Design Architectures and Performance Evaluation of Publisher/Subscriber Services in QoS-enabled Component Middleware*

George Edwards, Gan Deng and Aniruddha Gokhale
{george.t.edwards,gan.deng,a.gokhale}@vanderbilt.edu
Department of Electrical Engineering and Computer Science, Vanderbilt University
Nashville, TN 37235, USA

Abstract

The publisher/subscriber communication paradigm, such as the one provided by event-based communication services, is inherently well-suited to support large-scale distributed real-time and embedded (DRE) systems, such as avionics mission computing or distributed audio/video processing. Recent trends indicate that DRE systems are increasingly being developed by means of the component-based software development paradigm, in particular, using quality of service (QoS)-enabled implementations of the CORBA Component Model. However, these platforms do not yet provide any standard support for incorporating real-time event-based communication services and optimizations that have previously been developed and tested for object-oriented middleware like CORBA. The result is that developers of these large and complex DRE systems must custom-build publisher/subscriber solutions within component middleware. Applications that are developed in this ad hoc, tedious, and error-prone way are costly to design, implement and maintain, and more importantly, may not satisfy real-time requirements of DRE systems scalably. Therefore, DRE system developers urgently require real-time event services that support component-based applications while simultaneously preserve the performance optimizations required by DRE applications.

This paper makes three contributions to the R&D on supporting component-based real-time event services scalably for DRE systems. First, we outline three different design choices for incorporating publisher/subscriber services within QoS-enabled component middleware. Second, we describe our novel approach that draws on the strengths of using a patterns-driven solution to integrating, configuring and deploying real-time event services scalably in QoS-enabled component middleware. Third, we illustrate via empirical benchmarking results how component-based real-time event services satisfy the QoS and scalability requirements of DRE applications.

Keywords: Real-time Publisher/Subscriber Service, Component Middleware, CORBA Component Model, Model-based Systems, Patterns.

*This work was sponsored in part by grants from NSF ITR CCR-0312859, Siemens, and DARPA/AFRL contract #F33615-03-C-4112

1 Introduction

Large-scale, distributed real-time and embedded (DRE) systems are increasingly being used to control critical aspects of global infrastructure. For instance, DRE systems are now deployed in commercial air traffic control, military systems, electrical power grid, industrial process control, and medical imaging domains. Below we illustrates current trends in the software development of DRE systems while identifying core requirements of DRE systems that must be addressed to fulfil the promise held by these trends.

1.1 Increasing Use of Component Middleware for DRE Systems Software Development

To meet the challenges of developing and maintaining DRE systems, research over the past decade has focused on developing, optimizing, and standardizing *object-oriented middleware* [1], which is systems software that (1) resides between applications and the underlying operating systems, network protocol stacks, and hardware and (2) helps shield DRE applications from the accidental complexities of heterogeneous platforms by defining communication abstractions that can be implemented over a variety of networks and operating systems. Object-oriented middleware also implements reusable *services* (such as asynchronous event-based communication, global scheduling, and dynamic resource management) that provide functionality common to many DRE applications. Examples of object-oriented middleware for DRE systems include Real-time Java [2] run-time environments (*e.g.*, jRate [3]) and Real-time CORBA object request brokers (ORBs) [4] (*e.g.*, TAO [5]).

More recently, *component middleware* [6, 7] has defined additional capabilities that enhance object-oriented middleware for DRE systems, as follows:

- It defines a component abstraction that consists of a collection of (1) *interface ports*, exposing operations that clients can invoke to use a service provided by the component, or indicating the methods/services that the component itself depends on to achieve its functionality and (2) *event ports* that are used to publish and consume events.
- It allows developers to focus on programming their application "core logic" (*i.e.*, primary functionality), rather

than wrestling with lower-level tasks (*e.g.*, network programming, scheduling, security, and event processing). Instead, the application functionality is associated with the configuration-related capabilities via auto-generated component "glue code" that standardizes the interaction with other components and the middleware.

- It supports component *containers* that define a common operating environment in which a set of related components execute. Containers also provide components with key resources (*e.g.*, priority levels, real-time threads of control, and transparent state replication) and shield components from many tedious, error-prone, and non-portable complexities of the underlying networks, operating systems, and object-oriented middleware.
- It makes a significant attempt to address problems of configuring and deploying DRE applications throughout networks of heterogeneous computing nodes by providing standardized component assembly, packaging, and distribution formats. These configuration and deployment mechanisms enable the core functional issues to be decoupled from QoS-related issues so that QoS properties can be developed, configured, monitored, and managed not by those developing application functionality, but by a separate set of specialists (*e.g.*, middleware developers, systems engineers, and administrators) who are often much better positioned to make the appropriate configuration and deployment choices.

Examples of component middleware for DRE systems includes Prism [8] and the Lightweight CORBA Component Model (CCM) [9] (*e.g.*, CIAO [10]). Appendix A provides an overview of the capabilities of CCM.

1.2 Importance of Publisher/Subscriber Communication Paradigm in DRE Systems

An increasing number of DRE systems require 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, industrial process control requiring large scale automated system instrumentation and sensing/actuation, or supervisory control and data acquisition (SCADA) systems requiring real-time robust control and data communications.

Perhaps the most critical middleware service, therefore, for the types of DRE systems outlined above are a publisher/-subscriber service [11, 12]. The publisher/subscriber design is a powerful architecture for event-based communication because it provides anonymity, by decoupling event publishers and subscribers, and asynchronism, by automatically notifying subscribers when a specified event is generated. Moreover, the publisher/subscriber services reduces software dependencies through anonymity and asynchrony, thereby supporting the loose coupling required of these DRE systems. Publisher/-subscriber services are particularly relevant for large-scale DRE systems because they can support a changing set of requirements and environments by defining clear and crisp boundaries between various entities of an application.

One such publisher/subscriber service that supports several important features required by event-driven DRE applications is a real-time event service [13] available as part of the TAO [5] project, which is our open-source implementation of real-time CORBA. The real-time event service provides low-latency/-jitter event dispatching, support for periodic processing, centralized event filtering, and efficient use of network and computational resources. Additionally, many of the requirements of these DRE systems can be met scalably and robustly via federations of such real-time publisher/subscriber services.

With the increasing use of QoS-enabled component middleware to develop DRE systems compounded by the need for real-time publisher/subscriber services to support a class of DRE systems, therefore, requires integration of real-time publisher/subscriber services within QoS-enabled component middleware. However, standards-based component middleware do not specify how publisher/subscriber services can be robustly supported within component middleware. Although performance evaluation metrics for real-time publisher/subscriber object-oriented middleware services are available [14], there is a general lack of information and insights into the design and performance evaluation of real-time publisher/subscriber services within QoS-enabled component middleware. This paper focuses on addressing this dimension of research.

Our earlier work in the space of integrating publisher/-subscriber services and supporting a federation of these within QoS-enabled component middleware has focused on (1) how software design patterns can be used to provide a robust integration of event-based communication services within QoS-enabled component middleware [15], (2) how a federation of real-time publisher/subscriber services can be configured and deployed in the context of QoS-enabled component middleware [16], and (3) how model-driven generative tools can be used to alleviate accidental complexities in configuring and deploying these services within QoS-enabled component middleware [17].

This paper extends our previous work by (1) analyzing the pros and cons of different design choices for integrating real-time publisher/subscriber services within QoS-enabled component middleware (2) illustrating how container-

based RT publisher/subscriber services leverage the benefits of the component-oriented software development paradigm, and (3) demonstrating empirically that QoS-enabled component middleware-based RT publisher/subscriber services (specifically the TAO Real-time Event Service) can successfully maintain the QoS requirements of large-scale DRE applications.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes three different architectural choices for integrating real-time publisher/subscriber services within QoS-enabled component middleware illustrating the pros and cons of each approach; Section 3 describes the challenges in integrating real-time event services within component middleware and our solutions to address these; Section 4 provides the results of benchmarking experiments that establish the feasibility of container-based RT event services; Section 5 compares our results to related research; and Section 6 presents concluding remarks.

2 Architectural Design Choices for Composing Real-time Event Services in QoS-enabled Component Middleware

The CORBA Component Model (CCM) standard does not specify how publisher/subscriber services can be composed within component middleware. This issue is further complicated by the need for real-time publisher/subscriber services to be integrated within QoS-enabled implementations of CCM, such as our Component Integrated ACE ORB (CIAO) [18].

This section describes three possible architectural choices for integrating real-time publisher/subscriber services within the CORBA Component Model. In this paper we focus on integration of a real-time event channel, which is a manifestation of the publisher/subscriber communication paradigm. Figure 1 illustrates how these architectural choices vary based on where within the component middleware architecture does an event channel gets composed. The three architectural choices include (1) *component-based* – where the event channel can be represented as an application-level component, (2) *container-based* – where the event channel can be encapsulated within the container, and (3) *component server-based* – where the event channel can reside within the component server. This section describes each architecture choice in detail analyzing the advantages and disadvantages of each approach. Section 4 subsequently provides empirical guidance to support our analysis.

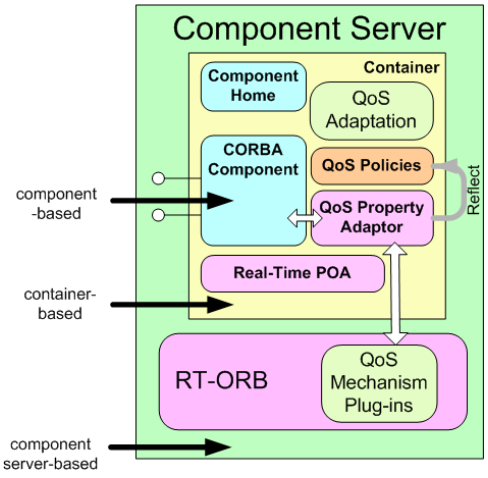

Figure 1: **Architectural Choices for Integrating Publisher/-Subscriber Services in CCM**

2.1 Component-level Event Channels

• **Design** The first architecture choice for providing real-time event services in component middleware is to instantiate event channels as application-level components as illustrated in Figure 2. In this architecture, the interface normally provided by the event channel is exposed as a facet. This interface contains methods to connect to the event channel, configure real-time properties, and push events. The event channel interface is transformed from that of a standard CORBA object into that of a CORBA component.

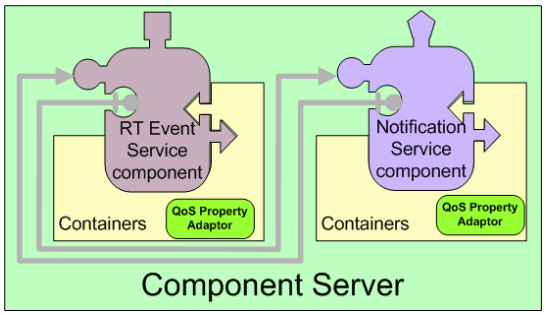

Figure 2: **Publisher/Subscriber Service As CCM Component**

• **Analysis** The primary advantage to this approach is its simplicity. The complexity needed to implement an event channel component is rudimentary since the encapsulated service already implements the event channel functions. Also, this architecture does not restrict event channel capabilities in any way. The full set of event channel features is available to other components. Instantiating and deploying multiple event

channels is straightforward and follows the same rules as apply to standard components.

There are a number of disadvantages to using component-level event channels. Generally speaking, the shortcomings of the object model remain present in this architecture. First, the component glue-code, or servant, must manipulate event service interfaces directly, which exposes low-level CORBA programming details. Second, the component servant logic must encapsulate QoS and RT properties, which inhibits the flexibility and reusability of components across different operating contexts and environments. Third, it is impossible to substitute or interchange different publisher/subscriber services without recompilation of components. Fourth, application level components must now be responsible for managing event channel lifecycles.

The net result of these effects is that component-level composition of event channel configurations must be decided at design-time, rather than at deployment time. Additionally, the event channel component architecture conflicts with the standard CCM container programming model, which establishes the container to be sole mediator between application-level components and common middleware services, such as real-time event services. Ironically, in this case the event service itself is encapsulated within the component. Finally, this architecture always results in a remote call to transmit an event to the event channel component, which must be handled by the ORB and 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 applications that require stringent QoS guarantees at a low cost.

2.2 Container-level Event Channels

• **Design** Figure 3 depicts a second possible architecture for providing real-time event services in component middleware where an event channel is encapsulated within the container. In this architecture, the container is responsible for managing event channel lifecycles, initializing channels and gateways that are necessary in the context of a federation, connecting publishers and subscribers, configuring QoS and RT properties, implementing publisher and subscriber servants, and terminating services. The container also exposes two distinct interfaces. One interface provides configuration methods and is invoked by the component deployment framework based on the properties specified in XML descriptors that describe configuration decisions. The second interface provides a push method and is invoked by application-level components.

• **Analysis** There are many advantages to this architecture. First, application components are totally isolated from event service implementation details. This 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 event service (since the deployment framework must instantiate and configure channels) rather than component servant glue code, it is not important for the deployment framework to be lightweight.

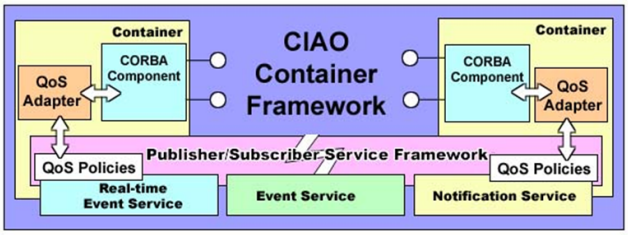

Figure 3: **Publisher/Subscriber Service Within CCM Container**

Second, QoS properties are specified via XML and encapsulated within the container. This 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 component logic or glue-code thereby obviating the need for recompilation. Third, this architecture aligns with the CCM container programming model and delays event configuration-related decisions until deployment time, which allows additional optimizations to be incorporated depending on the 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 event service configuration.

The disadvantage to the container-level event channel architecture is the difficulty encountered in actually implementing it effectively and efficiently for multiple dissimilar services. There are a number of design challenges that arise when pursuing this method of event channel integration. The various design challenges and patterns-based solutions appear in Section 3.

2.3 Component Server-level Event Channels

• **Design** The third possible architecture for providing real-time event services in component middleware is to place event channels within the component server. In this architecture, event channels are still accessed and manipulated via the container. However, the component server-level architecture is fundamentally different from the container-level architecture because the component server is a lower-level entity. Only a single component server exists on each host in the distributed

computing environment, so this architecture results in all the components that execute on a given host sharing the same event channel.

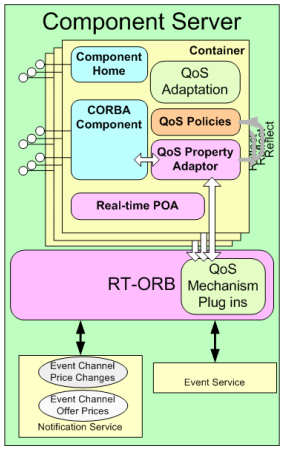

Figure 4: **Publisher/Subscriber Service Within Component Server**

• **Analysis** The advantages present in the container-level architecture are also applicable to the component server-level architecture: components are still isolated from event services in such a way that they remain configurable after compilation, their memory footprint is minimized, and push operations result in only local method invocations. However, the component server-level architecture is more coarse-grained *i.e.*, a large number of components may be required to share a single channel. This result may be an advantage or disadvantage depending on the specific component application.

For applications that require either multiple event channels on a single host or that wish to maximize component flexibility to allow for future enhancements or modifications, the component server-level architecture may be too restrictive. On the other hand, for applications that do not require these capabilities, the component server-level architecture results in a simpler configuration and deployment process, which reduces developer effort. In the case of very large-scale DRE systems, the savings may be substantial. Ultimately, the question of whether to employ container or component server event channels can only be answered by the application developer.

Based on our analysis of the pros and cons of each design choice, we have selected the container-based event channel as the target to obtain additional guidance on its applicability and performance. The container-based architecture provides the most flexible real-time event service, while preserving the benefits of the CCM container programming model. As described in [15], this architecture can be implemented in a way that is very fast, lightweight, and flexible enough to accommodate new services in the future with little modification. Consequently, the container-based architecture will be useful for the widest range of DRE component applications.

3 CIAO Container Framework Design Goals and Implementation Strategies

This section describes how the CIAO publisher/subscriber service architecture, shown in Figure 3, employs patterns to address the design goals of the container programming model outlined in Section 2. For example, while maintaining efficiency and reliability requirements, CIAO preserves the lightweight nature of components. An individual component need know nothing about the services that implement event-passing – the container encapsulates that complexity. It therefore follows that component application developers need not be concerned with these details, further simplifying the design of the core component logic.

For each design goal mandated by the CCM container programming model, our pattern-oriented solution is detailed below. Figure 5 illustrates a pattern language [19] demonstrating the interactions between patterns in the publisher/subscriber service framework integrated within a container.

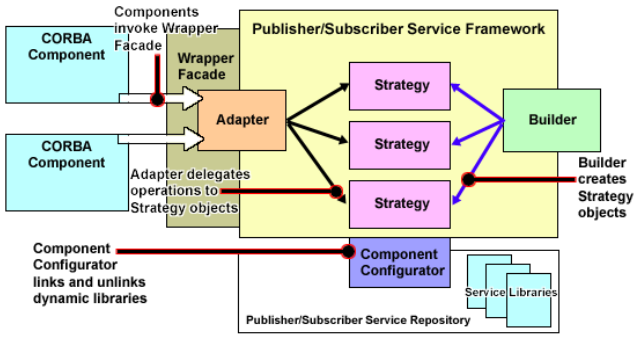

Figure 5: **Pattern Interactions in the CIAO Publisher/Subscriber Service Framework**

Design goal 1: Provide a service-independent representation of real-time properties.

Solution approach → Adapter pattern: Different publisher/subscriber services depend on different representations of real-time properties. The CIAO container implements an adapter that converts a service-independent representation of real-time properties (as specified in XML) into a service-specific representation. 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 publisher/subscriber services, the interface exposed to components does not change.

Design goal 2 : Enhance reuse and extensibility by allowing new publisher/subscriber services to be easily plugged-in.

Solution approach → Strategy pattern: This design 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: Reduce the memory footprint of the container by decoupling the creation of publisher/subscriber service instances from their representation.

Solution approach → Builder pattern: The creation of real-time event channel instances is somewhat complex in CIAO since they must be initialized properly. CIAO defines a builder class that encapsulates the complexity of creating and initializing event channels. The result is finer control of the construction process, isolation of construction code, and the ability to vary the publisher/subscriber service implementation.

Design Goal 4: Ensure components incur only the cost of services that are required by deferring publisher/subscriber service selection and configuration decisions until run-time.

Solution approach → Component Configurator pattern: In CIAO, publisher/subscriber service libraries are 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: Enable component access to the full set of QoS features available in publisher/subscriber 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 that forwards invocations to the corresponding service-specific operations for each publisher/subscriber service. This design results in a concise and robust programming interface capable of configuring the QoS features in multiple dissimilar publisher/subscriber services.

4 Performance Evaluation of Component Middleware-based Real-time Event Services

With increasing use of QoS-enabled component middleware to develop and deploy DRE systems, it is important to ascertain the real-time performance of QoS-enabled component middleware, in particular that of real-time publisher/subscriber services integrated within QoS-enabled component middleware. This section provides results of empirically evaluating performance of a real-time event channel in CIAO, which is our QoS-enabled component middleware implementation. Our results focus on the container-level integration of event channels described in Section 2.2.

4.1 Experiment Domain

The domain we used for our experiments to empirically benchmark performance of RT event channel integration within QoS-enabled component middleware comprises a hypothetical aerial battlefield theatre depicted in Figure 6.

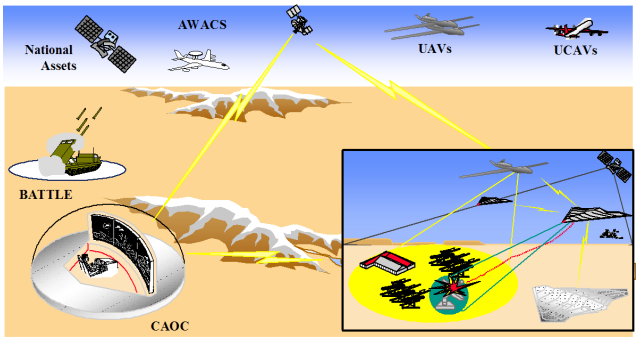

Figure 6: **Hypothetical Aerial Warfare Scenario**

In this military application scenario, a number of Unmanned Aerial Vehicles (UAVs) simultaneously transmit surveillance video from different regions in a battlefield. A UAV is a remote-controlled aircraft that is launched in order to obtain a view of an engagement and performs such functions as spotting enemy movements and locating targets in a battlefield.

A Distributor node receives video streams from UAVs and forwards it to different receivers on the *Command and Operations Control* (CAOC). The CAOC entity monitors these video streams and is responsible for making tactical decisions about vehicle deployment and weapon system guidance. Some of the CAOC receivers perform automatic target recognition to guide Unmanned Combat Aerial Vehicles (UCAV), while others provide commands to launch weapons from ground stations.

This application possesses features that make it a complex and challenging problem, including large-scale, stringent real-time requirements, resource constraints, and the distributed communication environment (which includes low-bandwidth, high-loss wireless links) that requires the use of publisher/-subscriber paradigm.

4.2 Experiment Description

This section describes our experiment by detailing the role of various entities. Figure 7 shows the topology of our aerial warfare theatre prototype. We modeled this topology using our EQAL modeling tool [15, 16]. EQAL is a modeling tool that resolves accidental complexities of configuring and deploying large-scale publisher/subscriber component middleware-based DRE systems. We applied our container-based real-time event service [20] to the prototype system.

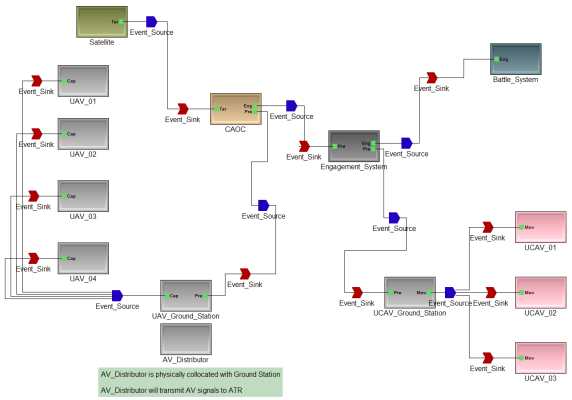

Figure 7: **BBN UAV Scenario**

In the prototype system shown in Figure 7, we have modeled and implemented seven different types of CIAO CCM components: Satellite component (`Satellite`), CAOC component (`CAOC`), UAV Ground Station component (`UAV_GS`), UAV component (`UAV`), Engagement System component (`Engagement_Sys`), Battle System (`Battle_Sys`), UCAV Ground Station component (`UCAV_GS`) and UCAV component (`UCAV`).

The interactions among these components is described below:

- **Satellite:** Satellite is a component that is deployed for a surveillance task. Whenever it detects abnormal situations in the battlefield, it will emit an event of type `TargetAbnormal`. The CAOC component is the only subscriber for such events.
- **CAOC:** The CAOC component acts as the command and control center for the battle. Whenever it receives a `TargetAbnormal` event, it will emit an event of type `PrepareCapture` to instruct the UAV Ground Station (UAV_GS) to prepare to capture an image in the field. The UAV_GS is the only subscriber for this event type. Also, whenever there is any military need to engage the battle systems, the CAOC will emit a `Prepare_Engage` event type to the Engagement System (Engagement_Sys).
- **UAV_GS:** When UAV_GS component receives a `PrepareCapture` event, it publishes a `CaptureImage` event. Only those UAVs that are in the surrounding target field will receive the `CaptureImage` event.
- **UAV** When a UAV component receives a *CaptureImage* event, it will begin to execute the surveillance task by capturing images of the target field and transmitting the image back to the CAOC through the A/V Distributor. The image transmission is implemented via the CORBA A/V stream service.
- **Engagement_Sys:** Whenever the Engagement_Sys component receives a `Prepare_Engage` type of event, it will publish an `Engage` event to the UCAV Ground Station (UCAV_GS) and Battle System (Battle_Sys). The UCAV_GS and Battle_Sys are the only components that are the subscribers for `Engage` events.
- **Battle_Sys:** Whenever the Battle_Sys component receives an `Engage` event, it will initiate engagement by preparing for weapon launching.
- **UCAV_GS:** Whenever the UCAV_GS component receives an `Engage` event, it will dispatch a `StartLaunch` event to the appropriate UCAV components.
- **UCAV:** Whenever a UCAV component receives a `StartLaunch` event, it will direct itself to the target location.

There are a few interesting observations to note in this scenario:

- most transmission links are wireless links imposing stringent bandwidth and latency constraints,
- there is a need for prioritization between different types of event transmissions, and
- reliability of delivery of event must be guaranteed.

Building flexible application software and object-oriented middleware that meets these requirements is challenging because the need for determinism and predictability often results in tightly-coupled designs. For instance, conventional mission-critical applications built with object-oriented middleware consist of closely integrated responsibilities; each component, even with the benefit of a real-time event service, will still have to write application code to handle multiple aspects, such as real-time event dispatching, scheduling and periodic event processing. Tight coupling often yields inflexibility and thus can substantially increase the effort and cost of integrating new and improved system features. By using the container-based real-time event service, however, it is still possible to build flexible and loosely-coupled components, yet simultaneously meet QoS requirements.

As shown in the Figure 7, there are three event propagation chains in the system. The first one begins when the Satellite component emits a `TargetAbnormal` event and ends when the UCAV components receive the `CaptureImage` events. The second one starts with the CAOC emitting the `PrepareEngage` event and terminates with the Battle System receiving the `Engage` event. The third one begins when the CAOC emits the `PrepareEngage` event and ends when the UCAVs receive the `StartLaunch` events.

The most important metric for any event service is event latency, i.e., the time elapsed from when a supplier sends an event until the last consumer interested in the event receives it. Our goal is to demonstrate and benchmark the publisher/subscriber features in the experiment described above when used in the context of a component middleware and a container-based event channel. The rest of the section presents our experiment results with different real-time event service settings, illustrating their effect on the provisioning of event communication QoS in the DRE system.

4.3 Experimental Measurement Approach

Measuring the latency in the event service is hard since events are delivered via a uni-directional flow of communication from suppliers to consumers. As shown below, event delivery time and jitter is comparable to the network propagation delay, because a distributed clock precision is bounded by the jitter [21], measuring the latency of event propagation, where event supplier and event consumer are deployed in different hosts, is impossible. Fortunately, the latency for the centralized configuration can be measured directly using a consumer located in the same host as the supplier and measuring the roundtrip delay as shown in Figure 8.

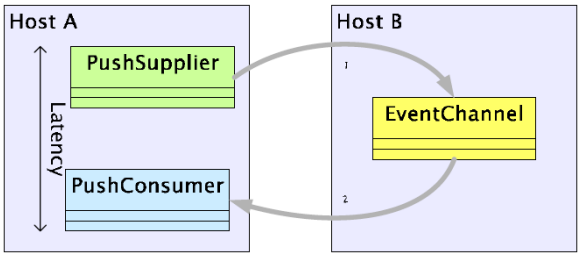

Figure 8: **Experimental Setup to Measure Event Service Latency**

The performance tests were conducted on a testbed where each machine was a dual Intel Xeon 2.4G CPU, with 1G RAM running Windows XP Professional, and all these machines are connected through 100M LAN. For all our measures we use the high-resolution timer (under 2 nanosecond resolution) available on Pentium processors. This timer is implemented via a special register that counts the number of clock ticks since the CPU was reset.

4.4 Metrics for Collocated Components

Before we tested our prototype system and measured results in the experiment situations where all CCM components are distributed into different hosts, we first show our results where all the components are collocated in the same host and same process, *i.e.*, within the same component server. By doing so we are able to mask the effects of network latencies, data rate loss and distribution, thereby, permitting us to gauge the actual performance of the real-time event dispatching within a QoS-enabled component middleware. Figures 9, 10 and 11 show the collocated event propagation delays for the three event propagation chains described earlier, respectively.

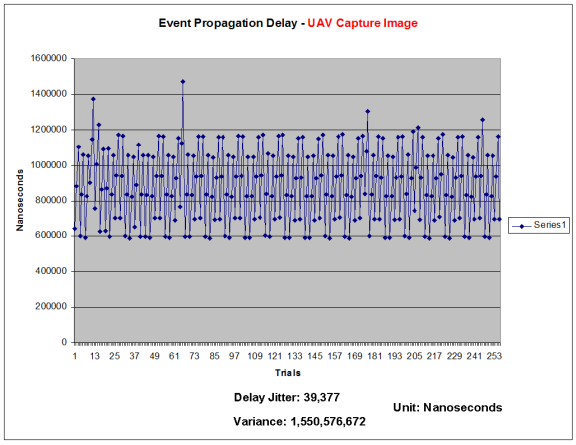

Figure 9: **Collocated UAV Results**

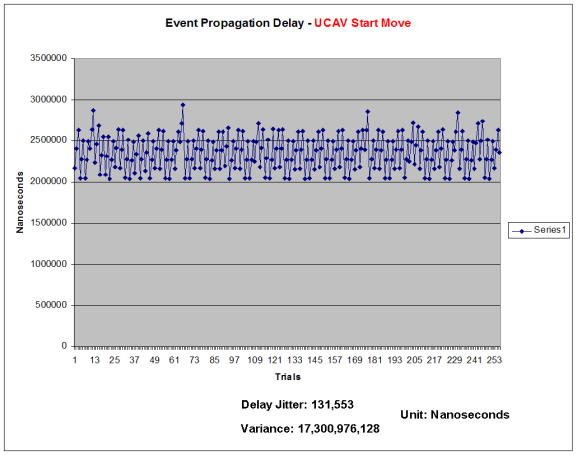

Figure 10: **Collocated UCAV Results**

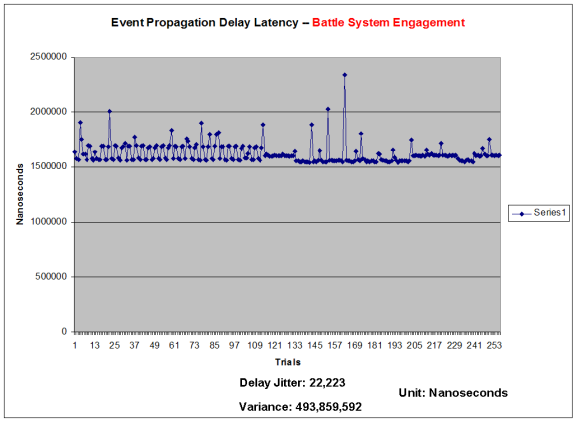

Figure 11: **Collocated Battle Results**

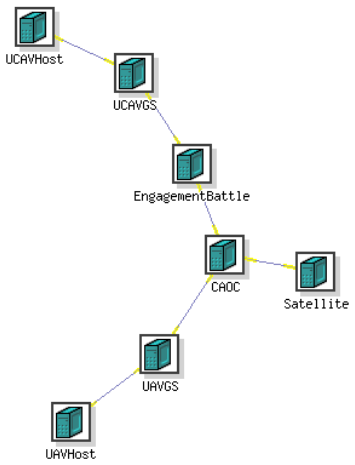

Figure 12: **The Network Topology of Experiment for Distributed Case**

Our collocation experiment results indicate that the event dispatching overhead is minimal thereby confirming that performance optimizations of object-oriented real-time event dispatching are preserved within a container-based solution.

In many real-world situations, multiple components are deployed within the same process but still communicate with each other through CCM ports, *i.e.*, facets, receptacles, event sources, and event sinks. To improve the performance and predictability of collocated component communication, the process-collocation optimization was applied to CIAO's implementation. Process-collocation improves the performance and predictability transparencies for objects that reside in the same address space as the servant implementation, while maintaining locality transparency. To implement process-collocation, the ORB must identify the location of the component's reference without explicit application programmer intervention and without violating the policies specified by POAs and the component containers. Once the ORB determines that a reference is collocated in the same process, all operation invocations can be forwarded to a special collocation stub. The goal of process-collocation optimization is to ensure the performance of accessing in-process collocated components is comparable to accessing standard C++ components, while still providing predictability transparency in the framework.

4.5 Metrics for Distributed Components

In this experiment, we performed latency and jitter measurements on the prototyped UAV application where components are deployed across multiple hosts. This experiment provides a realistic picture of the system performance in a typical computing environment. The network topology is shown in Figureẽffig:network-topology.

Our measurements demonstrate that CIAO's container-based real-time event service provides the performance predictability needed by a representative DRE application. Delays for the same three event propagation chains are given below.

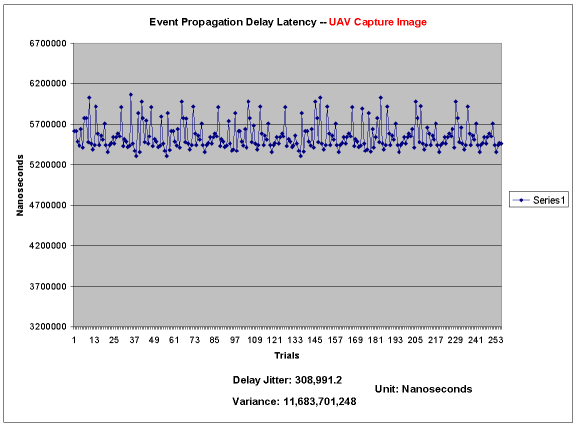

Figure 13: **Distributed Case UAV Results**

As can be seen from the above figures, the performance of the federation of event channels integrated within different containers continues to be predictable with minimal jitter, however, with an additional cost of network latencies. Our testbed comprised networks (LANs) that were lightly loaded. We are currently performing additional experiments on the Emulab [22] testbed, where we can emulate different network types and link characteristics.

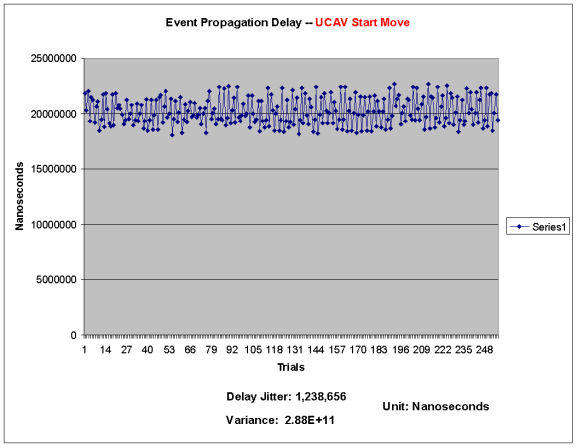

Figure 14: **Distributed Case UCAV Results**

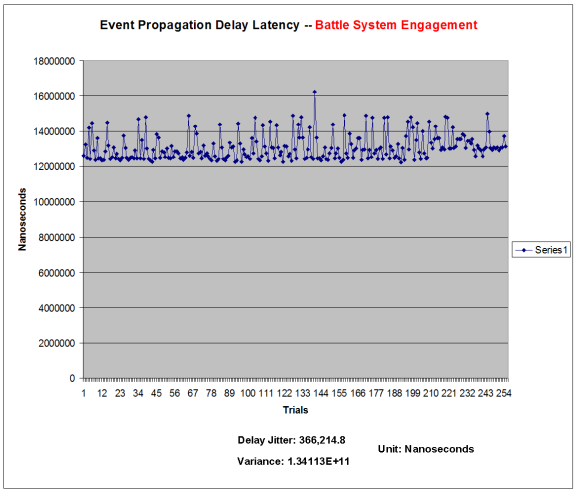

Figure 15: **Distributed Case Battle Results**

5 Related Work

This section surveys literature on Publisher/Subscriber systems, both standards based and proprietary, concentrating on publisher/subscriber systems for real-time systems. None of the prior work deals with component middleware with *real-time QoS support*, which is the focus of our work.

5.1 Standards-based Publisher/Subscribers

• **The CORBA Event Service** The CORBA Event Service is a specification that defines a basic Publisher/Subscriber architecture for CORBA systems. In the CORBA Event Service the Publishers are called *Suppliers*. Suppliers can be either push-style or pull-style, the first pro-actively generate events and send those events to the event channel for distribution. Pull-style suppliers passively wait until the event channel requires an event and supply one if the event is available. Likewise, the CORBA Event Service uses *Consumers* to play the Subscriber role. Consumers can also be pull-style or push-style, the first style requires consumers to periodically or pro-actively query events from the event service. Push-style consumers are simply notified when a new event is available. Both styles can be mixed, in other words, push-style consumers can receive events generated by pull-style suppliers, or vice-versa. The CORBA Event Service does not provide any mechanisms to control QoS behavior, and filtering is limited to the so-called *Typed Event Channels* where consumers and suppliers can receive a specific IDL-type. However, Typed Events is an optional compliance point in the CORBA Event Service and very few implementations for this variant of the service exist. Therefore, in practice, filtering is not supported.

There are several commercial CORBA-compliant Event Service implementations available from multiple vendors, such as IONA and Borland. IONA also produces OrbixTalk, which is a messaging service based on UDP/IP multicast. Since the CORBA Event Service specification does not address issues critical for real-time applications, the QoS behavior of these implementations are not acceptable solutions for many application domains.

• **The CORBA Notification Service** The OMG has issued a specification for a Notification Service [23], which is a superset of the CORBA Event Service. This specification adds interfaces for event filtering, configurable event delivery semantics (e.g., at least once or at most once), security, and event delivery QoS.

• **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" [24]. 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. Also, both the CORBA Notification Service and the CORBA Distributed Notification Service are based on Object-Oriented middleware rather than component middleware.

5.2 Proprietary Publisher/Subscriber Systems

• **Cadena Event Channel Framework** Cadena Event Communication Framework [25] includes a CORBA-based event channel which has been integrated into the OpenCCM infrastructure. The framework implements a number of features of the event service middleware, such as event filtering, event correlation, and direct-dispatching. Although this work comes close to our work, however, it does not address the problems

such as real-time event scheduling and dispatching and periodic event processing, which are crucial for a lot of mission-critical real-time applications. Our work, on the other hand, leverages the RT event channels and QoS-enabled component middleware to provide the properties outlined above.

• **SIENA** SIENA [26, 27] 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. For example, the authors propagate filtering information as close to the sources as possible, likewise, filtering constraints are combined and simplified to minimize the use of computation resources in the routers.

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

• **ECO** The Distributed Systems Group at Trinity College, Dublin has developed ECO [28], for "Events, Constraints and Objects." The authors propose building programs out of cooperating objects that publish or subscribe to events as needed. Filtering, concurrency and timeliness constraints are expressed as constraints on the events that a particular object publishes or subscribes to.

The authors propose extending general-purpose programming languages, such as C++ or Java, to include explicit declarations for events, as well as the types of events that a class can subscribe or publish. Naturally, this static publications or subscriptions can be further restricted at run-time, the authors propose using new language statements for this purpose.

Objects can add *Notify* constraints to limit the objects that they subscribe to, this conditions are evaluated at the source of the event and thus are limited to constraints on the event parameters or the source identity. Objects can also define *Pre* and *Post* constraints, this are evaluated on the destination object and can use the state of the receiving object to affect the event processing. *Pre* constraints are evaluated before the event is delivered and can determine if the event is dropped, enqueued or processed immediately.

• **CMU Real-time Publisher/Subscriber** Rajkumar, *et al.*, describe a real-time publisher/subscriber prototype developed at CMU SEI [29]. Their Publisher/Subscriber model is functionally similar to the CORBA Event Service, though it defines its own programming APIs and communication protocols. The authors details how real-time threads and adequate synchronization primitives can be used to implement the RT P/S model without undue priority inversions. However, the authors also fail to recognize that adequate synchronization primitives are a necessary condition to address unbounded priority inversions, but it is not a sufficient condition. For example, the authors do not detail how their algorithms avoid critical sections with time bounds dependent on the number of participants in the system.

5.3 Summary

Much has been written about Publisher/Subscriber systems, but there is little effort has been documented on the patterns, optimizations and architectures required to implement QoS-aware Publisher/Subscriber 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, even when research concentrates on real-time applications.

6 Conclusions

R&D over the past decade on object-oriented standards middleware, such as CORBA, has demonstrated its effectiveness in developing and supporting QoS requirements of large-scale distributed and real-time systems. Over the past few years, OO middleware has been evolving into component-oriented middleware. Component-based software development has already received widespread acceptance in the enterprise business and desktop application domains. However, developers of distributed real-time and embedded (DRE) systems have encountered limitations with the available component middleware platforms, such as the CORBA Component Model (CCM) and the Java 2 Enterprise Edition (J2EE). These limitations often preclude developers of DRE systems from fully exploiting the benefits of component software. In particular, component middleware platforms lack standards-based publisher/subscriber communication mechanisms that support key quality-of-service (QoS) requirements of DRE systems, such as low latency, bounded jitter, and end-to-end operation priority propagation. QoS-enabled publisher/subscriber services are available in object middleware platforms, such as Real-time CORBA, but such services have not been integrated into component middleware due to a number of development and configuration challenges.

This paper provides three approaches to integrating real-time publisher/subscriber services within QoS-enabled com-

ponent middleware. The pros and cons of individual integration approaches are described. We also provide performance evaluation for the container-based integration approach, which provides the least intrusive and most standards-conforming approach. The collocated case results, which demonstrate event dispatching overhead, provides sufficient guidance to indicate that all the real-time properties of real-time event channels is preserved. The distributed case reveals the performance continues to remain predictable, however, with the additional cost of network latencies, which is expected.

Our ongoing R&D in this field involves the evaluation of the remaining design choices and application of this technology in the context of a variety of DRE domains, including telecom, avionics mission computing, software radio and industrial process control. The software described in this paper is available for download from `www.dre.vanderbilt.edu/CIAO` (for the CIAO QoS-enabled component middleware) and `www.dre.vanderbilt.edu/cosmic` (for the CoSMIC modeling tools we used for configuring and deploying event channels within CIAO).

References

[1] Michi Henning and Steve Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley, Reading, MA, 1999.

[2] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull, *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[3] Angelo Corsaro and Douglas C. Schmidt, "The Design and Performance of Real-time Java Middleware," *IEEE Transactions on Parallel and Distributed Systems*, 2003.

[4] Object Management Group, *Real-time CORBA Specification*, OMG Document formal/02-08-02 edition, Aug. 2002.

[5] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

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

[7] Clemens Szyperski, *Component Software—Beyond Object-Oriented Programming*, Addison-Wesley, Santa Fe, NM, 1998.

[8] David C. Sharp and Wendy C. Roll, "Model-Based Integration of Reusable Component-Based Avionics System," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

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

[10] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill, "QoS-enabled Middleware," in *Middleware for Communications*, Qusay Mahmoud, Ed. Wiley and Sons, New York, 2003.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture—A System of Patterns*, Wiley & Sons, New York, 1996.

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

[13] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, Atlanta, GA, Oct. 1997, ACM, pp. 184–199.

[14] Douglas C. Schmidt and Carlos 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.

[15] George Edwards, Douglas C. Schmidt, Aniruddha Gokhale, and Bala Natarajan, "Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems," in *Proceedings of the 42nd Annual Southeast Conference*, Huntsville, AL, Apr. 2004, ACM.

[16] Gan Deng, Aniruddha Gokhale, and Bala Natarajan, "Model-driven Integration of Federated Event Services in Real-time Component Middleware," in *Proceedings of the 42nd Annual Southeast Conference*, Huntsville, AL, Apr. 2004, ACM.

[17] George Edwards, Gan Deng, Douglas C. Schmidt, Anirudda Gokhale, and Balachandran Natarajan, "Model-driven Configuration and Deployment of Component Middleware Publisher/Subscriber Services," in *Submitted to the Third International Conference on Generative Programming and Component Engineering (GPCE)*, Vancouver, CA, Oct. 2004, ACM.

[18] Institute for Software Integrated Systems, "Component-Integrated ACE ORB (CIAO)," www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.

[19] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel, *A Pattern Language*, Oxford University Press, New York, NY, 1977.

[20] David A. Karr, Craig Rodrigues, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas C. Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," in *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, Rome, Italy, Sept. 2001, OMG.

[21] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[22] Robert Ricci and Chris Alfred and Jay Lepreau, "A Solver for the Network Testbed Mapping Problem," *SIGCOMM Computer Communications Review*, vol. 33, no. 2, pp. 30–44, Apr. 2003.

[23] Object Management Group, *Notification Service Specification*, Object Management Group, OMG Document telecom/99-07-01 edition, July 1999.

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

[25] Gurdip Singh, Bob Maddula, and Qiang Zeng, "Event Channel Configuration in Cadena," in *Proceedings of the IEEE Real-time/Embedded Technology Application Symposium (RTAS)*, Toronto, Canada, May 2004, IEEE.

[26] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, Aug. 2001.

[27] Antonio Carzaniga, David S. Rosenblum, and Alexander 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, July 2000, pp. 219–227.

[28] Gradimir Starovic, Vinny Cahill, and Brendan Tangney, "An event based object model for distributed programming," in *OOIS (Object-Oriented Information Systems) '95*, London, 1995, pp. 72–86, Springer-Verlag.

[29] Ragunathan Rajkumar, Mike Gagliardi, and Lui 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.

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

[31] Aniruddha Gokhale, Douglas C. Schmidt, Balachandra Natarajan, and Nanbor Wang, "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, vol. 45, no. 10, Oct. 2002.

[32] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *The Journal of Microprocessors and Microsystems*, vol. 27, no. 2, pp. 45–54, mar 2003.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

Appendices

A Overview of CORBA Component Model

The CORBA Component Model (CCM) forms a key part of the CORBA 3.0 standard [30]. CCM is designed to address the limitations with earlier versions of CORBA 2.x middleware that supported a distributed object computing (DOC) model [31]. Figure 16 depicts the layered architecture of the CCM model. The remainder of this section describes the key

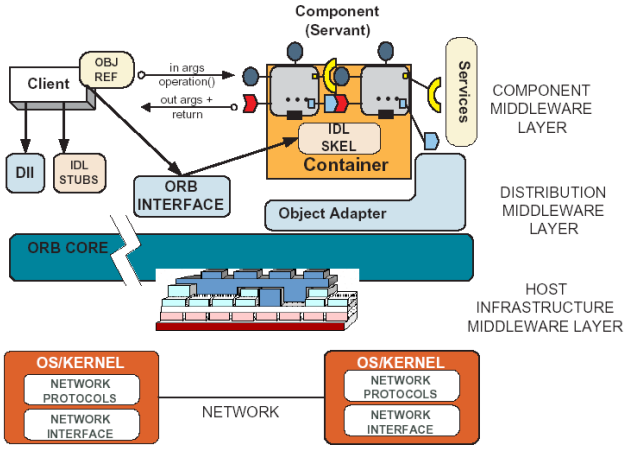

Figure 16: **Layered CCM Architecture**

CCM elements in Figure 16.

Components. *Components* in CCM are implementation entities that collaborate with each other via *ports*. CCM supports several types of 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 messages with one or more components.

Container. A *container* in CCM provides the run-time environment for one or more components that manages various pre-defined hooks and strategies, such as persistence, event notification, transaction, and security, used by the component(s). Each container is responsible for (1) initializing instances of the component types it manages and (2) connecting them to other components and common middleware services. Developer-specified metadata expressed in XML can be used to instruct CCM deployment mechanisms how to control the lifetime of these containers and the components they manage. The meta-data is present in XML files called *descriptors*.

Component assembly. In a distributed system, a component may need to be configured differently depending on the context in which it is used. As the number of component configuration parameters and options increase, it can become tedious and error-prone to configure applications consisting of many individual components. To address this problem, the CCM defines an *assembly* entity to group components and characterize the meta-data that describes these components in an assembly. Each component's meta-data in turn describes the features available in it (*e.g.*, its properties) or the features that it requires (*e.g.*, its dependencies). CCM assemblies are defined using XML Schema templates, which provide an implementation-independent mechanism for describing component properties and generating default configurations for CCM components. These assembly configurations can preserve the required QoS properties [32] and establish the necessary configuration and interconnections among groups of components.

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 CCM component server is a singleton [33] that plays the role of a factory to create containers and standardizes the role of a server process in the CORBA 2.x object model. Each component server is typically assigned a particular set of capabilities within a distributed system.

Component packaging and deployment. In addition to the run-time building blocks outlined above, the CCM also standardizes component implementation, packaging, and deployment mechanisms. Packaging involves grouping the implementation of component functionality – typically stored in a dynamic link library (DLL) – together with other meta-data that describes properties of this particular implementation. The CCM Component Implementation Framework (CIF) helps generate the component implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL).