 2007.

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

[22] P. Friedland and Y. Iwasaki. The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, 1(2):161–208, 1985.

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

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

[25] 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 Embedded 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.

[26] P. Haddawy, A. Doan, and R. Goodwin. Efficient decision-theoretic planning: Techniques and empirical analysis. In *In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 229–236. Morgan Kaufmann, 1995.

[27] G. T. Heineman and B. T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.

[28] J. Hill, D. C. Schmidt, J. Slaby, and A. Porter. CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments. In *Proceedings of the 15th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, Belfast, Northern Ireland, Apr. 2008.

[29] P. HnÄŻtynka and F. PlÃąÅąil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proceedings of CBSE 2006, Vasteras, Sweden, LNCS 4063*, pages 352–359. Springer-Verlag, 2006.

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

[31] L. H. Ihrig and S. Kambhampati. Design and implementation of a replay framework based on a partial order planner. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 1*, AAAI'96, pages 849–854. AAAI Press, 1996.

[32] T. Kalibera and P. Tuma. Distributed component system based on architecture description: The sofa experience. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS*

and ODBASE 2002, pages 981–994, London, UK, UK, 2002. Springer-Verlag.

[33] Ke Jin. Component-Based CORBA+DDS Applications in PocoCapsule vs CCM. http://www.pocomatic.com/docs/whitepapers/corba/.

[34] J. S. Kinnebrew, A. Gupta, N. Shankaran, G. Biswas, and D. C. Schmidt. A decision-theoretic planner with dynamic component reconfiguration for distributed real-time applications. In *Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems*, pages 461–472, Washington, DC, USA, 2007. IEEE Computer Society.

[35] J. S. Kinnebrew, W. R. Otte, N. Shankaran, G. Biswas, and D. C. Schmidt. Intelligent Resource Management and Dynamic Adaptation in a Distributed Real-time and Embedded Sensor Web System. Technical Report ISIS-08-906, Vanderbilt University, 2008.

[36] J. S. Kinnebrew, W. R. Otte, N. Shankaran, G. Biswas, and D. C. Schmidt. Intelligent resource management and dynamic adaptation in a distributed real-time and embedded sensor web system. In *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '09, pages 135–142, Washington, DC, USA, 2009. IEEE Computer Society.

[37] P. Labone and M. Ghallab. Planning with sharable resource constraints. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, pages 1643–1649, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[38] S. Lacour, C. Perez, and T. Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 284–287, Washington, DC, USA, 2005. IEEE Computer Society.

[39] 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.

[40] T. Lu, E. Turkay, A. Gokhale, and D. C. Schmidt. CoSMIC: An MDA Tool suite for Application Deployment and Configuration. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, Oct. 2003. ACM.

[41] M. Mikic-Rakic and N. Medvidovic. Architecture-level support for software component deployment in resource constrained environments. In *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 31–50, London, UK, 2002. Springer-Verlag.

[42] S. Miksch, Y. Shahar, and P. Johnson. Asbru: A Task-Specific, Intention-Based, and Time-Oriented Language for Representing Skeletal Plans. In *Proceedings of the 7th Workshop on Knowledge Engineering: Methods & Languages (KEML-97)*, pages 9–19, 1997.

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

[44] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[45] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *Proceedings of EuroSys 2010*, pages 237–250, Paris, France, Apr. 2010.

[46] Object Management Group. *Interceptors FTF Final Published Draft*, OMG Document ptc/00-04-05 edition, Apr. 2000.

[47] Object Management Group. *Lightweight CORBA Component Model RFP*, realtime/02-11-27 edition, Nov. 2002.

[48] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, Dec. 2002.

[49] Object Management Group. *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 edition, July 2003.

[50] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.

[51] Object Management Group. *Lightweight CCM FTF Convenience Document*, ptc/04-06-10 edition, June 2004.

[52] Object Management Group. *CORBA Components v4.0*, OMG Document formal/2006-04-01 edition, Apr. 2006.

[53] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.2 edition, Jan. 2007.

[54] Object Management Group. *Quality of Service For CCM Specification*, OMG Document formal/2008-10-02 edition, Oct. 2008.

[55] 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.

[56] 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.

[57] Object Management Group. *DDS for Lightweight CCM Version 1.0 Beta 2*. Object Management Group, OMG Document ptc/2009-10-25 edition, Oct. 2009.

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

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

[60] W. Otte, J. Kinnebrew, D. Schmidt, and G. Biswas. A flexible infrastructure for distributed deployment in adaptive sensor webs. In *Aerospace conference, 2009 IEEE*, pages 1 –12, march 2009.

[61] W. Otte and D. C. Schmidt. Labor-Saving Architcture: an Object-Oriented Framework for Networked Software. In *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. O'Reilly Media, 1st edition, July 2007.

[62] W. R. Otte, A. Gokhale, and 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, pages 21–30, New York, NY, USA, 2011. ACM.

[63] W. R. Otte, A. Gokhale, D. C. Schmidt, and J. Willemsen. Infrastructure for component-based dds application development. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 53–62, New York, NY, USA, 2011. ACM.

[64] 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.

[65] 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)*, Bengaluru, India, Nov. 2010.

[66] 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.

[67] 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.

[68] Rackspace Hosting. Rackspace Cloud Hosting. `rackspacecloud.com`, 2010.

[69] 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.

[70] N. Shankaran, D. C. Schmidt, Y. Chen, X. Koutsoukous, and C. Lu. The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems. In *Proc. of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*

(ISORC 2007), Santorini Island, Greece, May 2007.

[71] B. Srivastava and S. Kambhampati. Scaling up planning by teasing out resource scheduling. In *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*, pages 172–186, London, UK, 2000. Springer-Verlag.

[72] M. Stal, D. C. Schmidt, and W. R. Otte. Efficiently and transparently automating scalable on-demand activation and deactivation of services with the activator pattern. *Software Practice and Experience, Special Issue on Pattern Languages: Addressing Challenges*, 41(10), Oct. 2011.

[73] 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.

[74] 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.

[75] 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.

[76] D. Suri, A. Howell, N. Shankaran, J. Kinnebrew, W. Otte, D. C. Schmidt, and

G. Biswas. Onboard Processing using the Adaptive Network Architecture. In *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, College Park, MD, June 2006.

[77] 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.

[78] N. Wang, K. Balasubramanian, and C. Gill. Towards a real-time corba component model. In *OMG Workshop On Embedded & Real-time Distributed Object Systems*, Washington, D.C., July 2002. Object Management Group.

[79] N. Wang and C. Gill. Improving real-time system configuration via a qos-aware corba component model. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90273.2, Washington, DC, USA, 2004. IEEE Computer Society.

[80] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill. QoS-enabled Middleware. In Q. Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2004.

[81] D. A. Wheeler. Sloccount, a set of tools for counting physical source lines of code, 2009.

[82] 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.

[83] 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.

[84] M. Xiong, J. Parsons, J. Edmondson, H. Nguyen, and D. C. Schmidt. Evaluating Technologies for Tactical Information Management in Net-Centric Systems. In *Proceedings of the Defense Transformation and Net-Centric Systems conference*, Orlando, Florida, Apr. 2007.