Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems

George T. Edwards Balachandran Natarajan Douglas C. Schmidt Aniruddha Gokhale

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37203

Contact Author: g.edwards@vanderbilt.edu

Abstract

Although component-based software development has widespread acceptance in the enterprise business and desktop application domains, developers of distributed, real-time and embedded (DRE) systems have encountered limitations with the available component middleware platforms. 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, 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 contributions to the integration of publisher/subscriber services in component middleware. First, we outline key challenges associated with integrating publisher/subscriber services into component middleware. Second, we describe a methodology resolving these challenges based on software design patterns and Model-Driven Middleware (MDM). Third, we describe a pattern-oriented component middleware platform and a complementary MDM tool that we have developed to integrate publisher/subscriber services into component middleware applications.

1 Introduction

To reduce the complexity of designing robust, efficient, and scalable distributed real-time and embedded (DRE) software systems, developers increasingly rely on *middleware* [1], which is software that resides between applications and the lower-level run-time infrastructure, such as operating systems, network protocol stacks, and hardware. Middleware isolates DRE applications from lower-level infrastructure complexities, such as heterogeneous platforms and error-prone network programming mechanisms. It also enforces essential end-to-end quality of service (QoS) properties, such as low latency and bounded jitter; fault propagation/recovery across distribution boundaries; authentication and authorization; and weight, power consumption, and memory footprint constraints.

Over the past decade, middleware has evolved to support the creation of applications via composition of reusable and flexible software *components* [2]. Components are implementation/integration units with precisely-defined interfaces that can be installed in application server run-time environments. Examples of commercial-off-the-shelf (COTS) component middleware include the CORBA Component Model (CCM), J2EE, and .NET.

Component middleware generally supports two models for component interaction: (1) a *request-response* communication model, in which a component invokes a point-to-point operation on another component, and (2) an *event-based* communication model, in which a component transmits arbitrarily-defined, strongly-typed mes-

sages, called *events*, to other components. Event-based communication models are particularly relevant for large-scale DRE systems (such as avionics mission computing, distributed audio/video processing, and distributed interactive simulations) because they help reduce software dependencies and enhance system composability and evolution. In particular, the *publisher/subscriber* architecture [3] of event-based communication allows application components to communicate anonymously and asynchronously. The publisher/subscriber communication model defines three software roles:

- **Publishers** generate events to be transmitted.
- **Subscribers** receive events via hook operations.
- **Event channels** accept events from publishers and deliver events to subscribers.

The publisher/subscriber design is an especially powerful architecture for event-based communication because it provides (1) *anonymity* by decoupling event publishers and subscribers and (2) *asynchrony* by automatically notifying subscribers when a specified event is generated.

QoS-enabled component middleware platforms leverage the benefits of component-centric software development while simultaneously preserving the optimization patterns and principles of distributed object computing middleware. Before DRE application developers can derive benefits from QoS-enabled components, however, common middleware services must be integrated with component middleware in a manner that minimizes the complexity associated with configuration and deployment. This paper describes a novel scheme that employs model-based techniques to integrate a family of publisher/subscriber services within QoS-enabled CORBA component middleware.

Our previous work on publisher/subscriber architectures focused on the patterns and performance optimizations of event channels in the context of real-time object middleware [4], specifically a *highly scalable* [5] and *real-time* [6] CORBA Event Service [7]. This paper extends our previous work by describing how patterns and Model-Driven Middleware (MDM) [8] can be applied to simplify the integration of publisher/subscriber services in QoS-enabled component middleware. We have developed a MDM tool-suite named *Component Synthesis with Model-Integrated Computing* (CoSMIC) [9, 10] that employs our QoS-enabled CCM implementation, the *Component-Integrated ACE ORB* (CIAO). The CoSMIC toolsuite supports the development, assembly, and deployment of QoS-enabled component applications. This paper focuses on the *Event QoS Aspect Language* (EQAL), which is the CoSMIC tool that supports the specification and configuration of publisher/subscriber services.

The remainder of this paper is organized as follows: Section 2 details the key challenges associated with implementing and configuring publisher/subscriber services in component middleware; Section 3 describes a pattern-oriented component middleware framework and a complementary MDM tool we developed to address these challenges; and Section 4 presents concluding remarks.

2 Publisher/Subscriber Service Integration Challenges in Component Middleware

This section describes the inherent R&D challenges that must be overcome in order to utilize QoS-enabled publisher/subscriber services for component-based application development. For each challenge, we describe the context in which the challenge arises, identify the specific problem that must be addressed, and outline an approach to resolving the challenge. Section 3 then illustrates how we applied those solution approaches in CIAO and CoSMIC.

2.1 Challenge 1: Providing Publisher/Subscriber Service Access via the Container Programming Model

Context. The *container programming model* establishes a paradigm for component interaction with a runtime execution environment. Specifically, the container programming model designates a software entity, known as a *container*, to manage a set of components. The container supports an API framework through which developers control component lifecycles and access common middleware services, including publisher/subscriber services. The container architecture decouples components from application server implementations, thereby enhancing flexibility and reuse.

A CCM container provides component implementations with access to common CORBA services, including (but by no means limited to) two distinct publisher/subscriber services: the Event Service and the Notification Service. CIAO supports both these services, as well as an extended version of the Event Service, the Real-Time Event Service. Each publisher/subscriber service that CIAO supports has different capabilities and is accessed via a distinct interface. Figure 1 illustrates the different aspects of QoS-enabled publisher/subscriber service support within the CIAO container framework.

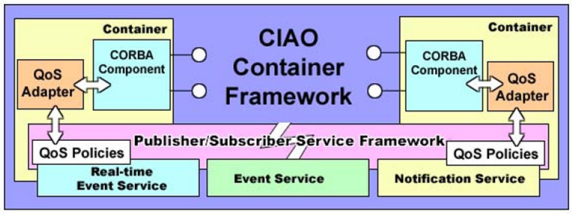

Figure 1: **CIAO Publisher/Subscriber Service Framework**

Problem. The standardized CCM container interface is designed to encapsulate only a subset of the Notification Service, rendering the broader range of CORBA-based publisher/subscriber services inaccessible. The container interface is generic by design, simplifying its use and enabling component interoperability among CCM implementations. Its design, however, prevents access to any advanced publisher/subscriber service capabilities, such as QoS guarantees. For example, the standard container interface lacks any mechanism to specify priorities, timeouts, or event correlation. In the case of DRE applications, such capabilities are essential for effective system operation. At the same time, exposing components to dissimilar and proprietary publisher/subscriber services requires developers to manipulate an overly complicated and confusing range of interfaces, prevents compliance with relevant OMG standards, and bloats component memory footprint.

Solution approach → Enhance containers to encapsulate, implement, and configure publisher/subscriber services. CIAO's container framework presents application components with a uniform interface (shown as the QoS adapter in Figure 1) through which various publisher/subscriber services can be selected and configured. This interface enables application components to use any combination of the publisher/subscriber services supported by CIAO. More importantly, the container framework supports QoS configuration of publisher/subscriber services (*e.g.*, assignment of priorities and latency thresholds), without exposing components to service implementation details by encapsulating service-specific QoS specification operations within a high-level interface. Consequently, components can invoke these services in a straightforward manner (*i.e.*, without becoming tightly coupled to low-level CORBA programming details), while preserving the flexibility and customizability of the underlying services.

Section 3.1 describes how we applied patterns to enhance CIAO's container framework to support a range of publisher/subscriber services.

2.2 Challenge 2: Configuring Publisher/Subscriber Quality-of-Service in Component Deployments

Context. Component-based DRE software deployments often require custom QoS configurations to target multiple OS, network, and hardware platforms, each of which may have slightly different requirements. In such cases, event QoS requirements (such as latency thresholds and priorities) may only be known when components are deployed, rather than when they are developed. Contemporary component middleware frameworks use XML descriptor files to specify the publisher/subscriber configurations and QoS constraints associated with particular software deployments. The component deployment mechanism is responsible for parsing these files and making the appropriate invocations on the publisher/subscriber configuration interface provided by the container.

Problem. It is tedious and error-prone to *manually* specify the QoS requirements of large-scale DRE component deployments. Component middleware has become complex to configure due to an increasing number of operating policies (such as transaction and security properties, persistence and lifecycle management, and publisher/subscriber QoS configurations) that exist at multiple middleware layers and employ legacy specification

mechanisms not based on XML. To further complicate matters, many combinations of policies are semantically invalid and will result in system failure.

Solution approach → Leverage modeling tools to configure and deploy components in a graphical, intuitive way. In the context of this paper, a modeling tool is a software package that allows users to construct graphical representations of complex concepts from primitive elements. To be useful, the tool must enforce syntactic rules among the modeling elements and specify the semantics of any valid model. In the context of component applications, modeling tools enable the creation of reusable component deployment models that are easier to build, understand, and maintain than manually written deployments. Moreover, modeling tools can automatically (1) generate XML descriptor files that describe component interactions and configuration files that specify middleware operating policies and (2) validate models to ensure consistency and coherency among component properties.

Section 3.2 describes the publisher/subscriber QoS configuration tool we developed as part of the CoSMIC tool-suite [9].

3 Pattern-Oriented and Model-Driven Middleware Solutions in CIAO and CoSMIC

The two challenges described in Section 2 require different solution approaches. To address the publisher/subscriber service access challenge discussed in Section 2.1, we employed pattern-oriented software [11, 3] techniques. To address the publisher/subscriber service configuration challenge described in Section 2.2, we employed Model Driven Middleware (MDM) techniques. This section describes our two solution approaches in detail.

3.1 Employing Patterns to Provide Publisher/Subscriber Service Access via the Container Programming Model

The CIAO publisher/subscriber service architecture, shown in Figure 1, employs patterns to address the design goals of the container programming model, outlined in Section 2.1. 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 CIAO container programming model, the pattern-oriented solution is detailed below.

Design goal 1 → Simplify component development by exposing a simple publisher/subscriber service interface. To achieve this design goal in CIAO, we used the *Adapter* pattern [11], which converts one interface into a different one expected by clients. Since the CORBA publisher/subscriber services were designed prior to CCM their interfaces are not ideal for components. The container therefore implements an adapter that provides components with a simple, uniform interface and translates calls on that interface into calls on a specific publisher/subscriber service interface. 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. To achieve this design goal, we used the *Strategy* pattern [11], which defines classes that encapsulate different algorithms and declares an interface common to all supported algorithms. In CIAO, a local CORBA interface serves as the common interface, and the various implementations encapsulate algorithms for the different publisher/subscriber services. 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. To achieve this design goal, we used the *Builder* pattern [11], which sep-

arates the construction of objects from their representation, allowing the same process to create multiple implementations of the same object type. The creation of publisher/subscriber service instances is somewhat complex in CIAO since (1) they must be initialized properly and (2) different implementations are possible. CIAO defines a builder class, or *factory*, that encapsulates the complexity of creating and initializing publisher/subscriber service implementations. 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 only incur the cost of services that are required by deferring publisher/subscriber service selection and configuration decisions until run-time. To achieve this design goal, we used the *Component Configurator* pattern [12], which allows an application to link and unlink its component implementations at run-time. 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. More generally, CIAO enables entire applications to be composed of independently developed services, thereby simplifying composition and deployment.

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. To achieve this design goal, we used the *Wrapper Facade* pattern [13], which defines a concise, robust, portable, and maintainable interface to encapsulate low-level functions and data structures. The CIAO publisher/subscriber framework implements CORBA interfaces that contain operations to configure QoS parameters for an individual publisher or subscriber connection. The operations of these interfaces forward 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.

3.2 Employing Model-Driven Design, Verification, and Synthesis to Configure QoS in Component Deployments

Section 3.1 describes how the CIAO container framework supports access to publisher/subscriber services. Specifying service configurations textually is unnecessarily complex, however. We therefore describe the *Event QoS Aspect Language* (EQAL), which is the CoSMIC MDM tool for *graphically* specifying publisher/subscriber service configurations. EQAL consists of two complimentary entities:

- A *meta-model* that defines a modeling language, or *paradigm*. The meta-model specifies the types of modeling elements, their properties, and relationships.
- *Model interpreters* that synthesize middleware configuration files from models. The EQAL model interpreters automatically generate publisher/subscriber service configuration files and component property description files.

The EQAL meta-model defines a modeling paradigm for the Generic Modeling Environment (GME) [14]. Within this paradigm, a modeler specifies the desired publisher/subscriber service (*i.e.*, event service, notification service, or real-time event service) and the configuration of that service for each component event connection. EQAL model interpreters can be executed to validate the models and synthesize text-based configuration files for a given component assembly. Component deployers can therefore build publisher/subscriber service configurations for component applications using the EQAL modeling paradigm and associated model interpreters. Below we describe each part of EQAL shown in Figure 2.

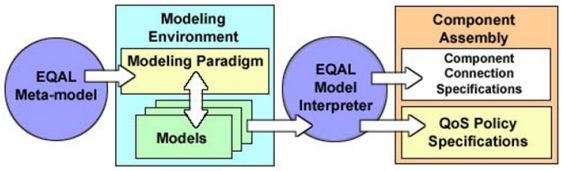

Figure 2: **The Event QoS Aspect Language Architecture**

Meta-model. A meta-model comprises a concrete manifestation of a modeling paradigm, *i.e.*, the syntactic, semantic, and presentation abstractions defined by a domain-specific modeling language [15]. The EQAL meta-model defines a modeling paradigm for publisher/subscriber service configuration models, which specify QoS configurations, parameters, and constraints. For example, the EQAL meta-model contains a distinct set of modeling constructs for each publisher/subscriber service supported by CIAO. Policies and strategies that can be modeled include (but are not limited to) filtering, correlation, timeouts, locking, disconnect control, and priority.

Publisher/subscriber service policies have differing scope, from a single port to an entire event channel. The EQAL meta-model allows modelers to provision reusable and sharable configurations at each level of granularity. A modeler assigns configurations to individual event connections and then constructs filters for each connection. Two forms of event generation using the push model are supported, *i.e.*, a component may be an exclusive supplier of an event type or a component may supply events to a shared channel.

Model constraints. Dependencies among publisher/subscriber QoS policies, strategies, and configurations are complex. Ensuring coherency among policies and configurations has been a major source of complexity in component middleware [16]. During the modeling phase, EQAL ensures that dependencies between configuration parameters are enforced by declaring constraints on the contexts in which individual options are valid. EQAL can then automatically verify the legitimacy of configurations and notify the user during modeling time of incompatible QoS properties. For example, priority-based thread allocation policies are only valid with component event connections that have assigned priorities.

Model interpreter. EQAL synthesizes multiple configuration files. A component deployment framework parses these files, creates event channels, and configures each connection, while shielding the actual component implementations from the lower-level middleware service details. A single component deployment may encompass any number of EQAL models and may require a large number of files to be synthesized. However, the EQAL interpreters can generate all the necessary configuration specifications with a single user command. Based on the root model specified for interpretation, EQAL generates as many descriptors as required, making multiple passes through the model hierarchy if necessary. These files must currently be written by hand, which is time-consuming and error-prone. At the same time, component deployment platforms are highly intolerant of syntactical and semantic errors, making debugging difficult and tedious. Accordingly, the automation of this process - and the associated assurance of model validity - improves the reusability of components across diverse deployment scenarios.

EQAL incorporates two distinct model interpreters. The first interpreter generates XML descriptor files that are used by a component deployment framework. This interpreter relies on all types of EQAL models during synthesis, and thus the modeler must have completely specified the component event configuration - from event channels down to individual ports - for this interpreter to function. The second interpreter generates service configuration `svc.conf` files for The ACE ORB (TAO) [17], which is the underlying Object Request Broker (ORB) used by CIAO. The `svc.conf` files are scripts that specify event channel policies and strategies. This interpreter only requires that event channel configurations have been modeled. Both file types - XML descriptors and TAO `svc.conf` files - have well-defined structure and syntax.

• **XML descriptor synthesis.** XML descriptors are the standard way to describe component deployments in CCM. These files specify how components are connected together, on what hosts they will run, how the components are instantiated, among numerous other properties. The XML descriptors synthesized by the EQAL interpreter indicate the QoS requirements of individual component event connections. Since the CCM specification does not explicitly address the mechanisms for ensuring component QoS properties, the EQAL-generated descriptors are based on official schema developed at Boeing for their extension mechanisms built into CCM, however, the descriptors remain compliant with the CCM specifications. We use Bold Stroke schema because it has been carefully crafted, refined, tested, and optimized in the context of real-world DRE avionics mission computing systems.

• **Service configuration file generation.** The interpreter that generates event channel `svc.conf` files is more straightforward than the XML descriptor generator. Although there are a large number of policies that result in many modeling element sub-types, the majority of the complexity in resolving event channel configurations is accomplished by the EQAL modeling constraints. Dependencies and interactions between event channel policies are complex, making handcrafting these files hard, but once a valid set of parameters is guaranteed, this EQAL interpreter need only discover the artifacts contained within a configuration and write the appropriate parameters to a file.

4 Concluding Remarks

The integration of QoS-enabled publisher/subscriber services into component middleware allows developers of DRE systems to leverage the benefits of component-centric software development. This paper described how QoS-enabled publisher/subscriber services can be integrated into a component middleware container framework using patterns and how such services can be configured using modeling tools. The integration techniques we described allow DRE system developers to utilize the full extent of publisher/subscriber service capabilities without becoming tightly-coupled to service implementations. The modeling techniques described allow DRE system deployers to rapidly create, validate, and synthesize component publisher/subscriber QoS configurations. Finally, this paper demonstrated each integration strategy in CIAO and CosMIC. The CIAO and CosMIC distributions are available for download at `www.dre.vanderbilt.edu/CIAO/` and `www.dre.vanderbilt.edu/cosmic/`, respectively.

References

[1] Richard E. Schantz and Douglas C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering*, John Marciniak and George Telecki, Eds. Wiley & Sons, New York, 2002.

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

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

[4] Douglas C. Schmidt and et al., "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, Feb. 2002.

[5] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell 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, no. 2, Mar. 2002.

[6] 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.

[7] Object Management Group, *Event Service Specification Version 1.1*, OMG Document formal/01-03-01 edition, Mar. 2001.

[8] Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang, "Model Driven Middleware," in *Middleware for Communications*, Qusay Mahmoud, Ed. Wiley and Sons, New York, 2003.

[9] Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind Krishna, George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt, "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications," *Submitted to The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2004.

[10] Aniruddha Gokhale, "Component Synthesis using Model Integrated Computing," `www.dre.vanderbilt.edu/cosmic`, 2003.

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

[12] Douglas C. Schmidt and Stephen D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, Reading, Massachusetts, 2002.

[13] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*, Wiley & Sons, New York, 2000.

[14] Akos Ledeczi, Arpad Bakay, Miklos Maroti, Peter Volgysei, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, Nov. 2001.

[15] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty, "Model-Integrated Development of Embedded Software," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003.

[16] Douglas C. Schmidt and Chris Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, no. 4, Apr. 1999.

[17] 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.