Evaluating Real-time Publish/Subscribe Service Integration Approaches in QoS-enabled Component Middleware[*]

Gan Deng, Ming Xiong and Aniruddha Gokhale
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN 37235, USA
{dengg,mxiong,gokhale}@dre.vanderbilt.edu

George Edwards[†]
Department of Computer Science
University of Southern California
Los Angeles, CA 37235
gedwards@usc.edu

Abstract

As quality of service (QoS)-enabled component middleware technologies gain widespread acceptance to build distributed real-time and embedded (DRE) systems, it becomes necessary for these technologies to support real-time publish/subscribe services, which is a key requirement of a large class of DRE systems. To date there have been very limited systematic studies evaluating different approaches to integrating real-time publish/subscribe services in QoS-enabled component middleware. This paper makes three contributions in addressing these key research questions. First, we evaluate the pros and cons of three different design alternatives for integrating publish/subscribe services within QoS-enabled component middleware. Second, we describe how we applied pattern-driven and meta-programming approaches in realizing the most promising choice based on the container programming model. Third, we empirically evaluate the performance of our design and compare it with mature object-oriented real-time publish/subscribe implementations. Our studies reveal that both the performance and scalability of our design and implementation are comparable to its object-oriented counterpart, which provides a key guidance to the suitability of component technologies for DRE systems.

Keywords: Real-time Publish/Subscribe services, Component Middleware, Patterns, Meta-programming.

1 Introduction

An increasing number of distributed real-time and embedded (DRE) systems require middleware support for real-time transfer of control and data among large number of heterogeneous entities that coordinate with each other in a loosely coupled fashion. Examples of such systems include military systems like the Joint Battlespace Infosphere (JBI), telecommunications systems involving large scale network monitoring and management, environmental emergency response systems requiring real-time coordination between various civilian emergency response units, and supervisory control and data acquisition (SCADA) systems requiring real-time robust control and data communication.

Perhaps the most critical middleware service for the types of DRE systems outlined above are asynchronous, event-based publish/subscribe services [1]. The publish/subscribe architecture is a powerful paradigm for event-based communication because it provides *anonymity*, by decoupling the interfaces between event publishers and subscribers; and *asynchronism*, by automatically notifying subscribers when a specified event is generated. These design principles reduce software dependencies and support the loose coupling requirements of these DRE systems.

[*]This work was sponsored in part by grants from NSF ITR CCR-0312859, Siemens, and DARPA/AFRL contract #F33615-03-C-4112
[†]Work done by the author while at Vanderbilt University

Some widely used real-time publish/subscribe services for DRE systems based on *object-oriented* middleware include CORBA Real-time Event Service (RTES) [2], CORBA Real-time Notification Service (RTNS) [3] and OMG's Data Distribution Service (DDS) [4]. For example, the CORBA RTES is based on OMG's Real-time CORBA [5] and provides low-latency/jitter event dispatching, support for periodic processing, dynamic client connection management, centralized event filtering, and efficient use of network and computational resources, which are well suited for DRE systems with stringent QoS requirements.

Recent trends [6, 7] indicate that *QoS-enabled component middleware*, such as CIAO [8], PRiSM [9], and Qedo [10], are increasingly used to develop and deploy next-generation DRE systems. QoS-enabled component middleware inherits the benefits of conventional component middleware (*e.g.*, J2EE, .NET, and CCM) such as (1) standards-based interfaces for component interaction, (2) clear separation of application lifecycle stages, and (3) declarative composition capabilities, while additionally providing separation of QoS provisioning aspects of DRE systems from their functionality aspects, thereby yielding DRE systems that are less brittle and costly to develop, maintain, and evolve [8, 11].

The increasing use of QoS-enabled component middleware in DRE systems compounded by the need for real-time publish/subscribe services to support a large class of DRE systems requires the integration of the real-time publish/subscribe paradigm within QoS-enabled component middleware. Unfortunately standards-based component middleware do not yet specify how publish/subscribe services can be robustly supported within component middleware. Moreover, to date there is a general lack of systematic studies that address these concerns.

This paper systematically evaluates the pros and cons of different design alternatives for integrating real-time publish/-subscribe services within QoS-enabled component middleware architectures. We describe how we applied pattern-driven design and meta-programming techniques in realizing the most promising choice among of these alternatives, which is based on the container programming model. Our study shows that the container-managed real-time publish/subscribe services provide predictable and comparable performance when compared to their object-oriented counterparts, which provides key guidance in the suitability of real-time publish/subscribe services in component technologies for DRE systems.

Paper organization. The remainder of this paper is organized as follows: Section 2 evaluates different architectural choices for integrating real-time publish/subscribe services within QoS-enabled component middleware and analyzes the pros and cons of each approach; Section 3 describes the design challenges we faced to realize the most promising approach, and our solution which combines pattern-driven and meta-programming techniques to address them; Section 4 provides the results of our empirical studies that establish the feasibility of our solution approach, and validates its performance; Section 5 compares our work with related research; and Section 6 presents concluding remarks.

2 Architectural Design Choices for Integrating Real-time Publish/Subscribe Services

In this section we describe three different design choices for integrating real-time publish/subscribe services within QoS-enabled component middleware. To make our discussions concrete, we describe these choices in the context of OMG's

Lightweight CORBA Component Model (LwCCM) [12], which is an emerging component middleware standard for DRE systems and implemented by our CIAO QoS-enabled component middleware. It is worth noting that many of our discussions here are also applicable to other component models as well.

2.1 Overview of Lightweight CORBA Component Model

The OMG Lightweight CCM (LwCCM) specification provides a subset of features of normal CORBA Component Model (CCM) standard [13]. By standardizing only a subset of features, LwCCM is intended to be used for DRE systems with more stringent QoS requirements. Similar to CCM, LwCCM standardizes the development, packaging, configuration, and deployment of component-based DRE systems. As illustrated in Figure 1, LwCCM uses the CORBA distributed object computing (DOC) model as its underlying architecture so applications are not tied to any particular language or platform for their implementations. Throughout this paper we use both the "LwCCM" and "CCM" terms interchangeably without loosing the context.

Components. *Components* in LwCCM are the implementation entities that export a set of interfaces usable by conventional CORBA clients as well as other components. Components can also express their intent to collaborate with other components by defining *ports*, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed events with one or more components.

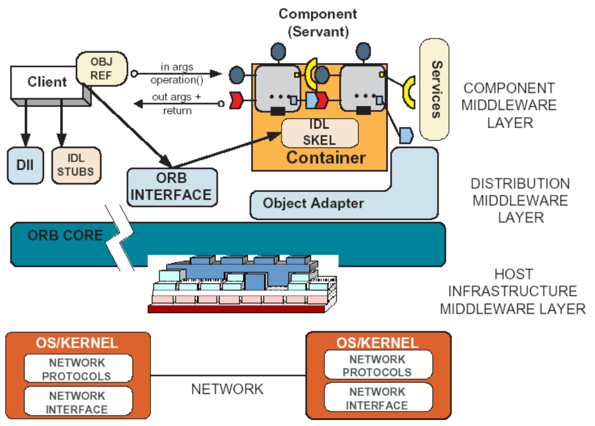

Figure 1. Layered LwCCM Architecture

Container. A *container* in LwCCM provides a runtime environment for one or more components, and manages various pre-defined hooks and strategies, such as persistence, notification, transaction, and security that are used by the components. Each container is responsible for (1) initializing instances of the component types it manages, (2) providing a runtime execution environment called *context* to the components, and (3) managing the policies and lifecycle of the components. Configuration expressed as XML descriptors can be used by component deployment mechanisms to control the lifetime of these containers and the components they manage.

Component server. A *component server* is an abstraction that is responsible for aggregating *physical* entities (*i.e.*, implementations of component instances) into *logical* entities (*i.e.*, distributed application services and subsystems). A component server plays the role of a factory to create containers and standardizes the role of a server process in the CORBA object model.

Component assembly. Assemblies of components in LwCCM are deployed and configured via the OMG Deployment and

Configuration (D&C) [14] specification, which manages the deployment of DRE systems onto different nodes in a target environment. A standards-based deployment framework parses the deployment metadata expressed as XML descriptors, extracts information on the binding of components to target nodes, and deploys the system by instantiating component servers, installing containers and components, and setting up the object-level connections.

Component implementation and packaging. In addition to the runtime building blocks mentioned above, LwCCM also standardizes component implementation and packaging. The Component Implementation Framework (CIF) helps generate the component implementation skeletons automatically using the Component Implementation Definition Language (CIDL) compiler. The CIF consists of patterns, languages and tools that simplify and automate the development of component implementations which are called **executors**. Packaging involves grouping the implementation of component functionality – typically stored in a dynamic link library (DLL) – together with other metadata that describes properties of this particular implementation.

2.2 Evaluating Publish/Subscribe Service Integration Design Choices

Before we delve into describing the design choices, we first provide an intuitive description of how CCM components in a QoS-enabled component middleware use the publish/subscribe paradigm for event communication. Figure 2 illustrates how CCM components can publish and subscribe events through real-time event channels. As illustrated in the figure, QoS configurations for the event dispatching are available at three different scopes, i.e., *channel scope*, *port scope*, and *event scope*, which should ideally be configured and integrated into the component middleware architecture in an intuitive manner. The goal of this paper is to evaluate different design choices of integrating real-time publish/subscribe mechanisms within QoS-enabled component middleware.

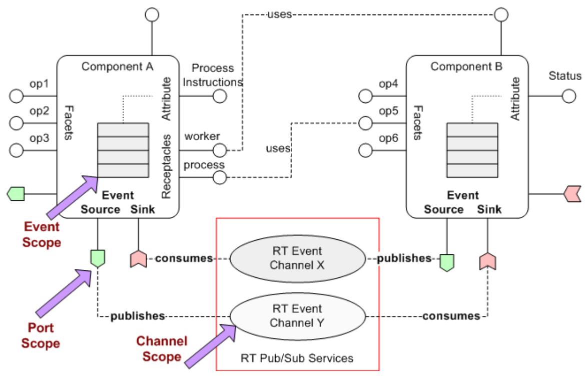

Figure 2. **Using Publish/Subscribe Services in QoS-enabled Component Middleware**

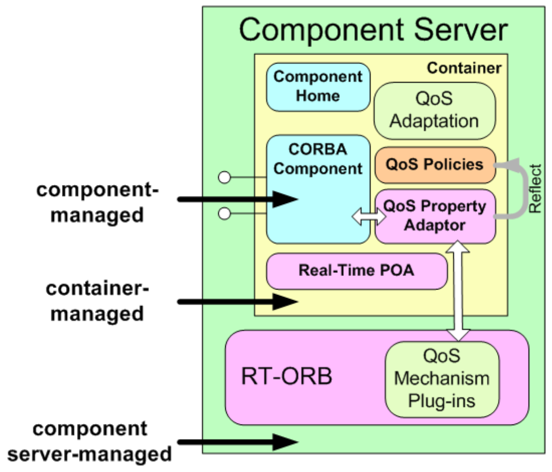

Figure 3. **Architectural Choices for Integrating Publish/Subscribe Services in CCM**

Figure 3 illustrates how these architectural choices vary based on where within the component middleware architecture does a publish/subscribe service gets integrated. The three architectural choices include (1) *component-managed* – where the event channel can be represented as an application-level component, (2) *container-managed* – where the event channel can be encapsulated within the container, and (3) *component server-managed* – where the publish/subscribe can reside within the component server. This section describes each architecture choice in detail and analyzes the advantages and disadvantages of each approach.

2.2.1 Component-Managed Publish/Subscribe Services

Design: The first architecture choice for providing real-time publish/subscribe services in component middleware is to instantiate them as application-level CCM components as illustrated in Figure 4. In this architecture, the interfaces provided by the publish/subscribe services are exposed as component facet ports. These ports contain methods to connect component event sources/sinks to the event channel, configure event service real-time properties, and push events.

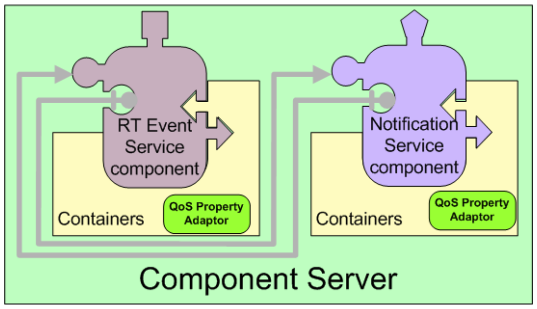

Figure 4. Publish/Subscribe Service as CCM Component

Analysis: The primary advantage of this approach is its simplicity. The complexity needed to implement a publish/subscribe service component is rudimentary since the encapsulated service already implements the publish/subscribe service functionalities, making the full set of service features readily available to other components. Instantiating and deploying multiple publish/subscribe service components follows the same rules that apply to standard components.

However, there are a number of disadvantages of using component-managed publish/subscribe mechanisms. Generally speaking, the shortcomings of the *object-oriented* model still manifest in this architecture. First, the component glue-code, or servant, must manipulate publish/subscribe interfaces directly, which exposes low-level CORBA programmatic details thereby defeating the declarative approaches used by component middleware. Second, the component servant logic must encapsulate QoS and real-time properties, which inhibits the flexibility and reusability of components across different operating contexts and environments. Third, it is impossible to substitute or interchange different real-time publish/subscribe services without recompilation of components because the servant implementation within a component is tightly coupled with a specific type of publish/subscribe service. Finally, application level components must now be responsible for managing the publish/subscribe lifecycles.

In conclusion, this architecture tightly couples the service provisioning behaviors into the component implementation thereby hampering the reusability and evolution of DRE systems. Additionally, the component-based publish/subscribe architecture conflicts with the standard CCM container programming model, which makes the container a mediator between application-level components and common middleware services. Ironically, in this case the publish/subscribe service itself is encapsulated within the component. Finally, this architecture results in a remote call to transmit an event to the publish/-

subscribe service component, which must be handled by the ORB and hence requires additional processing and levels of indirection. Given the number of unfavorable consequences of utilizing this architecture, it is not appropriate for the majority of component-based DRE systems, especially large-scale systems that require highly flexible and customizable QoS guarantees at a low cost.

2.2.2 Container-Managed Publish/Subscribe Services

Design: Figure 5 depicts a second architecture for providing real-time publish/subscribe services in component middleware where a publish/subscribe service is encapsulated within the CCM container. In this architecture, the container is responsible for managing publish/subscribe service lifecycles and their clients, initializing channels and gateways, connecting publishers and subscribers, configuring QoS and real-time properties, managing publisher and subscriber component servants, and setting up the federation among multiple event channels across different containers. In this design, the container exposes two distinct interfaces. One interface provides configuration methods and is invoked by the component deployment framework based on the properties specified in XML-based metadata descriptors that describe configuration decisions. The second interface provides a push method and is invoked by application-level components.

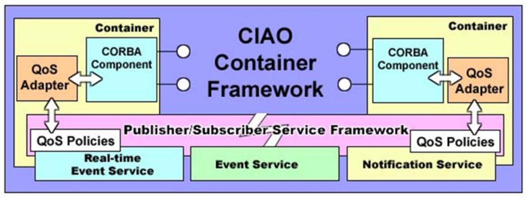

Figure 5. Publish/Subscribe Services Within Container

Analysis: There are many advantages in this architecture. First, since the publish/subscribe services are managed by the container, the business logic of application components are decoupled from the publish/subscribe service configuration. This decoupling enables real-time publish/subscribe service configurations and specifications to be validated and synthesized via high-level model driven engineering (MDE) tools [15] prior to system deployment time, which increases the level of abstraction and automation of the DRE system development process. This separation of concerns maximizes the flexibility and reusability of components by allowing them to be reconfigured with different QoS properties and/or services as required by new and changing operating contexts without making any changes to the application component logic or glue-code thereby obviating the need for recompilation.

Second, this design reduces the memory footprint of individual components and preserves their lightweight nature. Although the component deployment framework is exposed to the implementation details of the real-time publish/subscribe services (since the deployment framework must instantiate and configure the channels) rather than component servant glue code, it is not important for the deployment framework to be as lightweight as the CCM components because the deployment framework is not part of the runtime system and does not consume resources after a DRE system is deployed.

Third, this architecture aligns with the CCM container programming model and defers publish/subscribe configuration-related decisions until deployment time, which allows additional optimizations to be incorporated depending on knowledge of the deployment context. For example, it may not be known until deployment time which network links have high latency or low reliability, yet this information is critical to determining the best possible real-time publish/subscribe service configuration.

The disadvantage to the container-managed event channel architecture is the difficulty encountered in actually implementing it effectively and efficiently due to the complexity of CCM container architecture and its programming model. There are a number of design challenges that arise when pursuing this design choice. We discuss in Section 3 how we resolve these design challenges based on our patterns-driven solutions.

2.2.3 Component Server-Managed Publish/Subscribe Services

Design: The third alternative architecture for providing real-time publish/subscribe services in component middleware is to host them within the component server, which is similar to existing approaches of supporting services in object-oriented middleware. In this architecture, publish/subscribe services are still accessed and manipulated via the container. However, the component server-managed architecture is fundamentally different from the container-managed architecture in that the component server is a lower-level entity which hosts all the components, which in turn end up sharing the same publish/-subscribe service.

Analysis: The advantages present in the container-managed architecture are also applicable to the component server-managed architecture: components are still isolated from publish/subscribe services in such a way that they remain configurable after compilation, and push operations result in only local method invocations. However, the component server-managed architecture is more coarse-grained *i.e.*, a large number of components may be required to share a single service thereby affecting differentiated treatment to application components depending on their real-time needs.

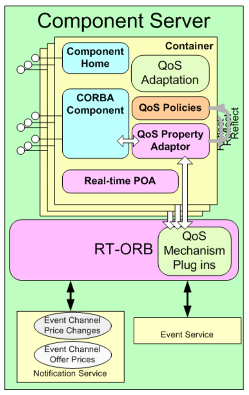

Figure 6. Publish/Subscribe Services Within Component Server

For applications that require either multiple publish/subscribe services on a single host or those who wish to maximize component flexibility to allow for future enhancements or modifications, the component server-managed architecture may be too restrictive. On the other hand, for applications that do not require these capabilities, the component server-managed architecture results in a simpler configuration and deployment process, which reduces development effort. In the case of very large-scale DRE systems, the savings may be substantial if sharing is desired, however, for DRE systems that require partitioning and configuring the system capabilities based on priorities, load balancing and reliability, this coarse-grained approach is not suitable.

Summary: Based on our analysis of the pros and cons of each design choice, we have selected the *container-managed* architecture as our design choice to obtain additional guidance on its applicability and performance.

Due to the complexity of CCM container architecture, component programming model, and the associated D&C model, there are a number of challenges in the context of this design architecture. In Section 3, we show how this architecture can be implemented in a way that is very efficient, lightweight, and flexible enough to accommodate new services to be plugged in with little modification.

3 Container-based Integration of Real-time Publish/Subscribe Paradigm

This section describes the design and implementation of container-based integration of real-time publish/subscribe services in CIAO, which draws on the combined strength of pattern-driven design [16] and meta-programming techniques [17]. We divide this section into two parts. First, we discuss the integration design goals and our implementation strategies. We follow this by the deployment and configuration issues, which arise due to the declarative as opposed to imperative approaches used for deployment in component middleware.

3.1 Pattern-Driven Integration Strategies

Figure 7 gives an overview of the design of the container-managed real-time publisher/subscribe service architecture as outlined in Section 2. The fundamental goal driven by this design is to increase the efficiency and flexibility of large-scale DRE systems, while preserving the lightweight nature of CCM components and CIAO middleware framework. The numbered bullets in this diagram depicts the flow of control among different entities in CIAO QoS-enabled component middleware architecture [8].

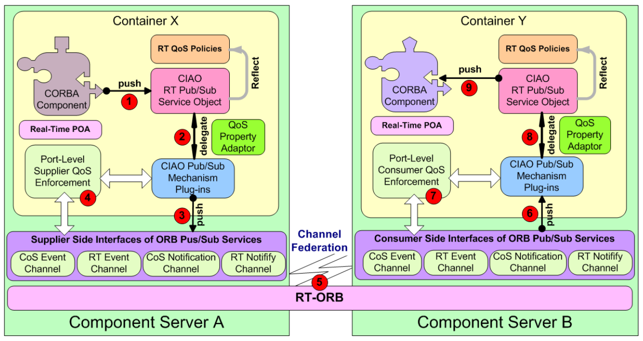

Figure 7. CIAO Publish/Subscribe Architecture

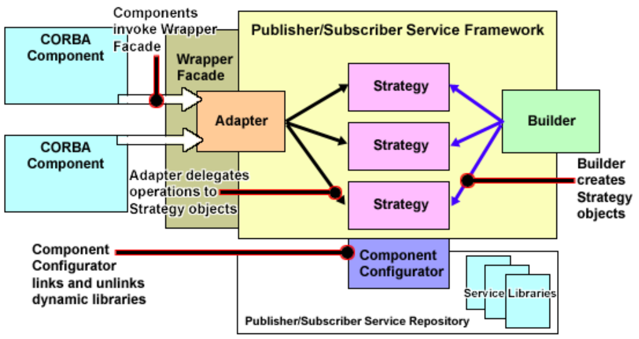
Components invoke Wrapper Facade
CORBA Component
CORBA Component
Wrapper Facade
Publisher/Subscriber Service Framework
Adapter
Strategy
Strategy
Strategy
Builder
Builder creates Strategy objects
Adapter delegates operations to Strategy objects
Component Configurator links and unlinks dynamic libraries
Component Configurator
Service Libraries
Publisher/Subscriber Service Repository

Figure 8. Pattern Interactions in the CIAO Publish/Subscribe Service Framework

For each design goal mandated by the CCM container programming model, our pattern-oriented solution to integrating the real-time publish/subscribe services within the QoS-enabled component middleware is implemented as illustrated in Table 1.

Figure 8 illustrates a pattern language [18] demonstrating the interactions between 5 different patterns in the publish/subscribe service framework we integrated within the CIAO container.

Design Goal 1, which calls for providing a service-independent representation of real-time properties since different publish/subscribe services depend on different representations of real-time properties.	**Solution approach → Adapter pattern:** We apply the *adapter* pattern that converts service-specific representations of real-time properties into service-independent representations. The benefits of this design are twofold: (1) component developers need not concern themselves with peculiar configuration interfaces and (2) no matter what changes occur to the underlying publish/subscribe services, the interface exposed to components does not change.
Design Goal 2, which requires enhancing reuse and extensibility by allowing new publish/subscribe services to be easily plugged-in.	**Solution approach → Strategy pattern:** This design goal is satisfied using the *strategy* pattern, which results in service implementations that are interchangeable from the container perspective. After object creation, the container has no knowledge of the actual algorithm being used, which enables fast operation delegations and simplifies container design.
Design Goal 3, which emphasizes reduction in the memory footprint of the container by decoupling the creation of publish/subscribe service instances from their representation.	**Solution approach → Builder pattern:** The creation of most real-time publisher/subscribe service instances is complex since a lot of objects must be instantiated and configured properly. CIAO container defines a *builder* class that encapsulates the complexity, which results in finer control of the construction process, isolation of construction code, and the ability to vary the service configurations.
Design Goal 4, which requires ensuring that the components incur only the cost of services that are required by deferring publish/subscribe service selection and configuration decisions until *run time* instead of *design time*.	**Solution approach → Component Configurator pattern:** In CIAO, a *component configurator* enables publish/subscribe service libraries to be loaded dynamically on-demand to avoid encumbering the application with unused services, while still allowing components to wait until deployment time to select a particular service. This mechanism provides the flexibility to initiate, suspend, resume, and terminate services.
Design Goal 5, which requires a component be able to access the full set of QoS features available in real-time publish/subscribe services by encapsulating service-specific QoS specification operations within a high-level interface.	**Solution approach → Wrapper Facade pattern:** The CIAO container framework implements a high-level configuration interface based on *wrapper facades* that forwards invocations to the corresponding service-specific operations for each publish/subscribe service. This design results in a concise and robust common programming interface capable of configuring the QoS features in multiple dissimilar publish/subscribe services.

Table 1. Container-based Integration Design Goals and Solutions

3.2 Deployment and Configuration Design Goals and Implementation Strategies

Component-based DRE systems require real-time publish/subscribe services to be deployed and configured (D&C) onto the target execution environment. Common D&C concerns include (1) choices of publish/subscribe services and their bindings to the CCM components (2) process-collocation strategies between CCM components and publish/subscribe services, (3) host-collocation strategy between CCM components and these services, (4) real-time properties on event channels, event ports, and individual events, (5) choices of event channel federation strategies, such as using IIOP based CORBA gateways or UDP based unicast or multicast. To further simply the DRE system D&C tasks, the real-time publish/subscribe service should ideally be automatically deployed and configured within the container and bound to components as an integral part of the standardized D&C process as defined by the OMG Deployment and Configuration (D&C) [19], and even using the same set of D&C tools based on the above standard.

Although the container-based publish/subscribe service integration approach decouples the functional aspect of CCM components from their publish/subscribe QoS requirements, there is a lack of a mechanism to automate and orchestrate the deployment and configuration process of publish/subscribe services. This section describes how we have employed meta-programming techniques to automate the D&C process of real-time publish/subscribe services for DRE systems. Our solution is based on the OMG Deployment and Configuration (D&C) specification and consists of two complementary abstraction models, one called the *data model* based on XML representation and one called the *runtime model* based CORBA 2.x IDL.

Data Model. The data model uses XML descriptors to describe the real-time publish/subscribe service configurations. In order to make our data model compatible with the OMG standard D&C model, we decouple DRE system publish/subscribe service configuration concerns from standards-based component assembly and packaging these concerns into a different set of XML descriptors. These descriptors are based on our proprietary XML schema called *CIAO Events Descriptors*, which defines a rich set of elements called *policies* that capture different D&C concerns of dissimilar real-time publish/subscribe services. Figure 9 shows the policies available for CORBA Real-time Event Service (RTES) that can be specified based on this *data model*.

On the other hand, system functional aspects are captured through a set of standards-based *Component Deployment Plan Descriptors*, which describe the interaction among a set of CCM components. The *Component Deployment Plan Descriptors* can refer to any *CIAO Events Descriptors* and any elements defined in them through two generic, standards-based XML elements called `deployRequirement` and `InfoProperty`. The `InfoProperty` element specifies which *CIAO Events Descriptor* files to use within this deployment plan, and the `deployRequirement` elements can specify which policies to be associated with which entities in the deployment plan, including *components*, *connections* and *ports*. Figure 10 shows an example where we associate a CCM event sink port with a particular event filter, which is defined in a separate XML file.

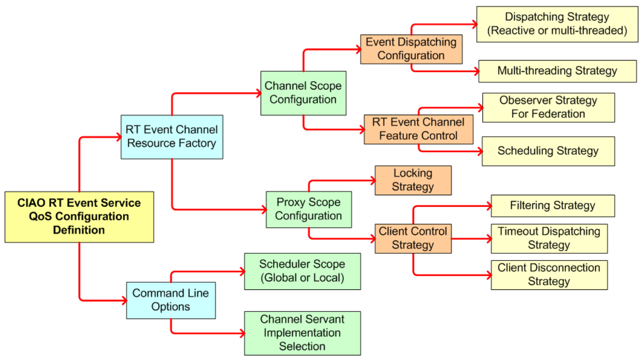
CIAO RT Event Service QoS Configuration Definition
RT Event Channel Resource Factory
Command Line Options
Channel Scope Configuration
Proxy Scope Configuration
Scheduler Scope (Global or Local)
Channel Servant Implementation Selection
Event Dispatching Configuration
RT Event Channel Feature Control
Locking Strategy
Client Control Strategy
Dispatching Strategy (Reactive or multi-threaded)
Multi-threading Strategy
Obeserver Strategy For Federation
Scheduling Strategy
Filtering Strategy
Timeout Dispatching Strategy
Client Disconnection Strategy

Figure 9. RTES QoS Configuration Dimensions in Data Model

This declarative approach offers a much more powerful and flexible reconfiguration mechanism than traditional object-oriented approaches. For example, when some deployment and configuration concerns of publish/subscribe services in an existing DRE system needs to be modified to accommodate a different set of system deployment and configuration requirements, e.g., due to the changes of target infrastructure resources, operating conditions or mission goals, a system deployer can easily modify the *Component Deployment Plan Descriptors* by referring to another set of publish/subscribe configuration policy elements because all the XML-based configuration elements in *CIAO Events Descriptors* can be predefined and reused through the *lazy instantiation* idiom.

```
<connection>
  ...
  <deployRequirement>
    <resourceType>EventFilter</resourceType>
    <name>source_filter_id_01</name>
    <property>
      <name>EventFilter</name>
      <value>
          <type>
            <kind>tk_string</kind>
          </type>
          <value>
            <string>source_filter_id_01</string>
          </value>
      </value>
    </property>
  </deployRequirement>
  ...
</connection>
```

Figure 10. Example QoS Configuration for a CCM Connection

```
  /// Create one CIAOEventService object in
  /// the container, which will be used to mediate
  /// the communication of CCM events
  module Deployment
  {
    /// Extension interface pattern
    interface CIAOContainer : Container
    {
      ...
      readonly attribute
        ::Deployment::Properties properties;

      // installs event service
      CIAO::CIAOEventService install_es (
          in CIAO::EventServiceDeploymentDescription
          es_info)
        raises (InstallationFailure);
      ...
   }
};
```

Figure 11. IDL for CIAO Pub/Sub Service Deployment and Configuration

Runtime Model. To deploy the data model as described above into the target environment, we extend the standards-based *runtime model* of the OMG D&C Specification as a set of CORBA 2.x IDL interfaces to compose real-time publish/subscribe QoS concerns into the system functional concerns. Our extended runtime model applies the *Extension Interface* pattern [20] to make our implementation capable of handling various publish/subscribe service QoS management yet are strictly compatible to the standardized interfaces. Figure 11 shows part of the CORBA 2.x IDL interfaces of the runtime model.

The D&C framework and tools we developed based on this runtime model are integrated as a part of the *Deployment And Configuration Engine* (DAnCE) [21], which is our implementation of OMG D&C specification. As shown in Figure 12, our solution consists of a set of daemon processes plus a utility program called *plan launcher*. The daemon processes include a global-level

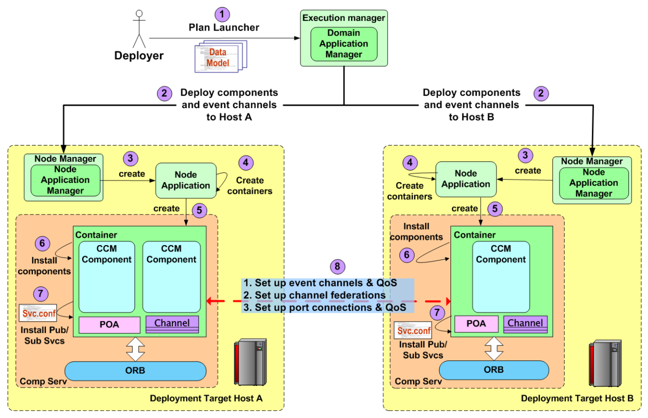

Figure 12. DAnCE: Deployment And Configuration Engine for Pub/Sub Services

daemon process called the `ExecutionManager`, which acts as the central portal for the the deployment and configuration of different DRE systems within a particular domain. Also, there is another type of daemon process called `NodeManager`, which serves the deployment and configuration within an individual node.

All daemon processes can be deployed onto a set of distributed nodes and then cooperatively deploy the publish/subscribe services as an integral part of the standards-based deployment process, all driven by the *plan launcher* utility. The numbered bullets show the flow of control among different DAnCE entities, starting from a system deployer uses the *plan launcher* utility to invoke the service on the `ExecutionManager`, which conforms to the standardized D&C process. The design goals and implementation of the meta-programmable architecture are summarized in Table 2.

Design Goal 1: Eliminate the need for manually writing code to bridge the CCM Eventtype with ORB publish/subscribe service eventtypes.	**Solution approach → Automatic Code Generation:** We enhanced the CIAO CIDL compiler to automatically generate the necessary servant code for each CCM component port, which handles many low-level tedious and error-prone details, such as registering CCM Event valuetype factory with the ORB, marshaling and demarshaling different event types, and converting CCM Eventtype between publish/service typed events.
Design Goal 2: Eliminate the need for manually writing code to identify the event publishers and subscribers which are necessary for certain QoS configurations such as constructing event filters.	**Solution approach → DAnCE Orchestration:** When DAnCE performs deployment, it automatically generates unique identifiers for every event publisher and subscriber, and maps them to the specific event types of the corresponding object-oriented services. This will eliminate the dependencies between the event filters we constructed and the event sources we declared.
Design Goal 3: Eliminate the need for manually writing code to set up event channel federations, which involves tedious and error-prone details such as instantiating gateway objects, activating gateway endpoints, and binding them with event channels.	**Solution approach → Service-based Event Channel Federation** We developed a reusable *Event Channel Federation Service* within the CIAO-container framework to allow different event channel federation mechanisms to be pluggable. For example, UDP Unicast based federation mechanism can easily be replaced with UDP Multicast based federation or CORBA IIOP based federation.

Table 2. Meta-programming Design Goals and Solutions

4 Empirical Performance Evaluation

The success of QoS-enabled component middleware technologies to develop and deploy DRE systems depends on the real-time performance of the publish/subscribe mechanisms supported by them. This section provides empirical results for the container-managed CORBA real-time event service (RTES) integrated within our CIAO QoS-enabled component middleware. We choose RTES as a vehicle to evaluate our design because it is a mature real-time publish/subscribe implementation based on real-time CORBA and has been widely used in many DRE systems [22].

Our performance evaluation of container-managed RTES focuses on answering two key questions: (1) how well does the performance of container-managed RTES in CIAO middleware compare with that of the widely used CORBA 2.x RTES implementation in TAO object-oriented middleware, which is used as a baseline for RTES performance, and (2) how well does the container-managed RTES in CIAO scale with different number of publishers and subscribers under different QoS configurations. We illustrate how the flexible configuration capabilities of CIAO's container-managed RTES can provide desired event delivery QoS without modifications to the component implementations.

4.1 Experimental Testbed

All our benchmarks were conducted on ISISlab (`www.isislab.vanderbilt.edu`), which is a testbed of computers and network switches powered by Emulab software suite that can be arranged in many configurations. ISISlab consists of 6 Cisco 3750G-24TS switches, 1 Cisco 3750G-48TS switch, 4 IBM Blade Centers each consisting of 14 blades (for a total of 56 blades), 4 gigabit network IO modules and 1 management module. Each blade has two 2.8 GHz Xeon CPUs, 1GB of RAM, 40GB HDD, and 4 independent Gbps network interfaces. The underlying hardware used by ISISlab can be configured to provide a virtual network topology and configure various parameters of that network including link bandwidth capacities, node characteristics for use in routing, traffic shaping or traffic generation, and link error rates.

In our tests, we used up to 5 blades. Each blade ran Fedora Core 4 Linux, version 2.6.16-1.2108_FC4smp. All our benchmark applications were run in the Linux real-time scheduling class to minimize extraneous sources of memory, CPU, and network load. For each test, we run the iteration at least 10,000 times.

4.2 Comparing Performance of CIAO's Container-Managed RTES and TAO's RTES

In this test we measure the end-to-end latency introduced between publishers and subscribers. In order to measure the performance of CIAO's container-managed RTES, both the publishers and subscribers are developed as reusable CCM components, which can be deployed by DAnCE for different test cases. On the other hand, for TAO's RTES both the publishers and subscribers are developed in the form of CORBA objects. To ensure an accurate comparison between the CIAO container-managed RTES and TAO's RTES implementations, we designed both tests using the same set of QoS configuration settings on publishers, subscribers and event channels. The subscriber does nothing with the events it receives other than storing the data in a preallocated array. Both tests are configured to measure the end-to-end publish/subscribe latency using the IIOP communication mechanism, which uses point-to-multipoint event delivery rather than IP multicast. We measure the end-to-end latency based on different event payload size as well as increasing number of subscribers.

Latency Results for Process Collocated Event Processing. In this test all the event publishers, subscribers, and the event channel are collocated in the same process, which eliminates effects of ORB remote communication overhead. In the container-managed RTES case, both the publisher and subscriber components are deployed into the same container which in turn is hosted in a single CIAO component server. The end-to-end latency is determined by the publisher sending out the timestamp right before the `push` call and subsequently the subscribers calculating the difference between the timestamp at the publisher side and the subscriber side. We also use variable-sized octet sequence as the payload so that we can easily control the volume of the payload.

One important fact worth mentioning concerns the event data type we used in the TAO's RTES benchmarking test. Since all the event source and event sink ports of CCM components are defined as `Eventtype`, which is a specialized `valuetype`, it is unavoidable to eliminate the additional overhead incurred due to marshaling/demarshaling of such a data type. To ensure a fair comparison between the performance of TAO's RTES and CIAO's container-managed RTES, we send `valuetype` data in both test cases, and make the octet sequence payload as the member of this data type.

Figure 13 shows the latency results for the *point-to-point* (i.e., one-to-one) configuration. In this test, there is only one publisher and one subscriber both of which are collocated within the same event channel in a single OS process. We use variable-sized octet sequence payload so that we can easily control the volume of the payload.

Figure 14 shows the latency results for the *one-to-many* case. In this test, we increase both the number of subscribers and the payload size to see how the end-to-end latency is affected. For the one-to-many case, the measured end-to-end latency is determined by the publisher sending out the timestamp then calculating the difference between the timestamp at the publisher side and the *last* subscriber that receives it.

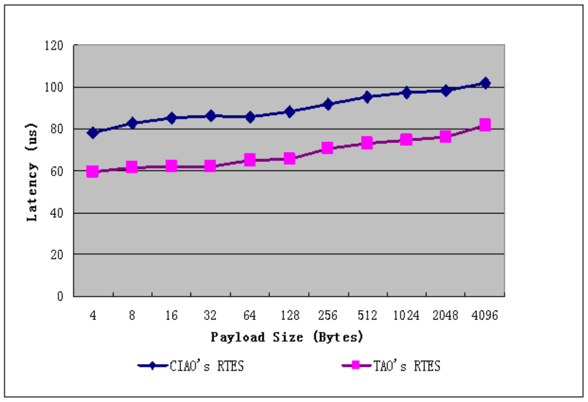

Figure 13. Collocated Point-to-Point Latency

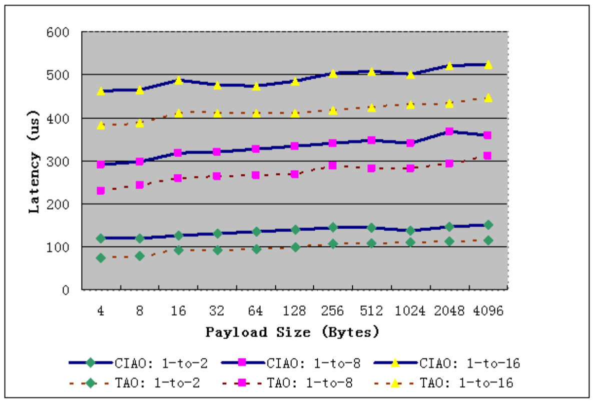

Figure 14. Collocated One-to-Many Latency

Analysis. Our collocation experiment results indicate that the event dispatching overhead in CIAO's container-managed RTES incurs about 20∼25% latency performance overhead consistently over the TAO's RTES implementation. This overhead is primarily due to two reasons: (1) in contrast to programming with CORBA RTES, where the publisher *CORBA objects* and the subscriber *CORBA objects* can directly interact with the event channel as RTES clients, the OMG CCM standard introduces multiple indirections involving *component executors* that must communicate through their individual *contexts*, which in turn interact with the *component servants* and CORBA RTES, and (2) the additional indirection introduced with the higher-level container mediation interface as described in Section 3.

While it is inevitable to avoid the overhead caused by the indirection defined in the OMG CCM standard without breaking standards-based interfaces, we applied process-collocation optimization to CIAO's implementation to improve the performance and predictability of collocated component communication. The process-collocation optimization we conducted improves the performance and predictability for objects that reside in the same address space as the servant implementation, while maintaining locality transparency.

Our process-collocated experiment results demonstrate that the performance optimizations of object-oriented real-time event dispatching are preserved within a container-based solution. It is also interesting to observe that as the number of subscribers increase, the increase in latency is less than linear in both TAO's CORBA RTES implementation and CIAO's container-managed RTES implementation, due in large part to the Handle/Body idiom used to optimize the processing of CORBA `Any` data types in TAO's RTES implementation [23], which is inherited by CIAO RTES. This idiom presents multiple logical copies of the same data while sharing the same physical copy.

Latency Results for Remote Event Processing. [1] In this test, we run the experiment by creating two different processes within a single node to allow the events to be sent remotely. The event publisher and the event channel are collocated in one process, while the subscribers are in the other process. Figure 15 and Figure 16 show the latency results of *point-to-point* and *one-to-many* configurations, respectively.

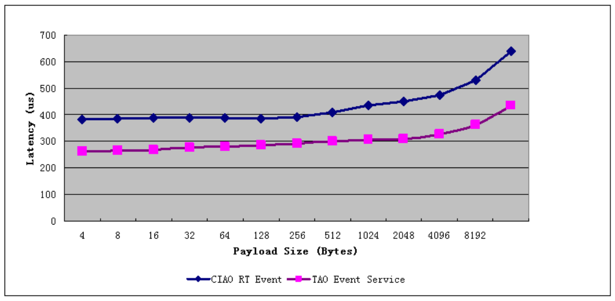

Figure 15. Two Process Point-to-Point Latency

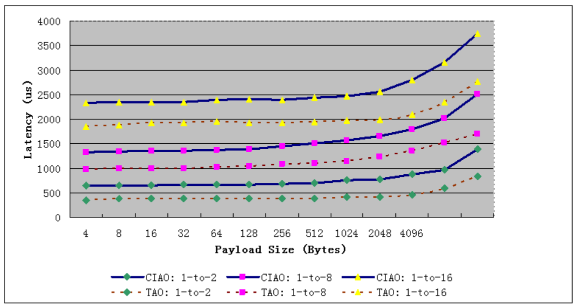

Figure 16. Two Process One-to-Many Latency

Analysis. The results indicate that the remote event processing latency in both CIAO's container-managed RTES and TAO's RTES are much higher than that of process-collocated cases (both incurring about about 6~10 times overhead) due to the process boundary crossing, though they are close to the performance of a remote operation invocation. With increasing number of event subscribers and payload size, the results still show that the event dispatching performance in container-managed RTES in CIAO consistently has about 70~80% performance of TAO's RTES in all test cases. In conjunction with the results of process-collocated tests, these results further confirm that the CIAO container-based RTES solution has predictable performance which is comparable with TAO's RTES, and is thus suitable for DRE systems.

4.3 Evaluating Scalability of CIAO's Container-Managed RTES

Another important characteristic of real-time publish/subscribe services is the scalability. As discussed in Section 3, CIAO's container-managed RTES provides support for both single event channel based event dispatching as well as federated

[1] In this test configuration, a "remote" event is one intended for a subscriber located in the other process.

event channels across multiple containers. To evaluate scalability therefore in this test we measure the throughput of CIAO RTES' event dispatching under different configuration settings. Our goal is to demonstrate the efficiency of remote event processing using federated event channels where there are multiple remote subscribers or multiple distributed nodes. As a result, we split our tests into two parts, one with multiple subscribers on the same node but different processes, and one with multiple subscribers distributed on different nodes, and observe how different configurations can affect scalability.

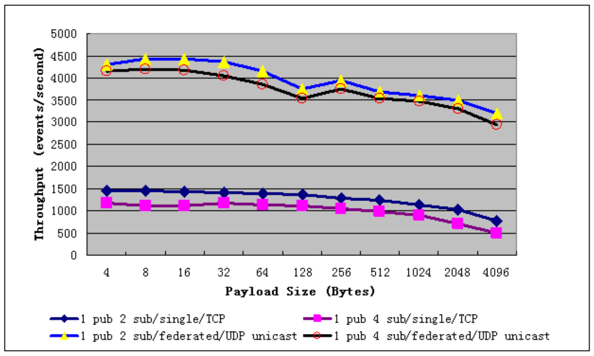

Figure 17. Host Collocated Throughput Measurement: All Subscriber on a Different Component Server

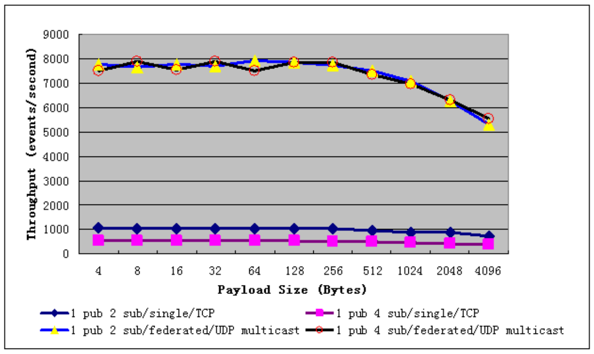

Figure 18. Distributed Throughput Measurement: Each Subscriber on a Different Node

Scalability Results for Host Collocated Remote Event Processing. This test measures the throughput of CIAO's container-managed RTES where all the subscriber components are hosted in a remote container, while the publisher component is hosted in another container. Both containers are hosted in different CIAO component server processes but on the same node. We send a fixed number of events on the publisher side and measure the rate of the number of events dispatched per second. Figure 17 shows the throughput results for this configuration. The bottom two curves show the throughput results when there is no container-managed event channel configured on the subscriber side, and all the subscriber components are directly connected to the real-time event channel hosted in the publisher side container. The upper two curves show the throughput results where both containers have real-time event channel configured locally and they are federated through UDP unicast gateway objects [24].

Analysis. Our results indicate that a federated group of event channels can improve the throughput performance dramatically because publishers and subscribers connect only to their local event channel, while event channel instances talk to each other via the CORBA bus. When multiple subscribers are collocated in the same process, instead of making multiple remote calls (one for each subscriber), only one remote call is necessary from the publisher process to the subscriber process. This configuration improves the total throughput by reducing the average latency for all the subscribers in the system because subscribers and publishers exhibit locality of reference, *i.e.*, most subscribers for any event are in the same domain as the publisher generating the event. We can imagine that in a networked environment, when multiple remote subscribers are interested in the same event only one message is sent to each remote event channel thereby also minimizing wastage of network resources.

Scalability Results for Distributed Event Processing. This test measures the throughput of CIAO's container-managed RTES when the subscriber components are hosted in different containers on different physical nodes, while the publisher component is hosted in another remote node. Again, we send a fixed number of events on the publisher side, and measure the dispatching rate of number of events per second. Figure 18 shows the throughput results for this configuration. The bottom two curves show the throughput results when there is no container-managed event channel configured on the subscriber side, and all the subscriber components are directly connected to the real-time event channel hosted in the publisher's side container through TCP-based IIOP protocol. The upper two curves show the throughput results where all the containers have its own real-time event channel configured locally and they are federated through UDP multicast gateway objects [24].

Analysis. The results indicate that the throughput performance improves drastically thanks to the UDP multicast federation settings among different container-managed event channels distributed across different nodes. Using multicast protocol can avoid duplicate network traffic, and offload event dispatching workload from the CPU to the network infrastructure. The use of multicast is ideal for the scenario where many subscribers are distributed across different nodes in a networked environment since increasing the number of nodes distributed across the network has little effect on UDP multicast. Since currently TAO's RTES only supports unreliable multicast, we are working on a reliable multicast solution within TAO's RTES design so components using CIAO's container-managed RTES can take advantage of this feature.

5 Related Work

This section surveys literature on some available real-time publish/subscribe systems, both standards-based and proprietary, concentrating on the abstraction layers these publish/subscribe mechanisms are based on and the QoS capabilities they support. Some of the prior work provide real-time QoS support for DRE systems, however, their QoS assurance mechanisms are based on the traditional object-oriented middleware layer, which hinders system reusability and maintainability and makes complex DRE systems hard to develop. On the other hand, some others take advantage of the strengths of component middleware, but they are not yet suitable for DRE systems due to the limited real-time QoS support, addressing which is the focus of our work.

5.1 Standards-based Publish/Subscribe Architectures for DRE Systems

• **The OMG Data Distribution Service** The OMG Data Distribution Service (DDS) specification [4]is a standard for QoS-enabled publish/subscribe communication aiming at mission-critical DRE systems. It is designed to provide (1) *location independence* via anonymous publish/subscribe protocols that enable communication between collocated or remote publishers and subscribers, (2) *scalability* by supporting large numbers of topics, data readers, and data writers, and (3) *platform portability and interoperability* via standard interfaces and transport protocols. Multiple implementations of DDS are now available ranging from high-end COTS products to open-source community-supported projects, such as NDDS [25], OpenSplice [26] and TAO DDS [27]. The OMG DDS standard is based on the object-oriented middleware. Currently we are evaluating the performance of various DDS implementations, and plan to integrate an open source implementation into our CIAO component middleware.

• **The CORBA Distributed Notification Service** The OMG has also issued a specification to build distributed versions of the Notification Service via its "Management of Event Domains Specification" [28]. This document describes how multiple instances of the Notification Service can be interconnected to avoid the excessive overhead and eliminate the single point of failure represented by the event channel object. This specification, however, does not incorporate any mechanisms to reduce event delivery based on filters. Similar to DDS, both the CORBA Notification Service and the CORBA Distributed Notification Service are based on object-oriented middleware rather than component middleware, and are a target of our integration approach.

5.2 Proprietary Publish/Subscribe Architectures for DRE Systems

• **Cadena Event Channel Framework** Cadena Event Communication Framework [29] includes a CORBA-based event channel which has been integrated into the OpenCCM component middleware infrastructure. The framework implements a number of features of the event service middleware, such as event filtering and event correlation. Although this work comes close to our work, however, it does not address the problems such as event channel federation, real-time event scheduling and dispatching and periodic event processing, which are crucial for a number of mission-critical real-time applications. Our work, on the other hand, leverages the real-time event channels and QoS-enabled component middleware to provide the properties outlined above.

• **SIENA** SIENA [30, 31] is a notification service architecture for Internet-scale event distribution. The architecture is based on *content-based networking*, where a network of routers propagate packets based not on a specific destination address, but on the contents of the packet. The authors propose using an event format similar to the CORBA Notification Service, *i.e.*, sequence of $(name, value)$ tuples. Using this format, consumers use a boolean predicate on the tuple values to describe the sets of events they are interested in. The authors describe algorithms to reduce the use of network resources.

Both SIENA and TAO's RTES use similar techniques to conserve network and CPU resources, however, SIENA's event and filtering models are more powerful than the model in RTES. In contrast, however, our real-time Event Service is better suited for applications with stringent latency and predictability requirements, such as avionics mission computing.

• **CMU Real-time Publish/Subscribe** Rajkumar, *et al.*, describe a real-time publish/subscribe prototype developed at CMU SEI [32]. Their Publish/Subscribe model is functionally similar to the CORBA RTES, though it defines its own programming APIs and communication protocols. The authors detail how real-time threads and adequate synchronization primitives can be used to implement the RT publish/subscribe model without undue priority inversions. However, the authors do not consider the fact that adequate synchronization primitives are a necessary condition to address unbounded priority inversions, but it is not a sufficient condition.

5.3 Summary

Much has been written about real-time publish/subscribe systems, but little effort has been expended in documenting the patterns, optimizations and architectures required to implement QoS-enabled publish/subscribe models in component-based software architectures. Also, there is very little or no empirical evidence to support the performance and predictability claims of several of these systems when used in component-based systems, even when research concentrates on real-time applications. We have addressed some of these shortcomings in this paper.

6 Concluding Remarks

Component middleware has already received widespread acceptance in the enterprise business and desktop application domains. However, developers of DRE systems have encountered limitations with the available component middleware platforms, such as the CCM and J2EE. In particular, component middleware platforms lack standards-based real-time publish/-subscribe communication mechanisms that support key QoS requirements of DRE systems, such as low latency, bounded jitter, and end-to-end event propagation. This paper provides guidance to establish the feasibility of integrating mature object-oriented real-time publish/subscribe services in QoS-enabled component middleware architectures. In particular, our results indicate that the approach of using container-managed real-time publish/subscribe service not only provides the most flexible way to developing DRE systems by taking advantage of standards-based component models, and the deployment and configuration model, but also provide predictable end-to-end performance and scalability.

The lessons we learned in this study indicate that component middleware standards often introduce some event dispatching overhead compared to the object-oriented middleware. For example, the CCM standard inevitably introduces some performance overheads for event dispatching caused mainly by valuetype based CCM Eventtype marshaling/demarshaling costs and some levels of indirection between different entities including component executors, servants, contexts and containers. It is worth noting that discussion of these overheads is orthogonal to the focus of this paper.

Our ongoing R&D in this field will examine performance optimization opportunities in our CIAO container-managed real-time publish/subscribe service architecture, and further evaluate our approach in the context of a variety of concrete DRE domains with our industry partners, including telecom, avionics mission computing, software defined radio and industrial process control. The software described in this paper is available for download from `www.dre.vanderbilt.edu`, which includes the modeling tools, the D&C tool chain and the middleware.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley & Sons, 1996. 1

[2] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), pp. 184–199, ACM, Oct. 1997. 1

[3] P. Gore, D. C. Schmidt, C. Gill, and I. Pyarali, "The Design and Performance of a Real-time Notification Service," in *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04)*, (Toronto, CA), IEEE, May 2004. 1

[4] Object Management Group, *Data Distribution Service for Real-time Systems Specification*, 1.0 ed., Mar. 2003. 1, 5.1

[5] Object Management Group, *Real-time CORBA Specification*, OMG Document formal/05-01-04 ed., Aug. 2002. 1

[6] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998. 1

[7] J. P. Loyall, R. E. Schantz, D. Corman, J. L. Paunicka, and S. Fernandez, "A Distributed Real-Time Embedded Application for Surveillance, Detection, and Tracking of Time Critical Targets," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 88–97, 2005. 1

[8] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, "Configuring Real-time Aspects in Component Middleware," in *Lecture Notes in Computer Science: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04)*, vol. 3291, (Agia Napa, Cyprus), pp. 1520–1537, Springer-Verlag, Oct. 2004. 1, 3.1

[9] W. Roll, "Towards Model-Based and CCM-Based Applications for Real-time Systems," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, IEEE/IFIP, May 2003. 1

[10] T. Ritter, M. Born, T. Unterschütz, and T. Weis, "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, Jan. 2003. 1

[11] M. Jordan, G. Czajkowski, K. Kouklinski, and G. Skinner, "Extending a J2EE Server with Dynamic and Flexible Resource Management," in *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2004), Toronto, Canada*, pp. 439–458, 2004. 1

[12] Object Management Group, *Lightweight CCM FTF Convenience Document*, ptc/04-06-10 ed., June 2004. 2

[13] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002. 2.1

[14] Object Management Group, *Deployment and Configuration Adopted Submission*, OMG Document mars/03-05-08 ed., July 2003. 2.1

[15] G. Edwards, G. Deng, D. C. Schmidt, A. Gokhale, and B. Natarajan, "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, Oct. 2004. 2.2.2

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995. 3

[17] J. K. Cross and D. C. Schmidt, "Meta-Programming Techniques for Distributed Real-time and Embedded Systems," in *Proceedings of the* 7^{th} *Workshop on Object-oriented Real-time Dependable Systems*, (San Diego, CA), IEEE, Jan. 2002. 3

[18] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*. New York, NY: Oxford University Press, 1977. 3.1

[19] OMG, *Deployment and Configuration of Component-based Distributed Applications, v4.0*, Document formal/2006-04-02 ed., Apr. 2006. 3.2

[20] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000. 3.2

[21] 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*, (Grenoble, France), Nov. 2005. 3.2

[22] C. O'Ryan and D. C. Schmidt, "Applying a Real-time CORBA Event Service to Large-scale Distributed Interactive Simulation," in 5^{th} *International Workshop on Object-oriented Real-time Dependable Systems*, (Monterey, CA), IEEE, Nov. 1999. 4

[23] D. C. Schmidt and C. O'Ryan, "Patterns and Performance of Real-time Publisher/Subscriber Architectures," *Journal of Systems and Software, Special Issue on Software Architecture - Engineering Quality Attributes*, 2002. 4.2

[24] C. O'Ryan, D. C. Schmidt, and J. R. Noseworthy, "Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations," *International Journal of Computer Systems Science and Engineering*, vol. 17, pp. 115–132, Mar. 2002. 4.3, 4.3

[25] Real-time Innovations, "NDDS: The Real-time Publish-Subscribe Middleware." www.rti.com/products/ndds/ndwp0899.pdf, 1999. 5.1

[26] Prism Technologies, "OpenSplice Data Distribution Service." http://www.prismtechnologies.com/, 2006. 5.1

[27] OCI Integrated Information Systems, "TAO DDS Open Source Project." http://downloads.ociweb.com/DDS/, 2006. 5.1

[28] Object Management Group, *Management of Event Domains Specification*. Object Management Group, OMG Document formal/01-06-03 ed., June 2001. 5.1

[29] G. Singh, P. S. Kumar, and Q. Zeng, "Configurable Event Communication in Cadena," in *IEEE Real-time and Embedded Technology and Applications Symposium*, pp. 130–139, 2004. 5.2

[30] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, pp. 332–383, Aug. 2001. 5.2

[31] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, (Portland, OR), pp. 219–227, July 2000. 5.2

[32] R. Rajkumar, M. Gagliardi, and L. Sha, "The Real-time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-time Systems: Design and Implementation," in *First IEEE Real-time Technology and Applications Symposium*, May 1995. 5.2