CaDAnCE: A Criticality-aware Deployment And Configuration Engine

Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale

Dept. of EECS, Vanderbilt University, Nashville, TN

{dengg,schmidt,gokhale}@dre.vanderbilt.edu

Abstract

Predictable deployment and configuration (D&C) of components in response to dynamic environmental changes or system mode changes is essential for ensuring open distributed real-time and embedded (DRE) system real-time QoS. This paper provides three contributions to research on the predictability of D&C for component-based open DRE systems. First, we describe how the dependency relationships among different components and their criticality levels can cause deployment order inversion of tasks, which impedes deployment predictability. Second, we describe how to minimize D&C latency of mission-critical tasks with a multi-graph dependency tracing and graph recomposition algorithm called CaDAnCE. Third, we empirically evaluate the effectiveness of CaDAnCE on a representative open DRE system case study based on NASA Earth Science Enterprise's Magnetospheric Multi-Scale (MMS) mission system. Our results show that CaDAnCE avoids deployment order inversion while incurring negligible (<1%) performance overhead, thereby significantly improving D&C predictability.

Keywords: Component middleware, Open Distributed Real-time and Embedded systems, Deployment and Configuration.

1. Introduction

Emerging Trends and Challenges. Open DRE system are often large and complex, *e.g.*, a shipboard computing system may consist of thousands of software components that run a wide range of missions, such as ship navigation, ship structural health monitoring, vision-based object tracking and object characterization. To manage the overall complexity of such systems, the missions are often decomposed into many domain-related tasks that can be modeled as operational strings [1]. An operational string is an assembly of software components that capture the partial order and workflow of a set of executing software capabilities for particular domain tasks.

In complex DRE systems, many operational strings must often be deployed and configured dynamically and simultaneously in response to system operational mode changes or environmental changes. Different operational strings can have different *criticality* levels, which are determined by the importance of the operational strings. In the context of D&C, criticality of an operational string means the urgency for its service startup, *i.e.*, the latency of deploying and configuring all components in an operational string is important so the string can serve client requests.

Unfortunately, when dependencies exist among operational strings, *deployment order inversions* may occur during the deployment and configuration (D&C) process. A deployment order inversion occurs when a high-criticality operational string is deployed *after* a low-criticality operational string due to one or more dependencies from the high-criticality operational string to the low-criticality operational string. Deployment order inversions are problematic for open DRE systems since they impede the predictability of the D&C process. Existing D&C frameworks [2, 3] and standards [4], however, only consider dependency relationships between operational strings to determine service deployment order while ignoring their criticality levels, which can cause deployment order inversions that adversely affect open DRE system QoS.

Solution Approach → Criticality-aware Deployment and Configuration Engine (CaDAnCE).

To address the D&C challenge described above, we describe a D&C framework called the *Criticality-aware Deployment and Configuration Engine* (CaDAnCE), which enhances our prior work on the DAnCE [5] open-source D&C framework implementation of the OMG Deployment and Configuration of component-based distributed applications specification [4]. CaDAnCE uses a multi-graph based algorithm to analyze the dependency relationships between operational strings and removes all the dependencies from higher-criticality operational strings to lower-criticality ones by promoting[1] components from lower-criticality operational strings to higher-criticality ones. By applying the operational string recomposition algorithm, a D&C framework like CaDAnCE can avoid deployment order inversions when multiple operational

---

1 In the context of this paper, *promoting* a component means that before this component is deployed it is temporarily moved from a lower-criticality operational string to a higher-criticality operational string for deployment purposes only.

strings with different criticality levels have complex dependencies on each other.

By analyzing dependency relationships among operational strings and temporarily promoting components across them, CaDAnCE provides two key benefits: (1) deployment order inversion of operational strings can be avoided, which improves the predictability of D&C, and (2) both the functional behavior and QoS behavior of the component-based DRE system are preserved. The novelty of CaDAnCE also stems from its transparency to system deployers, *i.e.*, no additional input is required from system deployers besides standard deployment descriptors.

2. The Design of CaDAnCE

To avoid deployment order inversion, CaDAnCE uses a multi-graph operational string recomposition algorithm. This algorithm is an integral part of CaDAnCE's `ExecutionManager` object in the OMG D&C specification, which runs as a daemon and manages the operational string deployment workflow. An `ExecutionManager` also manages `NodeManagers` in a DRE system execution environment, which run as daemon processes and handle the deployment of components to applications residing on each node. This section gives an overview of CaDAnCE and then describes the operational string recomposition algorithm used in CaDAnCE.

2.1. Overview of CaDAnCE

CaDAnCE converts a set of operational strings into a set of graphs, one graph for each operational string. Each vertex and edge of the graph represents a component and a connection/dependency between two components, respectively. The operational string recomposition algorithm in CaDAnCE *promotes* components from one operational string to another based on two factors: (1) the criticality level of the operational string and (2) the dependency relationship between operational strings. After graphs for all the operational strings are recomposed to account for the component promotion, a new set of operational strings are populated from these recomposed graphs to avoid deployment order inversion.

Figure 1 outlines the operational string recomposition algorithm in CaDAnCE via an example with three operational strings having criticality levels: high, medium, and low. The dotted arrows in the figure represent criticality-inverted dependencies, *i.e.*, dependencies from higher-criticality operational strings to lower-criticality operational strings. Likewise, the solid arrows represent external dependencies that do not cause such inversions.

The algorithm recomposes the graphs by parsing the input set of graphs and removing dotted arrows by promoting some component(s) from a lower-criticality operational string to a higher-criticality string. This process
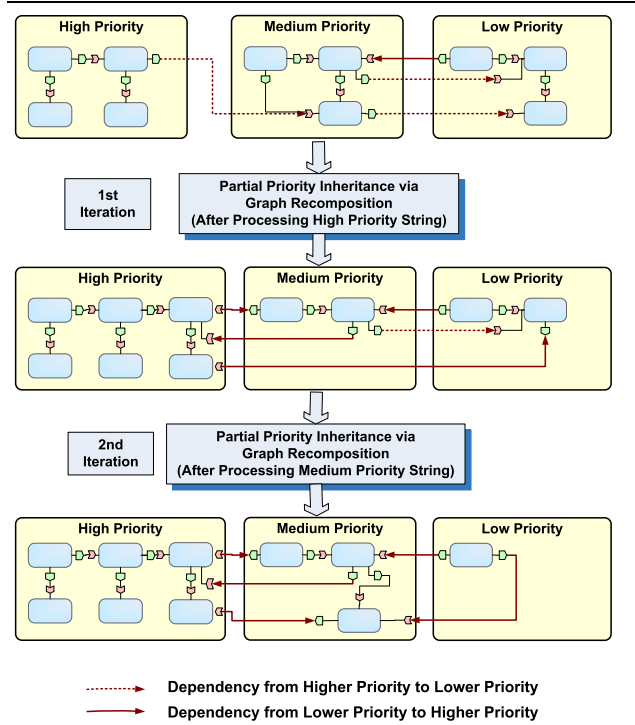

Figure 1: CaDAnCE's Algorithm in Action

may introduce some new dependencies between operational strings due to component promotion. The algorithm only introduces solid arrows, however, *i.e.*, only dependencies from lower-criticality operational strings to higher-criticality strings exist after the recomposition.

When the algorithm finishes, all dotted arrows in the graphs will be removed and there will be no dependencies from higher-criticality operational strings to lower-criticality operational strings. As a result, the deployment order inversion is avoided.

2.2. Characteristics of CaDAnCE

CaDAnCE recomposes operational strings by promoting components, as described above. The following characteristics make it particularly applicable for open DRE systems:

1. CaDAnCE does not affect the functional behavior of component-based DRE systems. CaDAnCE recomposes operational strings to avoid deployment criticality inversion. When components are promoted from one operational string to another, however, no connections to/from these components are changed and no component port interfaces are modified. The topology of all components in all operational strings thus remains the same, *i.e.*, CaDAnCE does not affect the functionality of operational strings because the topology of the strings (including all components and connections) that provide the system's functional be-

havior remains unchanged. Moreover, the operational string recomposition effect by CaDAnCE is visible only within the CaDAnCE D&C framework itself at deployment time, but is transparent to system deployers.

2. CaDAnCE does not affect the QoS behavior of operational strings. When components are promoted from a lower-criticality operational string to a higher-criticality one, component criticality is also increased to match the criticality of the higher-criticality string, which is essential to avoid criticality inversion at deployment-time [6]. Since CaDAnCE promotes components only at deployment time, it does not change the actual real-time QoS configurations (such as thread priorities, component placement and collocation) used later during runtime, *i.e.*, CaDAnCE does not affect the QoS behavior of operational strings.

2.3. Design of the CaDAnCE Operational String Recomposition Algorithm

The goal of CaDAnCE is to remove *all* dependencies from higher-criticality operational strings to lower-criticality strings by promoting components. To avoid the overhead of promoting the same components multiple times, the operational string recomposition algorithm in CaDAnCE processes each operational string in decreasing criticality order. The algorithm thus starts with an operational string having the highest criticality value and processes all its external dependencies by finding all its *reachable* components. After all external dependencies from the highest-criticality operational string are removed, the algorithm then processes the operational string with the next highest criticality, etc.

We define a *dependency trace* as a total ordered sequence where an element is a component of an operational string that has a criticality value associated with it. The starting element of the sequence is the source component of the external dependency of interest. CaDAnCE operational string recomposition algorithm analyzes all the dependency traces in the operational strings and recomposes the operational strings based on dependency trace characteristics.

A dependency trace that spans across multiple operational strings can be further decomposed into the following two situations:

• **Ordered dependency trace.** In an ordered dependency trace, the criticality levels of the elements in the sequence appear in decreasing order, *i.e.*, all external dependencies in the sequence are criticality-inverted. As a result, all criticality-inverted external dependencies must be removed through component promotion. Figure 2 shows an ordered dependency trace.

• **Unordered dependency trace.** In an unordered dependency trace criticality levels of elements in the dependency trace have no order. Figure 3 shows an unordered dependency trace where some external dependencies are
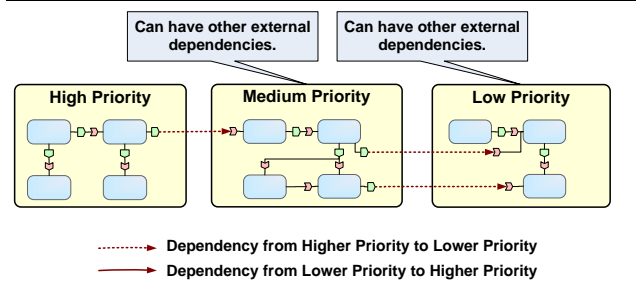

Figure 2: An Ordered Dependency Trace

criticality-inverted (shown as dotted lines) and others are not (shown as solid lines).
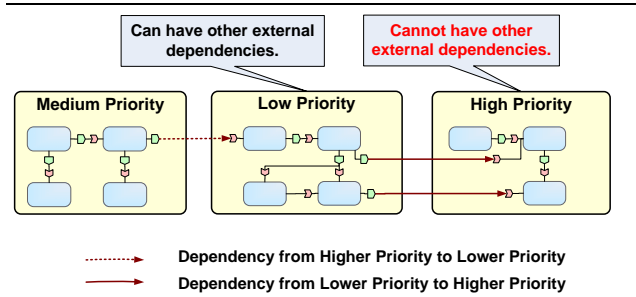

Figure 3: An Unordered Dependency Trace

2.4. Analysis of the Algorithm

The time complexity of the algorithm is nearly linear to the input of the sum of the total number of components and total number of connections, which make CaDAnCE well-suited for deploying operational strings even at run-time.[2] To show how CaDAnCE can deploy operational strings dynamically in an open DRE system, we now analyze two effects of the algorithm on the predictability of operational string deployment:

• **Operational string growth effect.** This effect measures the cost of promoting components from lower-criticality operational strings to higher-criticality strings. The deployment of each component takes time and consumes resources. The fewer components promoted, therefore, the more benefits the algorithm can provide since criticality-inverted dependencies can be satisfied without deploying many components in lower-criticality strings.

In the worst case, *all* components from lower-criticality operational strings could be promoted to higher-criticality

---

[2] Due to space limitations, we do not include the complete analysis of the algorithm in this paper (see [7] for more details).

operational strings, which essentially merges different operational strings together. To handle such a situation, CaDAnCE applies an optimization technique that uses the original deployment descriptors to deploy operational strings to avoid D&C latency increase of the lower-criticality operational string. In production DRE systems, such worst cases happen rarely, *i.e.*, all the components in all operational strings have only one dependency trace.

- **Component host distribution effect.** This effect arises due to component promotion, whereby components that can be deployed by contacting the node once now contact the same node multiple times during deployment. As a result, the overall deployment time grows due to the increasing number of round trip delays. One way to alleviate this problem is to leverage parallel processing among different nodes by using asynchronous techniques, such as CORBA's Asynchronous Method Invocation (AMI) messaging mechanisms [8], between the `ExecutionManager` and `NodeManagers`. For example, AMI can coordinate the parallel deployment of operational strings to `NodeManagers` in a domain, thereby alleviating component host distribution effects.

Section 3 shows empirical results of how CaDAnCE alleviates the two effects described above.

3. Empirical Results

To evaluate the benefits of the CaDAnCE D&C framework, we applied it to a representative open DRE system prototype of the NASA MMS mission system [9]. This section summarizes our hardware/software testbed and then presents the results of using it to evaluate the effectiveness of CaDAnCE empirically.

3.1. Experiment Testbed

Our experiments were conducted in the ISISlab testbed (`www.dre.vanderbilt.edu/ISISlab`), using up to 6 nodes configured with Linux FC4 patched with Ingo Molnar's real-time kernel enhancements. Up to 64 operational strings and 960 components in MMS prototype were deployed and configured using CaDAnCE D&C framework. Each experiment compared CaDAnCE with our baseline D&C framework DAnCE, which is a standards-compliant D&C framework based on OMG D&C specification [4].

3.2. D&C Latency vs. Criticality

Hypothesis. The hypothesis of this experiment is that CaDAnCE can avoid deployment order inversion when deploying multiple operational strings, where higher-criticality operational strings have dependencies on lower-criticality operational strings.

Experimental design. We conducted two experiments with different configurations of operational string dependencies. The first experiment consisted of 3 operational strings with criticality level high, medium, and low, respectively. Each operational string had 15 components evenly distributed across 5 nodes, *i.e.*, each node had 9 components. The high-criticality operational string had one dependency on the medium-criticality operational string, which in turn had one dependency on the low-criticality operational string. The dependency between these operational strings had a low growth rate, *i.e.*, only 1 component in a lower-criticality operational string required promotion.

Our second experiment had two operational strings with criticality level high and low, respectively. The configuration of each operational string is the same as our first experiment. The dependency between these two strings had worst-case operational string growth rate. All components in the lower-criticality operational string thus required promotion to the higher-criticality operational string to avoid deployment order inversion.

Empirical results and analysis. Figures 4 and 5 show the end-to-end D&C latency for each operational string in the two experiments described above. As shown in Fig-
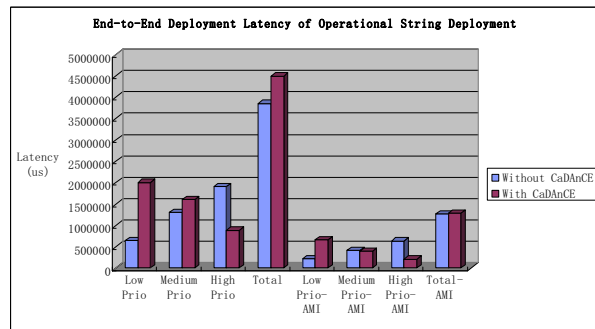

Figure 4: D&C Latency Changes by CaDAnCE

ure 4, without using CaDAnCE, the high-criticality operational string yielded the highest D&C latency, whereas the low-criticality operational string yielded the lowest D&C latency. The medium-criticality operational string lies in between the other two. This result occurs due to the dependency relationships among the 3 operational strings. Without using CaDAnCE, the low-criticality operational string must be deployed first among the three, followed by medium-criticality and high-criticality operational strings, respectively.

Figure 4 shows that when using CaDAnCE, the high-criticality operational string incurs the smallest D&C latency among the three. CaDAnCE therefore effectively avoids deployment order inversion. This figure also

shows how the *component host distribution effect* introduced by the operational string recomposition algorithm in CaDAnCE is masked by applying CORBA AMI, as described in Section 2.4.

Applying AMI improves CaDAnCE deployment performance in two ways. First, the D&C latency of *each* operational string is reduced because the `ExecutionManager` coordinates with the `NodeManagers` to parallelize deployment. Second, AMI masks the component host distribution effect, which reduces total D&C latency of all the operational strings, as shown in Figure 4.

Figure 5 shows the D&C latency results of the worst case scenario, where an operational string *merge* occurs. In this
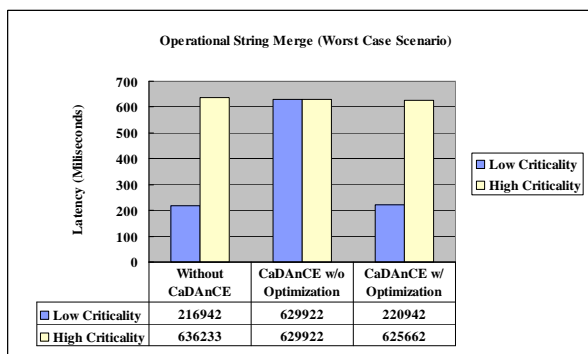

Figure 5: D&C Latency Changes by CaDAnCE

experiment, the dependency between the two operational strings caused all components in the low-criticality operational string to be promoted to the high-criticality string, essentially merging the two operational strings together. As a result, the latency of deploying the high-criticality operational string is nearly the same as our baseline. Without applying the optimization technique described in Section 2.4, the D&C latency of the low-criticality operational string increases without decreasing the D&C latency of the high-criticality operational string. With such an optimization, however, the D&C latency of the low-criticality operational string is the same as our baseline D&C framework. These results validate our hypothesis that CaDAnCE can preserve the D&C latency of low-criticality operational strings, even in the worst-case scenario.

4. Concluding Remarks

The predictability of deployment and configuration (D&C) is essential to support run-time QoS demands of open DRE systems. This paper describes how the CaDAnCE D&C framework can minimize D&C latency of mission-critical operational strings, thereby improving the predictability of D&C. At the heart of CaDAnCE

is a multi-graph dependency tracing and graph recomposition algorithm that promotes components from one operational string to another to ensure D&C predictability. By using information available at deployment time, D&C frameworks can effectively identify the complex dependency relationships among operational strings and perform on-line optimizations, such as the multi-graph algorithm presented in Section 2 of this paper.

The CaDAnCE D&C framework is available in open-source form at `www.dre.vanderbilt.edu/ciao`.

References

[1] 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*, vol. 80, pp. 984–996, July 2007.

[2] M. Desertot, H. Cervantes, and D. Donsez, "FROGi: Fractal Components Deployment over OSGi," in *Software Composition*, pp. 275–290, 2006.

[3] 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*, (Edinburgh, UK), pp. 50–64, May 2004.

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

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

[6] R. Bettati and J. W.-S. Liu, "End-to-end scheduling to meet deadlines in distributed systems," in *International Conference on Distributed Computing Systems*, pp. 452–459, 1992.

[7] G. Deng, D. C. Schmidt, and A. Gokhale, "Ensuring Deployment Predictability of Distributed Real-time and Embedded Systems," Tech. Rep. ISIS-07-814, Vanderbilt University, November 2007.

[8] 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.

[9] 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.