DAnCE: A QoS-enabled Component Deployment and Configuration Engine

Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, and Aniruddha Gokhale

Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville, TN 37203, USA*

Abstract

This paper presents two contributions to the study of component deployment for distributed real-time and embedded (DRE) systems. First, it uses an inventory tracking systems (ITS) as a case study to elicit challenges involved in deploying DRE systems to account for their quality of service requirements. Second, it describes how we designed and implemented the Deployment And Configuration Engine (DAnCE), which is QoS-enabled middleware that addresses the challenges that arose in the context of our ITS case study. Our experience shows that DAnCE provides an effective platform for deploying DRE system components using a standard runtime environment and metadata.

1 Introduction

Component middleware platforms are an effective way of achieving systematic reuse and composition of software artifacts [1]. In these platforms, *components* are units of implementation and composition that collaborate with other components via *ports*, including (1) *facets*, which define interfaces that accept point-to-point method invocations from other components, (2) *receptacles*, which indicate dependencies on point-to-point method interfaces provided by other components, and (3) *event sources/sinks*, which enable the exchange of typed messages with one or more components. Groups of related components can be connected together via their ports to form component *assemblies* that can be deployed to particular nodes in a *target domain*. Implementations of component assemblies are bundled into *packages* that can contain (1) multiple binary executables of the same component written in different languages and for different OS platforms and (2) metadata that describes the package contents.

In large-scale distributed real-time and embedded (DRE) systems, such as shipboard computing environments [2], inventory tracking systems [3], and intelligence, surveillance and reconnaissance systems [4], component middleware features can help make the software more flexible by separating (1) application functionality from (2) system lifecycle activities, such as component configuration and deployment. Conventional component middleware platforms, such as J2EE and .NET, is not well-suited for these types of DRE systems since they do

* This work is supported in part by funding from DARPA, NSF, LMCO ATC, LMCO ATL, LMCO Eagan, Raytheon, and Siemens CT.

not provide real-time quality of service (QoS) support. *QoS-enabled component middleware*, such as CIAO [5], Qedo [6], and PRiSm [7], have been developed to address these limitations by combining the flexibility of component middleware with the predictability of Real-time CORBA.

QoS-enabled component middleware, however, also introduces new complexities that stem from the need to (1) deploy component assemblies into the appropriate DRE system target nodes, (2) activate and deactivate component assemblies automatically, (3) initialize and configure component server resources to enforce end-to-end QoS requirements of component assemblies, and (4) simplify the configuration, deployment, and management of common services used by applications and middleware. The lack of portable, reusable, and standard mechanisms to address these challenges is hindering the adoption of component middleware technologies for DRE systems.

To meet these challenges, we have developed the *Deployment and Configuration Engine* (DAnCE), which is an open-source (`www.dre.vanderbilt.edu/CIAO`) QoS-enabled middleware framework compliant with the OMG Deployment and Configuration specification [8] that enables the deployment of DRE system component assemblies by addressing various QoS-related concerns, such as collocation, memory constraints, and processor loading. The deployment and configuration of components in DAnCE, therefore, involves mapping known variations in the *application requirements space* (such as variations in QoS requirements) to known variations in the *software solution space* (such as configuring the underlying network, OS, middleware, and application parameters to satisfy the end-to-end QoS requirements).

The key capabilities provided by DAnCE to support deployment and configuration of DRE systems include:

- One-time parsing and storing of component configuration and deployment descriptions (which are represented as metadata in XML format) so that runtime parsing overhead is not incurred during component deployment.
- Automatic downloading of component packages so that the implementations can be changed seamlessly as components migrate from one node to another, even in a heterogeneous target domains.
- Automatic configuration of object request brokers (ORBs), containers, and component servers to (1) meet the desired QoS requirements and (2) reduce human operator mistakes introduced while configuring middleware and application components.
- Automatic connection[1] of component ports so that developers need not be concerned with these low-level details.
- Automatic deployment and lifecycle management of common middleware services, such as directory, event, security, and load balancing services, so that developers can concentrate on component business logic, rather than tedious and error-prone programming activities concerned with managing these common services.

[1] In the context of this paper, a *connection* refers to the high-level binding between an object reference and its target component, rather than a lower-level transport (*e.g.*, TCP) connection.

The remainder of this paper is organized as follows: Section 2 provides an overview of inventory tracking system (ITS) case study that elicits many challenges of deploying large-scale DRE systems; Section 3 describes how we designed and applied DAnCE to resolve key challenges in our ITS case study; Section 4 compares our work with related efforts; and Section 5 presents concluding remarks.

2 Deployment and Configuration Challenges in Component-based DRE Systems

To illustrate the deployment and configuration challenges in DRE systems, this section presents a case study of a representative component-based DRE system called the *inventory tracking system* (ITS) [3]. An ITS is a warehouse management infrastructure that monitors and controls the flow of goods and assets within a storage facility. Users of an ITS include couriers (such as UPS, DHL, and Fedex), airport baggage handling systems, and retailers (such as Walmart and Target). This section first provides an overview of the structure/functionality of our ITS case study and then uses the case study to describe configuration and deployment challenges.

2.1 Overview of ITS

An ITS provides mechanisms for managing the storage and movement of goods in a timely and reliable manner. For example, an ITS should enable human operators to maintain the inventory throughout a highly distributed system (which may span organizational boundaries), and track warehouse assets using decentralized operator consoles. In conjunction with colleagues at Siemens [3], we have developed the ITS shown in Figure 1 and deployed it using DAnCE. Figure 1

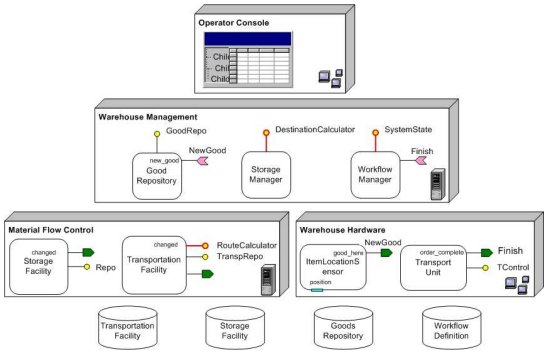

Fig. 1: Key Components in the ITS Case Study

shows how our ITS consists of the following three subsystems:

– **Warehouse management**, whose high-level functionality and decision-making components calculate the destination locations of goods and delegate the remaining details to other ITS subsystems.

– **Material flow control**, which handles all the details (such as route calculation and transportation facility reservation) needed to transport goods to their destinations. The primary task of this subsystem is to execute the high-level decisions calculated by the warehouse management subsystem.
– **Warehouse hardware**, which deals with physical devices (such as sensors) and transportation units (such as conveyor belts, forklifts, and cranes).

After the ITS components comprising the ITS subsystems described above are developed, they must be configured and deployed to meet warehouse operating requirements. In our ITS case study, ∼200 components must be deployed into 26 physical nodes in the warehouse. We focus on a portion of this system to motivate key challenges DAnCE faced when deploying and configuring the ITS. Figure 2 shows a subset of key component interactions in the ITS case study shown in Figure 1. As shown in this figure, the `WorkflowManager` com-

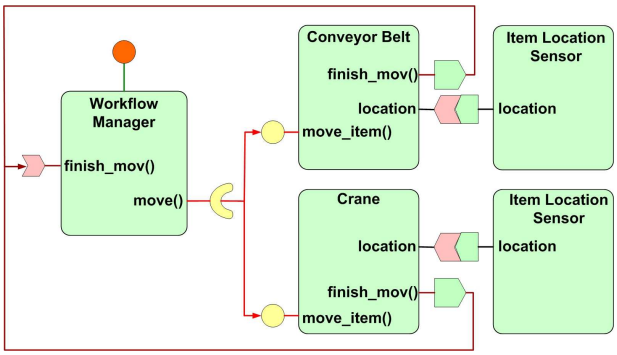

Fig. 2: Component Interactions in the ITS Case Study

ponent of the material flow control subsystem is connected to the conveyor belt and crane transportation units of the warehouse hardware subsystem. We focus on the scenario where the `WorkflowManager` contacts the `ConveyorBelt` and `Crane` components using the `move_item()` operation to move an item from a *source* (such as a loading dock) to a *destination* (such as a warehouse storage location). The `move_item()` operation takes source and destination locations as its input arguments. When the item is moved to its destination successfully, the `ConveyorBelt` and the `Crane` inform the `WorkflowManager` via the `finish_mov()` event operation. `ConveyorBelt` and `Crane` components are also connected to various `ItemLocationSensor` components, which periodically inform the other components of the location of moving items.

2.2 Challenges in Configuring and Deploying ITS

Using the ITS case study described in Section 2.1, we now illustrate the deployment and configuration challenges in component-based DRE systems.

Challenge 1: Efficiently storing and retrieving component implementations. Large-scale DRE systems need capabilities that enable application

developers and deployment runtime tools to (1) upload component implementations to storage sites and/or (2) fetch component implementations from storage sites for installation. These capabilities should allow multiple implementations of a component written in different programming languages and run on different OS platforms. Moreover, it should be possible to pre-stage component implementations to avoid downloading selected implementations from central storage sites during the deployment process.

As shown in Figure 2, it is conceivable that how an ITS `ConveyorBelt` component could have implementations for Linux in Java and Windows in C++, which will require that these implementations be fetched and deployed appropriately on a particular node in a small and bounded amount of time.

Challenge 2: Activation, passivation, and deactivation of component assemblies. To manage shared resources in a DRE system effectively, components in an assembly need to be activated to become functional, passivated when they will not be accessed for an extended period of time, and deactivated when they are no longer needed. A key challenge is to coordinate these operations in a complete assembly, rather than in an individual component or node. For example, components in an assembly that collaborate by sending messages or events must be *preactivated* to configure the necessary environment and resources so that messages are exchanged in the intended fashion. In particular, all collaborating components in an assembly must be preactivated before any component is activated. Similarly, all collaborating components need to be passivated before any component is deactivated so that no component tries to communicate after its recipient has been deactivated.

For instance, when the `ConveyorBelt` component in Figure 2 is being removed, the `WorkflowManager` component must already be passivated since otherwise it could continue to make `move_item()` invocations on the `ConveyorBelt`.

Challenge 3: Configuring NodeApplication component server resources. In large-scale DRE systems, QoS requirements (such as low latency and bounded jitter) are often important considerations during the deployment process since component (re)deployment may occur throughout the lifecycle of a large-scale system. To enforce these QoS requirements, component servers and containers must be configured in accordance with QoS properties, such as those defined in Real-time CORBA [9]. Component deployment and configuration tools must therefore be able to (1) specify the middleware configurations needed to configure components, containers, and component servers and (2) set the QoS policy options provided by the underlying middleware into semantically consistent configurations.

For instance, in the ITS case study (Figure 2), whenever a `ConveyorBelt` component's hardware fails, it should notify the `WorkflowManager` in real-time to minimize/avoid damage. Likewise, ITS `ConveyorBelt` and `Crane` components may need to be collocated with `WorkflowManager` in some assemblies to minimize latency.

Challenge 4: Configuring and deploying common middleware services. Traditional object-oriented middleware (such as CORBA 2.x) provides DRE sys-

tems with access to common middleware services (such as Naming and Trading) through the underlying Object Request Broker (ORB) and Portable Object Adapter (POA). Component-based middleware, such as Lightweight CORBA Component Model (CCM) [10] enables (1) reusability of components by implementing only application logic and (2) easier integration into different applications and runtime contexts. Component deployers thus need to support the integration of common middleware services into component-based applications for which no standard mechanisms yet exist.

For instance, Figure 2 shows how the ITS `ItemLocationSensor` and the `ConveyorBelt` components exchange messages using *event sources/sinks*, which may require the configuration of some middleware publish/subscribe services, such as the CORBA Real-time Notification Service or the Data Distribution Service (DDS).

Section 3.2 describes how DAnCE addresses these challenges for DRE systems and how our solutions have been applied to the ITS case study.

3 The Design of DAnCE

This section describes the *Deployment And Configuration Engine* (DAnCE), which is middleware we developed based on the OMG's Deployment and Configuration (D&C) specification [8]. This specification standardizes many aspects of configuration and deployment for component-based distributed systems, including component configuration, component assembly, component packaging, package configuration, package deployment, and target domain resource management. These aspects are handled via a *data model* and a *runtime model*. The data model can be used to define/generate XML schemas for storing and interchanging metadata that describes component assemblies and their configuration and deployment characteristics. The runtime model defines a set of managers that process the metadata described in the data model during system deployment. This section shows how the design and implementation of DAnCE has been tailored to address the D&C challenges of component-based DRE systems described in Section 2.2.

3.1 The Structure and Functionality of DAnCE

The architecture of the Deployment and Configuration Engine (DAnCE) is shown in Figure 3. This section describes how DAnCE provides a reusable middleware framework for deploying and configuring components in a distributed target environment, using the ITS case study in Section 2.1 to motivate its key capabilities. DAnCE is built atop The ACE ORB (TAO) [11] and CIAO [5], which makes it portable to most hardware and OS platforms in use today.

As shown in Figure 3, an ITS deployer creates XML descriptors that convey application deployment and configuration metadata, using external model driven development (MDD) tools [12]. This metadata is compliant with the data model defined by the OMG D&C specification. To support additional deployment and configuration concerns not addressed by this specification, we enhanced the spec-defined data model by describing additional deployment concerns (such as real-

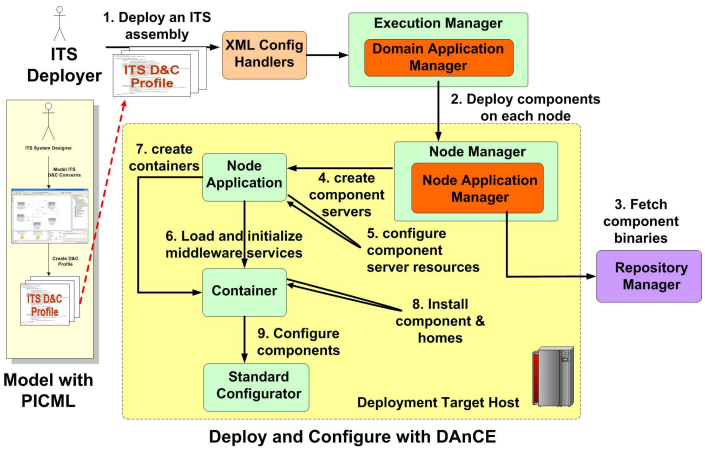

Fig. 3: Overview of DAnCE

time QoS requirements and middleware service configuration and deployment) discussed in Section 3.2.

All the metadata to describe these concerns is captured in an XML file called the *deployment plan*, which describes (1) the DRE system component instances to deploy, (2) what properties of these components should be initialized, (3) what QoS policies these components must contain, (4) what middleware services the components use, and (5) how the components are connected to form component assemblies. The various entities of DAnCE shown in Figure 3 are implemented as CORBA objects[2] that collaborate as follows:

ExecutionManager runs as a daemon and is used to manage the deployment process for one or more domains. In accordance with the D&C specification, DAnCE defines a *domain* as a target environment composed of *nodes*, *interconnects*, *bridges*, and *resources*. An ExecutionManager uses the *factory* and *finder* design patterns to manager a set of `DomainApplicationManagers`.

DomainApplicationManager manages the deployment of components within a single domain (to manage multiple domains, an `ExecutionManager` can coordinate with multiple `DomainApplicationManagers`). A `DomainApplicationManager` splits a deployment plan into multiple subplans, one for each node in a domain. In DAnCE, the `ExecutionManager` and `DomainApplicationManager` objects reside in the same daemon process to improve deployment performance by leveraging the collocation optimizations provided by TAO.

NodeManager runs in a daemon on each node and manages the deployment of all components that reside on that node, irrespective of which application they are associated with. Components are created by containers, which are hosted in component server processes called `NodeApplications`. The `NodeManager` creates

[2] The DAnCE deployment infrastructure is implemented as CORBA objects to avoid the circular dependencies that would ensue if it was implemented as components, which would have to be deployed by DAnCE itself!

the NodeApplicationManager, which in turn creates the NodeApplication processes that host containers, thereby enhancing the reuse of components shared between applications on a node.

NodeApplicationManager is collocated with a NodeManager to manage the deployment of all components within a NodeApplication which is a server process that hosts a group of related components in a particular application. To differentiate deployments in a node, DAnCE's DomainApplicationManager uses the node's NodeManager to create a NodeApplicationManager for each deployment and sends it the metadata it needs to deploy components.

NodeApplication plays the role of a component server process that provisions the computing resources (*e.g.*, CPU, memory and network bandwidth) for the components it hosts. Based on metadata provided by other DAnCE managers in the deployment process, the NodeApplication creates the initial containers that provide an environment for creating and instantiating application components. Components in a node are thus deployed in one or more NodeApplications in accordance with a deployment plan.

RepositoryManager runs as a daemon dedicated to a domain and is used by (1) deployer agents to store component implementations and (2) DAnCE's NodeApplicationManager to fetch necessary component implementations on demand. Each NodeApplicationManager uses its RepositoryManager to search component implementation binaries (stored in the form of dynamic linking libraries) and fetches them into the local node's storage cache.

3.2 Applying DAnCE to Address DRE Systems D&C Challenges

The remainder of this section describes how (1) the DAnCE managers in Figure 3 address key DRE systems D&C challenges described in Section 2.2 and (2) our solutions are applied to the ITS case study presented in Section 2.1.

Resolving Challenge 1: Storing and Retrieving Component Implementations via a Repository Manager. DAnCE's RepositoryManager provides efficient mechanisms where applications can (1) store component implementations at any time during the system lifecycle and (2) retrieve different versions of implementations as components are (re)deployed on various types of nodes. As shown in Figure 4, the RepositoryManager can also act as an HTTP client and download component implementations specified as *URLs* in a deployment plan. It caches these implementations in the local host where the RepositoryManager runs so they can be retrieved by NodeApplicationManagers.

Over a system's lifetime, a component could be migrated and redeployed on a node whose type is different than its earlier host(s), in which case a different component implementation must be provided. To support efficient deployment, DAnCE's NodeApplicationManagers periodically contact the RepositoryManager to download the latest implementations of designated components. When a component is redeployed, therefore, all its implementations can be cached locally on the target nodes, so downloading overhead need not be incurred during the deployment process.

DAnCE's `RepositoryManager` uses ZIP compression and file archiving mechanisms (`debin.org/zzip`) to provide an efficient representation of the contents of a ZIP archive and (de)compress all the implementations in a packaged format. It uses CORBA operation invocations to transfer the ZIP-encoded component assembly packages to the node(s) in a domain that run `NodeApplicationManager`s.

In the ITS case study, an initial deployment might write the `ConveyorBelt`

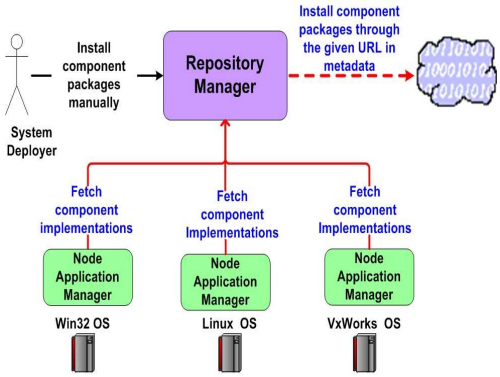

Fig. 4: Downloading implementations using the Repository Manager

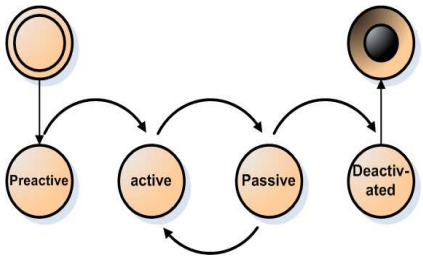

Fig. 5: Different States of a Component

component in Java and host the component on an Embedded Linux node. As the system runs, ITS developers could create a C++-based Win32 implementation of `ConveyorBelt` and submit it to DAnCE's `RepositoryManager`. At some point during the ITS lifecycle, the `ConveyorBelt` could be stopped at the current Linux node and moved to a Windows node. To execute that deployment request, DAnCE's `NodeApplicationManager` running on the Windows node could contact the `RepositoryManager` to retrieve the Windows implementation of the `ConveyorBelt` component and deploy it. The `RepositoryManager` thus helps decouple when and how ITS component implementations are developed from when they are deployed.

Resolving Challenge 2: Using the DomainApplicationManager to Coordinate the Component Assembly Lifecycle. During the lifecycle of the component assembly, DAnCE's `DomainApplicationManager` maintains PREACTIVE, ACTIVE, PASSIVE, and DEACTIVATED runtime state information on each component in the component assembly, as shown in Figure 5. The PREACTIVE state indicates that the component has been created and has been provided its environment settings. The ACTIVE state indicates that the component has been activated with the underlying middleware. The PASSIVE state indicates that the component is idle and all its resources can be used by other components. The DEACTIVATED state means that the component has been deactivated and removed from the system.

During the deployment process, DAnCE's `DomainApplicationManager` ensures that components are not connected and activated until all the components

are in the *preactive* state. Similarly, during assembly deactivation, DAnCE's `DomainApplicationManager` ensures that components in an assembly are deactivated only when all the components are in the *passive* state.

To ensure that a component's ongoing invocations are processed completely before it is passivated, all operation invocations on a component in CIAO are dispatched by the standard Lightweight CCM Portable Object Adapter (POA), which maintains a *dispatching table* that tracks how many requests are being processed by each component in a thread. CIAO uses standard POA reference counting and deactivation mechanisms to keep track of the number of clients making invocations on a component. After a server thread finishes processing the invocation, it decrements the reference count in the dispatching table. Only when the count is zero, is the component passivated. CIAO therefore ensures that the system is always in a consistent state to ensure that no invocations are lost. To prevent new invocations from arriving at the component while it is being passivated, the container blocks new invocations for this component in the server ORB using standard CORBA portable interceptors.

In the ITS case study, DAnCE's `DomainApplicationManager` ensures that the `ItemLocationSensor` components does not make operation invocations on the `ConveyorBelt` components unless both are active. Similarly, during the deactivation of the `ConveyorBelt` component, the `DomainApplicationManager` ensures that `WorkflowManager` components are passivated, which ensures that all `move_item()` requests are handled properly. Finally, the `ConveyorBelt` component's POA ensures that all requests being processed by the component are dispatched before deactivating the component.

Resolving Challenge 3: Configuring NodeApplication Component Server Resources. To enforce QoS requirements, DAnCE extends the OMG D&C [8] specification to define `NodeApplication` server resource configurations, which heavily influence end-to-end QoS behavior. Figure 6 shows the different cate-

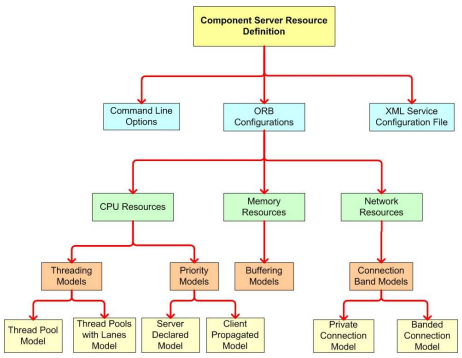

Fig. 6: Specifying RT-QoS requirements

An example server resources specification document:

```
<ServerResourceDef id="ITS_HIGH_PRIORITY_SETTING">
   <ServerCmdlineOptions>
      <!-- No command line options when starting the NodeApplication -->
   </ServerCmdlineOptions>

   <ACESvcConf URI="RTSvc.conf">
      <!-- an external service configruation file -->
   </ACESvcConf>

   <ORBConfigs>
      <resources>
         <threadpool id="high_prio_pool"
                     stacksize="0"
                     static_threads="5"
                     dynamic_threads="0"/>
      </resources>
   </ORBConfigs>
</ServerResourceDef>
```

Fig. 7: Example Server Resources Specification

gories of server configurations that can be specified using the DAnCE *server resources XML schema*, which are related to system end-to-end QoS enforcement.

In particular, each server resources specification can set the following options: (1) *ORB command-line options*, which control TAO's connection management models, protocol selection, and optimized request processing, (2) *ORB service configuration option*, which specify ORB resource factories that control server concurrency and demultiplexing models. Using this XML schema, a system deployer can specify the designated ORB configurations.

As described in Section 3.1, components are hosted in containers created by the *NodeApplication* process, which provides the run-time environment and resources for components to execute and communicate with other components in a component assembly. The ORB configurations defined by the *server resources XML schema* are used to configure *NodeApplication* processes that host components, thereby providing the necessary resources for the components to operate. Since the deployment plan describes the components and the artifacts required to deploy the components, DAnCE extends the standard OMG D&C deployment plan to specify the server resource configuration options.

As shown in Figure 3, `XMLConfigurationHandler` parses the deployment plan and stores the information as IDL data structures that can transfer information between processes efficiently and enables the rest of DAnCE to avoid the runtime overhead of parsing XML files repeatedly. The IDL data structure output of the `XMLConfigurationHandler` is input to the `ExecutionManager`, which propagates the information to the `DomainApplicationManager` and `NodeApplicationManager`. The `NodeApplicationManager` uses the server resource-related options in the deployment plan to customize the containers in the `NodeApplication` it creates. These containers then use other options in the deployment plan to configure TAO's Real-time CORBA support, including thread pool configurations, priority propagation models, and priority-banded connection models.

ITS components, such as `ItemLocationSensor` and `WorkflowManager`, have stringent QoS requirements since they handle real-time item delivery activities. The server resource configurations for all nodes hosting these components are specified via an MDD tool. Figure 7 shows an example XML document that specifies the server resource configurations defined by a system deployer. The `XMLConfigurationHandler` parses the descriptors produced the MDD tool to notify the `NodeApplicationManager`. To honor all the specified configurations, the component servers hosting these components are configured with server-declared priority model with the highest CORBA priority, thread pools with preset static threads, as well as priority-banded connections.

Resolving Challenge 4: Configuring Common Middleware Services During the Deployment Process. To support the integration of common middleware services into component-based applications, DAnCE provides a meta-programmable *service integration framework* shown in Figure 8. This figure shows how DAnCE uses the service integration framework to integrate various middleware services into a DRE system. At the heart of this service integration framework is the DAnCE *Service Configurator*, which is hosted in each `NodeApplication`. Common middleware services (such as the Naming Service,

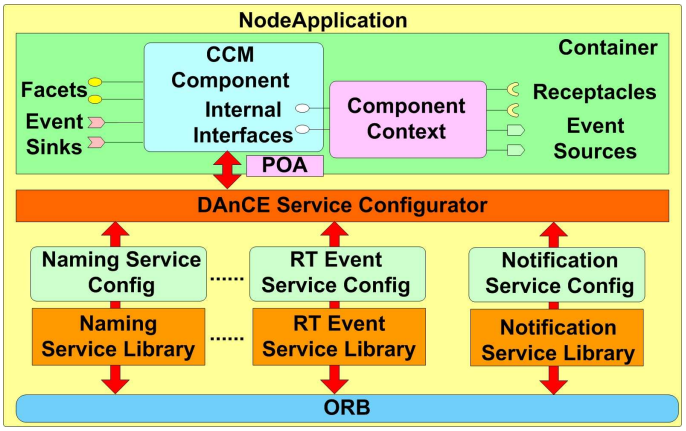

Fig. 8: Configuring Common Middleware Services

Event Service, and TAO Real-time Event Service) are configured using standard CORBA interfaces and hence the usage patterns of such middleware services can be formulated easily. For example, when an application uses TAO's Real-time Event Service, it needs to (1) initialize and configure the QoS properties of the event channel, (2) define the semantic behaviors of event publishers and event consumers, and (3) register the event publishers and event consumers through the event channel interfaces.

To configure and deploy middleware services via DAnCE, CIAO encapsulates these common usage patterns and provides a set of reusable service libraries, one for each type of middleware service, *e.g.*, we designed *CIAO Real-time Event Service library* for the Real-time Event Service provided by TAO. Each library is a wrapper facade of the middleware service provided by the underlying ORB that shields component developers from tedious and error-prone programming tasks associated with initializing and configuring QoS-enabled common middleware services. These libraries also expose interfaces to the DAnCE Service Configurator to manage the services. For example, the *Real-time Event Service Config* shown in the Figure 8 captures the various usage and configuration options (such as event dispatching threading model, event dispatching priority model and event filtering model) for the CIAO Real-time Event Service library. Our DAnCE Service Configurator is designed to support any CORBA service, even those developed to use the earlier CORBA 2.x object model.

During the deployment process, DAnCE uses the deployment plan to express service configuration properties associated with components and assemblies that inform the `NodeApplicationManager` how to initialize the middleware services with desired configuration settings. The `NodeApplicationManager` then conveys to the `NodeApplication` which components to create and which middleware services they require. In response, the `NodeApplication` triggers the DAnCE Service Configurator to load and configure the corresponding CIAO middleware service libraries automatically.

ITS deployment engineers can use MDD tools [13, 14] to model the interactions between the `ItemLocationSensor` and `ConveyorBelt` components and

could indicate whether a direct connection or an event channel is needed for communication. Moreover, stringent QoS requirements such as timing constraints and event delivery latency could also be specified in the communication between the two components. If a direct connection is specified, then at deployment time DAnCE makes the `ItemLocationSensor` component with an *event source* port cache the object reference of the *event sink* port of the `ConveyorBelt` component. After the deployment is complete, these two components can communicate directly through a CORBA remote invocation call. If the DRE system deployer specifies the use of CIAO Real-time Event Service, then DAnCE service configurator and its metadata-based configuration mechanisms configures and manages the service and its QoS settings to provide the desired QoS.

4 Related Work

As component middleware becomes more pervasive, there has been an increase in research on technologies, platforms, and tools for deploying components effectively within distributed systems. This section compares our work on DAnCE with other related efforts.

The OpenCCM (`corbaweb.lifl.fr/OpenCCM/`) Distributed Computing Infrastructure (DCI) federates a set of distributed services to form a unified distributed deployment domain for CCM applications. DCI, however, implements the Packaging and Deployment (P&D) model defined in the original CCM specification, which omits key aspects in the component configuration and deployment cycle, including package repository management, server real-time QoS configuration, and middleware service configuration and deployment. We are currently working with the OpenCCM team to enhance their DCI so that it is compliant with the OMG D&C specification and DAnCE.

[15] proposes using an architecture descriptive language (ADL) that allows assembly-level activation of components and describes assembly hierarchically. Although DAnCE is similar, it uses the XML descriptors synthesized by MDD tools to characterize the metadata regarding components to deploy. Likewise, DAnCE descriptors can specify QoS requirements and/or server resource configurations, so its deployment mechanisms are better suited to deploy applications with desired real-time QoS properties.

[16] proposes the use of the Globus Toolkit to deploy CCM components on a computational grid. Unlike DAnCE, this approach does not provide model-driven development (MDD) tools that enable developers to capture various concerns, such as deployment planning and server configuration, visually. Moreover, DAnCE is targeted at DRE systems with stringent real-time QoS requirements, rather than grid applications, which do not provide real-time support.

Proactive [17] is a distributed programming model for deploying object-oriented grid applications. Proactive defines applications as virtual structures and removes references to the physical machines from the functional code written for the applications. The functional code is mapped to the physical machines using XML descriptors. DAnCE is similar since it also separately describes the target environment using XML descriptors, but it goes further to specify component interdependencies and ensure system consistency at deployment time.

Moreover, Proactive work focuses on deploying Java applications on Java virtual machines, whereas DAnCE implements the OMG D&C specification, which focuses on deploying DRE systems using language- and platform-independent component middleware written in different languages and running on different operating systems.

5 Concluding Remarks

Component middleware is intended to enhance the quality and productivity of software and software developers by elevating the level of abstraction used to develop distributed systems. Conventional middleware, however, generally lacks mechanisms to handle deployment concerns for distributed real-time and embedded (DRE) systems. This paper describes how we addressed these concerns in the Deployment And Configuration Engine (DAnCE), which is an open-source implementation of the OMG's Deployment and Configuration (D&C) specification targeted for deploying and configuring DRE systems based on Lightweight CORBA Component Model (CCM). DAnCE leverages model-driven development (MDD) tools and QoS-enabled component middleware features to support (1) the efficient storage and retrieval of component implementations, (2) component activation, passivation, and removal semantics within component assemblies, (3) configuring QoS-related client/server resources, and (4) integrating common middleware services into applications.

Our future work on DAnCE will focus on (1) integrating reliable multicast mechanisms in TAO to the various `*Manager` objects described in Section 3.1 to improve the scalability and reliability of the deployment process, (2) extending DAnCE to support dynamic component assembly reconfiguration, redeployments, and migrations, (3) enhancing DAnCE to provide state synchronization and component redeployment or recovery support for a fault-tolerant middleware infrastructure, such as MEAD [18], and (4) applying specialization techniques (such as partial evaluation and generative programming) to optimize DRE systems using metadata contained in component assemblies.

References

1. Heineman, G.T., Councill, B.T.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, Reading, Massachusetts (2001)
2. Schmidt, D.C., Schantz, R., Masters, M., Cross, J., Sharp, D., DiPalma, L.: Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. CrossTalk (2001)
3. Nechypurenko, A., Schmidt, D.C., Lu, T., Deng, G., Gokhale, A.: Applying MDA and Component Middleware to Large-scale Distributed Systems: a Case Study. In: Proceedings of the 1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application, Enschede, Netherlands, IST (2004)
4. Sharma, P., Loyall, J., Heineman, G., Schantz, R., Shapiro, R., Duzan, G.: Component-Based Dynamic QoS Adaptations in Distributed Real-Time and Embedded Systems. In: Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04), Agia Napa, Cyprus (2004)

5. Wang, N., Gill, C., Schmidt, D.C., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus (2004)
6. Ritter, T., Born, M., Unterschütz, T., Weis, T.: A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In: Proceedings of the 36^{th} Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003, Honolulu, HW, HICSS (2003)
7. Sharp, D.C., Roll, W.C.: Model-Based Integration of Reusable Component-Based Avionics System. In: Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003. (2003)
8. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document ptc/03-07-08 edn. (2003)
9. Object Management Group: Real-time CORBA Specification. OMG Document formal/02-08-02 edn. (2002)
10. Object Management Group: Light Weight CORBA Component Model Revised Submission. OMG Document realtime/03-05-05 edn. (2003)
11. Schmidt, D.C., Levine, D.L., Mungee, S.: The Design and Performance of Real-Time Object Request Brokers. Computer Communications **21** (1998) 294–324
12. Balasubramanian, K., Krishna, A.S., Turkay, E., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: Applying Model-Driven Development to Distributed Real-time and Embedded Avionics Systems. International Journal of Embedded Systems special issue on Design and Verification of Real-Time Embedded Software (2005)
13. Balasubramanian, K., Balasubramanian, J., Parsons, J., Gokhale, A., Schmidt, D.C.: A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. In: Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Sym., San Francisco, CA (2005)
14. Edwards, G., Deng, G., Schmidt, D.C., Gokhale, A., Natarajan, B.: Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services. In: Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE), Vancouver, CA, ACM (2004)
15. Quema, V., Balter, R., Bellissard, L., Feliot, D., Freyssinet, A., Lacourte, S.: Asynchronous, Hierarchical and Scalable Deployment of Component-Based Applications. In: Proc. of the 2nd International Working Conference on Component Deployment (CD 2004), Edinburgh, UK (2004)
16. Lacour, S., Perez, C., Priol, T.: Deploying CORBA Components on a Computational Grid: General Principles and Early Experiments Using the Globus Toolkit. In: Proc. of the 2nd International Working Conference on Component Deployment (CD 2004), Edinburgh, UK (2004)
17. Baude, F., Caromel, D., Huet, F., Mestre, L., Vayssiere, J.: Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. In: Proc. of the 11th International Symposium on High Performance Distributed Computing (HPDC'02), Edinburgh, UK (2002)
18. Narasimhan, P., Dumitras, T., Paulos, A., Pertet, S., Reverte, C., Slember, J., Srivastava, D.: MEAD: Support for Real-Time Fault-Tolerant CORBA. Concurrency and Computation: Practice and Experience (2005)