Improving Real-Time System Configuration via a QoS-aware CORBA Component Model*

Nanbor Wang and Christopher Gill
Department of Computer Science and Engineering
Washington University, St.Louis,MO
{nanbor,cdgill}@cse.wustl.edu

Abstract

A fundamental tension exists in systems research and practice between (1) hiding extraneous details to simplify the system programming model, and (2) revealing crucial details to allow better customization to particular application requirements, data characteristics, or other constraints. For many modern systems, such as distributed real-time and embedded (DRE) systems and multimedia applications, this tension is even stronger: the correctness of the system depends not only on functional behavior (e.g., correctness of interfaces and the algorithms behind them) but also on quality of service (QoS) properties such as CPU speeds, frame rates, and end-to-end timeliness requirements. Satisfying both the functional and QoS requirements of modern applications running in various run-time environments requires programming models that can allow selective visibility and (re)configurability of key details in both the application and the services it uses.

Although prior research and practice has focused on decoupling system properties from the application logic to facilitate modification at different points in the system lifecycle, some system properties cross-cut both (1) multiple layers (application and supporting infrastructure) and (2) multiple configuration phases in the system lifecycle. To date, there has been little support for configuring such cross-cutting concerns in a systematic and standardized way across different kinds of applications. This has resulted historically in laborious and ad hoc hand-customization of systems to meet real-world constraints, especially for DRE systems.

However, the convergence of model-integrated computing with component-oriented middleware in the DRE middleware research community offers a new opportunity for automated coordination of system modifications at different layers and at different points in the system lifecycle, to meet application requirements end-to-end. We are extending the Component-Integrated ACE ORB (CIAO), to provide a unified system configuration capability that can be leveraged by higher-level modeling tools.

This paper provides two main contributions to the state of the art in component middleware for model integrated computing in DRE systems. First, we present an argument based on a representative real-world example from the avionics mission computing domain, that multiple points of coordination in the CORBA Component Model (CCM), i.e., component packaging, assembly and deployment, are needed to assure feasibility and increase performance at run-time. Second, we summarize key lessons learned for configuring QoS properties such as event dependencies, rates of execution, and feasibility of deadlines, and describe which of the alternative configuration points in CCM are most appropriate in our example under different design considerations.

Keywords: Component-Based Software, QoS-aware Component Models, Distributed Real-Time Embedded Middleware.

1 Introduction

The Component-Integrated ACE ORB (CIAO) [1] is a QoS-aware CORBA Component Model (CCM) based implementation that we are developing at the Center for Distributed Object Computing in the Department of Computer Science and Engineering at Washington University in St. Louis, in collaboration with researchers at ISIS, Vanderbilt University. This section is structured as follows. In Section 1.1, we first describe a simple example from the avionics mission computing [2] domain that motivates our work on a QoS-aware component model. We then examine the key challenges that arise from that example in Section 1.2. Those challenges form the basis by which we compare CIAO to previous middleware approaches.

*This work was supported by DARPA PCES contract F33615-01-C-3048, under subcontract to Boeing titled "CORBA Component Model for Real-Time Embedded Applications". We gratefully acknowledge the support and guidance of the Boeing PCES Principal Investigator Mr. David Sharp, and DARPA PCES Program managers Dr. Douglas C. Schmidt, Dr. John Bay, and Dr. Joe Cross.

1.1 Motivation

We have been working with researchers and engineers at the Boeing company to examine how QoS-aware component computing capabilities provided by CIAO can benefit both development and performance of avionics mission computing systems. The Boeing Bold Stroke framework [2] allows avionics mission computing applications to be built from separately developed components and provides domain-specific services to the resulting application. Boldstroke itself is built atop commercial-off-the-shelf (COTS) middleware, operating systems, and hardware. Its use of the The ACE ORB (TAO) [3] in particular makes it a natural focus of our work in CIAO.

In addition to the software engineering goals of reducing development costs and cycle times, we have focused on how CIAO can manage configuration decisions *at different points in the software development lifecycle* to improve system feasibility and performance. This paper examines these performance benefits, and describes in detail how CIAO can achieve them. We present a simple but representative example of an interaction between three representative components in Section 3, show how two problems of feasibility and optimization can arise, and describe how CIAO capabilities can help address those problems. While a complete avionics mission computing system would naturally have an order of magnitude greater number of components, the example presented in this paper (1) is representative of both the problems and solution techniques, and (2) is expected to have greater rather than less benefit as the complexity of the system increases.

1.2 Key Challenges

The example in Section 1.1 gives rise to the following challenges for developing DRE systems middleware:

Component-level Constraints: Individual components may have constraints that are a function of (1) the algorithm they perform, (2) a hardware, operating system, or middleware abstraction they encapsulate, or (3) quality requirements on the system in which they are used. For example a component wrapping an actuator may impose a minimum time between commands it can send to the device.

Unforeseen Combinations: Components are generally intended for re-use, and therefore it is common that they may be assembled in combinations that were not considered at the time each component was developed. This is of particular interest as components are used across *families* of applications, where the assumptions about the application requirements, the set of components used, and how the components are interconnected may vary significantly.

Interdependencies at Assembly: When components are assembled into applications, interdependencies between components are established. For example, a component that uses a result computed by another component would naturally receive a method call or event from the other component, either carrying the result itself or as a trigger to retrieve the result from elsewhere in the system. Interestingly, properties of a component may be affected by the interconnections made to other components, *e.g.*, the rate at which method calls or event pushes are made to it.

Resource Allocation: Some systems have very static resource allocation policies, often by design. In particular, the more stringent the requirements on assurance, verification, and certification of properties such as end-to-end timeliness, the more likely those decisions will be made statically. However, even in those kinds of systems, issues such as hardware upgrades and portability across families of related systems result in resource (re)allocation decisions that occur well after the original system was designed. For such static systems, it is more a matter of economic and opportunity cost due to engineering effort and long development cycles, but it is expensive nonetheless. For an increasingly wide class of systems that allow dynamic reallocation of resources to cope with varying operating environments, this issue is also directly related to correctness of system QoS properties under adaptation.

Early Binding of Decisions: Clearly, the previous challenges show that multiple constraints from different points in the system lifecycle result in a complex and interlocking set of decisions that must be made in each of several stages. Making some decisions as early as possible may simplify later decisions. If a component has inherent constraints, then expressing those constraints when the component itself is created allows those decisions to be assumed complete unless later conflicts with them arise. For example, expressing the inherent minimum inter-arrival constraint on a device wrapped by a component is a stable decision, as long as the device and the component are not used separately.

Preserving Flexibility: There is a direct tension between the wish to bind decisions early to reduce complexity of later decisions, and the benefits of preserving flexibility in decisions that may benefit from exploitation of later information. The key is to identify information that is fundamentally stable at each stage of the system lifecycle, and make decisions that only rely on stable information at that point. If it is unclear whether information is stable, then approximate or interim decisions should be made, but should be left flexible to be rebound if later better information becomes available. For example the transmission time to send data between two components may be known for a given communication link, but should be parameterized with additional information to rebind the transmission time if another link were used.

1.3 Paper Organization

This paper is structured as follows. Section 2 surveys QoS-aware component middleware, and contrasts it to previous middleware approaches. Section 3 describes how several QoS-configuration phases can be applied in CIAO to achieve verifiable improvements in performance for the example application described in Section 1.1. Section 4 summarizes related work, and Section 5 presents conclusions.

2 The Case for QoS-aware Component Middleware

This section examines previous approaches to distributed object computing (DOC) middleware, presents an overview of component middleware, *i.e.*, the CORBA Component Model, and explains why conventional component middleware fails to support the key QoS challenges described in Section 1.2. We then illustrate how QoS-aware component middleware such as CIAO expands the ability of middleware to facilitate QoS provisioning and enforcement for DRE applications. Section 2.1 gives an overview of conventional DOC middleware, and describes the limitations of that approach that gave rise to development of component middleware approaches. Section 2.2 then describes conventional component middleware and lists limitations of *that* approach, which led to our work on QoS-aware component middleware. Finally, Section 2.3 gives an overview of the CIAO project and presents key features of CIAO that we apply as Section 3 describes, to address the challenges described in Section 1.2.

2.1 Conventional DOC Middleware

Middleware's primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure, to coordinate how parts of applications are connected and how they inter-operate. For distributed object computing (DOC) systems, a first step toward reuse was standardization of middleware interfaces and request broker semantics for method invocations on remote objects.

Overview: Conventional DOC middleware, such as the Object Management Architecture (OMA) in the CORBA 2.x specification [4], shields application developers from low-level platform details, provides standard higher-level interfaces to manage system resources, and amortizes development costs through reusable frameworks. CORBA 2.x focuses on *interfaces*, which are essentially contracts between clients and servers that define how clients *view* and *access* object services provided by a server.

Limitations: However, the CORBA 2.x specification has the following limitations [5]:

- **Lack of functional boundaries:** The CORBA 2.x object model treats all interfaces as client/server contracts. To build complex distributed applications, therefore, application developers must explicitly program the connections among interdependent services and object interfaces, which can yield non-reusable implementations.
- **Lack of generic server standards:** CORBA 2.x does not standardize a generic server framework to perform common server configuration work. The lack of such a standard yields tightly coupled, *ad-hoc* server implementations, which reduce the reusability and flexibility of CORBA-based applications.
- **Lack of software packaging and deployment standards:** There is no standard way to distribute and start up implementations remotely in CORBA 2.x. The lack of higher-level software management standards yields systems that are hard to maintain and implementations that are hard to reuse.

2.2 Conventional Component Middleware

The second role of middleware is to standardize the context within which objects are managed, and the interfaces for managing them. *Component middleware* [6] enables reusable services to be composed, configured, and installed to create applications rapidly and robustly.

Overview: Component middleware has evolved to address the limitations of conventional DOC middleware by

- creating a virtual boundary around application component implementations that interact with each other only through well-defined interfaces,
- defining standard mechanisms needed to configure and execute components in generic component servers, and
- specifying the infrastructure needed to assemble, package, and deploy components.

The CORBA Component Model (CCM) [7] addresses limitations with earlier generations of DOC middleware by extending the CORBA object model to support the concept of components and establishes *standards* for implementing, assembling, packaging, and deploying component implementations. To a client, a CCM component is an extended CORBA object that encapsulates various interaction models via interfaces and connection operations called *ports*. CCM includes the following ports shown in Figure 1:

- **Facets**, which define named interfaces that service method invocations from other components synchronously;
- **Receptacles**, which provide named connection points to synchronous facets provided by other components;

- **Event sources/sinks**, which indicate a willingness to exchange event messages with other components asynchronously.

Components can also have *attributes* that specify named parameters that can be configured via metadata specified in component property files. A *home* interface implements a lifecycle strategy for a given type of component.

From a server perspective, components are implementation entities that can be installed and instantiated independently in standard component server runtime environments stipulated by the CCM specification. Figure 1 shows the server-side runtime architecture of the CCM model.

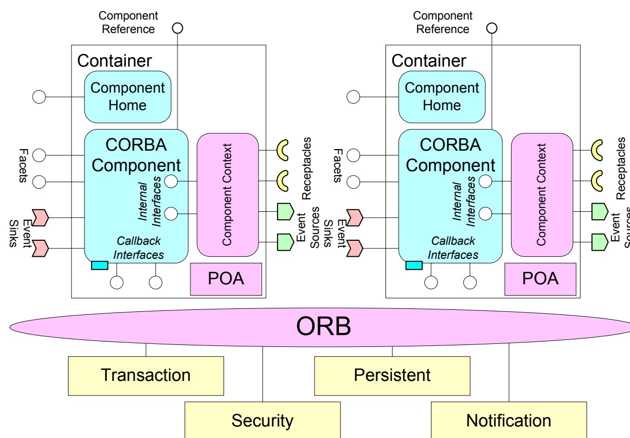

Figure 1: **Overview of the CCM Components and Runtime Architecture**

A *container* provides the server runtime environment for component implementations called *executors*. It contains various pre-defined hooks and operations that give components access to strategies and services such as persistence, event notification, transaction, and security. Each container defines a collection of runtime strategies and policies, such as event delivery strategies and component usage categories, and is responsible for initializing and providing runtime contexts for the managed components. Component implementations have associated metadata, written in XML, which specify the required container strategies and policies.

In addition to the building blocks outlined above, the CCM specification also standardizes component implementation, packaging, assembly, and deployment. The CCM Component Implementation Framework (CIF) automatically generates component implementation skeletons and support for persistent state management using the Component Implementation Definition Language (CIDL). XML-based metadata also describes component compositions, including component locations and interconnections between components, needed to form an assembled application. Deployment tools then use the component assemblies and composition metadata to deploy and initialize applications.

The tools and mechanisms defined by the CCM collaborate to address the limitations of conventional component middleware. The CCM programming paradigm separates concerns of composing and provisioning reusable software components into the following development roles in the application lifecycle:

- **Component designers** specify component features by defining how components collaborate with each other and with their clients by in terms of the interfaces and events each component offers and/or requires.
- **Component implementors** develop component implementations and specify the runtime support each requires via metadata in *component descriptors*.
- **Component packagers** bundle component implementations with their default properties and component descriptors together into *packages*.
- **Component assemblers** select component implementations, specify component instantiation constraints, and connect ports of component instances via specification in metadata called *assembly descriptors*, to configure an application.
- **System deployers** take assembly descriptors, analyze their runtime resource requirements, and prepare and deploy required resources for the assembled application.

Limitations: Large-scale systems require complicated provisioning, where developers must connect numerous distributed or collocated subsystems together and define the functionality of each subsystem. Component middleware can reduce the software development effort for these types of large-scale DRE systems by enabling application development through composition. Conventional component middleware, however, is designed more for the needs of business applications, rather than for the more complex QoS provisioning needs of DRE applications. Developers are therefore often forced to configure and control these mechanisms imperatively in their component implementations.

It is possible for component developers to take advantage of middleware or OS features to implement QoS-enabled components by embedding QoS provisioning code within a component implementation. Many QoS capabilities, however, cannot be implemented solely within a component due to the following limitations:

- QoS provisioning must be done end-to-end, *i.e.*, it needs to be applied to many interacting components. Implementing QoS provisioning logic internally in each component hampers reusability.
- Some resources, such as thread pools in Real-time CORBA 1.0 [8], can only be provisioned within a broader execution unit, *i.e.*, a component server rather than a component. Since component developers often have no *a priori* knowledge about other components,

the component itself is not the right place to provision QoS.

- Some QoS assurance mechanisms, such as checking whether rates of interactions between components violate specified constraints, affect component interconnections. Since a reusable component implementation may not know how it will be composed with other components, it is not generally possible for components to perform such QoS assurance in isolation.
- Many QoS provisioning policies and mechanisms require installation of customized ORB modules to work correctly. However, some requirements such as high throughput and low latency, may involve inherent trade-offs. It is hard for QoS provisioning mechanisms implemented within components to foresee incompatibilities without knowing the end-to-end QoS requirements *a priori*.

In general, forcing QoS provisioning functionality into a component prematurely commits each implementation to a specific QoS provisioning scenario. This tight coupling defeats one of the key benefits of component middleware: *separating component functionality from system management*. By creating dependencies between application components and the underlying component framework, component implementations become hard to reuse, particularly for large-scale DRE systems whose components and applications possess stringent QoS requirements. It is thus necessary to treat QoS configuration as a fundamental part of the component middleware model itself, as we describe next.

2.3 QoS-Aware Component Middleware

Large-scale DRE applications require seamless integration of many hardware and software systems. Figure 2 illustrates an air traffic control system that collects and processes real-time flight status from multiple regional radars across a country. Based on the real-time flight data, the system then reschedules flights, issues air traffic control commands to airplanes in flight, notifies airports, and updates the displays in an airport's flight bulletin boards. Model-based tools offer a high level of abstraction, and can simplify the development of large-scale DRE applications by integrating with component-based middleware at the following points [9], as shown in Figure 2:

1. Configuring/deploying an application end-to-end – for example, integrating functions such as navigation located within a single aircraft's cockpit information system, with interfaces to air traffic control and flight scheduling systems on the ground;
2. Composing components into an application server – for example, connecting the components *within* the cockpit information;

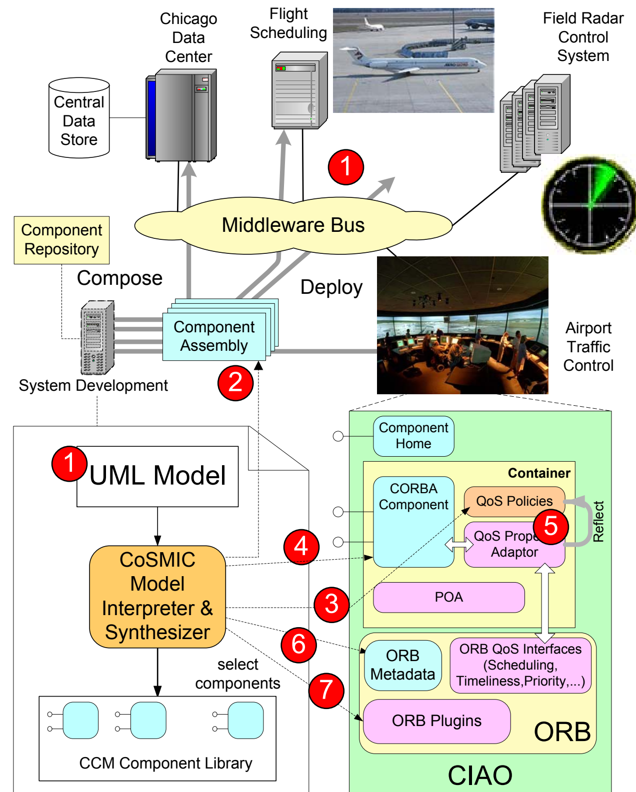

Figure 2: **Integrating DRE Applications with Component Middleware**

3. Configuring application component containers – for example connecting a hardware or operating system timer to a rate generation component;
4. Synthesizing component implementations – for example, generating code for a navigation algorithm from a modeling tool;
5. Synthesizing QoS provisioning and adaptation logic [10] – for example to ensure feasible allocations of resources;
6. Synthesizing middleware-specific configurations – for example configuring middleware thread pools;
7. Synthesizing middleware implementation – for example, generating QoS adaptation mechanisms, again from a modeling tool.

In traditional DRE systems, code for provisioning and enforcing QoS properties is often spread throughout the software and tangled with the application logic. This tangling makes DRE applications unnecessarily hard to maintain or to extend with new QoS mechanisms and behaviors. As we discussed in Section 2.1, as DRE systems grow in scope and criticality a key challenge is to decouple the reusable resource management aspects of the middleware from aspects that need customization and tailoring to the specific requirements of the application.

Static QoS provisioning involves pre-determining the resources needed to satisfy certain QoS requirements, and allocating the resources of a system before or during start-up time. Certain DRE applications use static provisioning because they (1) have a fixed set of QoS demands and (2) require tightly-bounded predictability for system functionality. For example, the commands for control surfaces in avionic systems should be assured access to resources *e.g.*, through planned scheduling of those operations or assigning them the highest priority [11]. In contrast, the handling of secondary functions, such as flight path calculation, can be delayed somewhat without significant impact on overall system functioning. In addition, static QoS provisioning is often the simplest solution available, *e.g.*, a video streaming application for the unmanned aerial vehicle (UAV) may simply choose to reserve a fixed network bandwidth for the audio and video streams [12].

To address the limitations with previous-generation middleware outlined in Sections 2.1 and 2.2, it is necessary to make QoS provisioning specifications an integral part of component middleware and apply aspect-oriented techniques [13] to decouple QoS provisioning specifications from component functionality. This separation of concerns relieves component developers from tangling the code to manage QoS resources with the component implementation. It also simplifies QoS provisioning that cross-cuts multiple interacting components end-to-end.

To perform robust QoS provisioning end-to-end throughout a component middleware system, QoS provisioning specifications should be decoupled from component implementations and specified instead in metadata associated with various application development lifecycles. This separation of concerns helps improve component reusability by preventing premature commitment to specific QoS provisioning parameters. QoS provisioning specifications can affect different scopes of components and their behaviors, *e.g.*, thread-pools are shared among multiple components, whereas a priority level can be assigned to a single component instance. Since different stages of a component software development lifecycle also bind design decisions that then affect certain scopes of an application, it is important to ensure that we are not binding QoS provisioning decisions too early to avoid adding unnecessary constraints in the development stage. We identify key component software development lifecycle stages and QoS specifications appropriate for these stages as follows:

- QoS constraints that are part of a component implementation need to be specified in the metadata associated with the component implementation, *i.e.*, component descriptors. Examples include (1) limitations on rates of invocation or (2) default execution times based on an *specified* CPU speed.
- Configurable specifications of a component, such as its priority level, can be assigned default values in the component package by associating a component implementation with a component property file as was described in Section 2.2.
- Resources such as thread-pools and prioritized communication channels that need to be shared among multiple components in a component server should be allocated separately as *logical* resource policies. Component instances within the same component server can then share these logical resources by associating with common logical resource policies declared in assembly metadata.
- Component assembly metadata must also be extended to provision QoS resources for component interconnections, such as defining feasible rates of event pushes from suppliers to consumers in a DRE system.
- As described in Section 2.2, generic component servers provide the execution environment for components. To ensure a component server is configured with the mechanisms and resources needed to support the provisioned QoS requirements, deployment tools should be extended to include middleware modules that can configure the component servers. Examples include customized communication mechanisms and custom priority mappings.

A detailed discussion of the design and implementation of CIAO is outside the scope of this paper, but documentation of those details is available [1]. Here we consider instead how a subset of those features can be *applied* to configure both functional and QoS properties in DRE systems.

3 Exploiting Configuration Phases

We now examine a simple but representative example of component QoS configuration, drawn from the motivating avionics mission computing domain described in Section 1.1, but also illustrating a part of the larger example shown in Figure 2 in Section 2.2.

In Section 3.1 we first examine the details of (1) the components themselves, (2) how they are assembled into an application, and (3) how they can be deployed feasibly and effectively. We focus on the *constraints* on decisions at each stage, and in particular how the constraints cross-cut multiple phases. In Section 3.2 we then consider how the lessons learned from this example relate to the overall relationship between CIAO and conventional CCM, and the decision lifecycles in each.

3.1 Example Application Revisited

Figure 3 illustrates a prototypical DRE application scenario which involves three components:

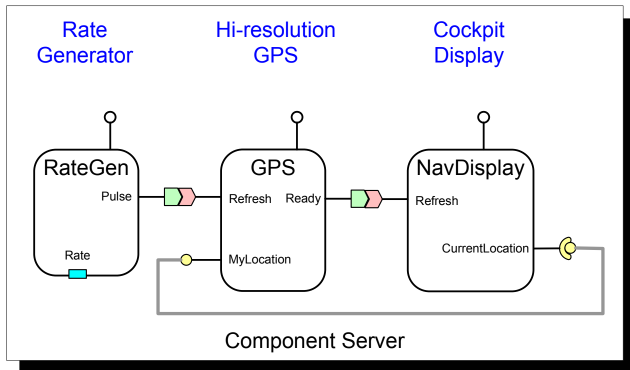

Figure 3: **A prototypical DRE application scenario.**

1. a **Rate Generator** component, which wraps a hardware timer and pushes events at specific periodic rates to components that register for those events,
2. a hypothetical **High-Resolution GPS** component, which wraps one or more hardware devices for navigation, and
3. a **Cockpit Display**, which wraps an output device in the cockpit which is visible to the pilot.

This simple example represents a broader class of cockpit information systems to which our work on avionics mission computing systems belongs, as does the cockpit information system within each aircraft in the larger system-of-systems example shown in Figure 2 in Section 2.3. Furthermore, although details the functional properties may differ, many DRE systems share the kinds of rate-activated computation and display/output QoS constraints illustrated here.

We now examine the information available and decisions made for this example at each of the three component lifecycle stages: packaging, assembly, and deployment.

Component Packaging: We start with the semantics of the components and their implementations, which are also called *executors* as Section 2.2 describes. For convenience of discussion, we label the components C_1 (Rate Generator), C_2 (High Resolution GPS), and C_3 (Cockpit Display).

Component C_1 has a *Rate* attribute that can be set to define the frequency at which it sends *Pulse* events out its event source port. Component C_2 has an event sink port called *Refresh* that when triggered causes (1) an internal state update from its underlying hardware device(s) to be performed and then (2) an event to be sent on its *Ready* event source port. It also provides a facet port called *MyLocation* at which the current GPS location value can be queried at any time (possibly subject to synchronization with the update step for data consistency). Finally, Component C_3 has an event sink called *Refresh* that when triggered causes a call to be made on a receptacle called *CurrentLocation*.

From a functional perspective, the conventional CCM type system enforces that each port is supported by the corresponding executor in each component. From a QoS perspective we found it useful to introduce an additional check, for a worst case execution time for each component port invocation. We label the relevant execution times for our example: W_1^{P1} (C_1 *Pulse*), W_2^{P1} (C_2 *Refresh*), W_3^{P1} (C_2 *Ready*), W_4^{P1} (C_2 *Mylocation*), W_5^{P1} (C_3 *Refresh*), and W_6^{P1} (C_3 *CurrentLocation*). Note the superscript *P1* on each value, designating the kind of processor on which they were calibrated. For purposes of simplicity in this example we will assume that all components run on the same kind of processor. While the choice of processor is normally a deployment decision, the ability of each component to advertise this information allows early checking of feasibility in cases such as versions of a single product line in which the processor does not change.

Finally, there may be component-level constraints on the rates at which particular ports can be used. We first label the port rates in our example: R_1 (C_1 *Pulse*), R_2 (C_2 *Refresh*), R_3 (C_2 *Ready*), R_4 (C_2 *Mylocation*), R_5 (C_3 *Refresh*), and R_6 (C_3 *CurrentLocation*).

For component C_1, we assume for the sake of discussion that the component can only generate events at one of several harmonic rates specified as an overall design constraint on the system. Such a design constraint might arise from hardware details, or from engineering objectives such as greater resource utilization under Rate Monotonic Analysis (RMA) [14]. We label these possible rates R_1^1, R_1^2, R_1^3, and R_1^4. The functional semantics of component C_2 implies that R_2 equals R_3. Furthermore, we assume an upper limit on the rate at which the C_2 *Ready* event can be produced, hypothetically due to the temporal complexity of the internal state update, which we label R_3^{max}. Finally, the functional semantics of component C_3 implies that R_5 equals R_6, and for the sake of discussion we assume that R_6 has a minimum acceptable rate for display refresh of $R_6^{display}$.

Application Assembly: When the components are assembled, the only new information is the dependences between components. From a functional perspective, each dependence requires checking of port type safety between the components it connects. From a QoS perspective, the dependences induce a requirement for "safety" of the *rates* of components, *i.e.*, they induce additional constraints with which component level constraints must be reconciled.

In particular, the connections shown in Figure 3 induce the requirements that R_1 and R_2 be equal, that R_3 and R_5 be equal, and that R_6 and R_4 be equal. With the previously stated rate constraints and the transitive property of equality, this results in the very strong constraint that *all rates in the application must be the same*. To simplify further discussion, we label this common rate R^{common}. For the sake of discussion, let us further assume that $R_1^1 < R_6^{display} < R_1^2 < R_1^3 < R_3^{max} < R_1^4$. Therefore, only R_1^2 and R_1^3 can satisfy the system of constraints, and that once one of them

Lifecycle Phase	Specified Attributes	Configuration Decision(s)	Benefits
Component Packaging	event sinks/sources$^+$ executors$^+$ **execution times**$^+$ **rate constraints**$^+$	Executor supports each port; Execution times calibrated	Functional type safety; Rehosting possible
Application Assembly	component dependences$^+$ **rate constraints***	Port types match; Source/sink rates match	Port type safety; Rate constraints satisfied
System Deployment	**rate constraints*** **CPU speed**$^+$	CPU use feasible; Utilization optimized	Assure timeliness; Better resource use

Table 1: Configuration Information at Each Lifecycle Phase

is chosen all components will run at that rate.

At this point, it is also possible to check feasibility of the application assuming Rate Monotonic Scheduling and that the components will in fact be hosted on processors of kind *P1*. It is then sufficient to test each remaining rate under RMA: $R^{common} * (W_1^{P1} + W_2^{P1} + W_3^{P1} + W_4^{P1} + W_5^{P1} + W_6^{P1}) < U_{P1}$, where U_{P1} is the feasible utilization bound on processor *P1*. For the sake of argument, let us further assume that at rate R_1^2 the system would be feasible but at rate R_1^3 it would be infeasible on processor *P1*.

System Deployment: One view of the assembly analysis above is that a clear design constraint has been established: the system should be deployed on processor *P1* with all components running at rate R_1^2. If both those criteria (the rate and processor choice) are suitable, then no further QoS checking is needed for deployment. However, if we assume availability of another faster processor *P2* on which the system would be feasible at rate R_1^3, then the design decision becomes ambivalent. Again for the sake of discussion, let us assume that the quality of the GPS solution will improve with a higher rate of execution and that this is a prominent requirement for the application. Then, we would deploy the application on processor *P2* with the components at rate R_1^3 where possible, or on processor *P1* with the components at rate R_1^2 otherwise.

3.2 Lifecycle Lessons Learned

Table 1 illustrates the attributes introduced or modified at each lifecycle phase in our example and summarizes the impact of each decision along the way. Attributes are marked with a $^+$ where they are introduced and a * where they are modified. Attributes solely involving QoS information are shown in bold, while functional attributes are shown in standard font though they too may indirectly impact QoS decisions. We draw three main lessons from this summary.

Bind decisions early if possible. Some information can be ignored after it is checked at a particular lifecycle phase. For example, after the component packaging phase *ensures* that each event sink has a corresponding executor, the application assembly and system deployment phases need not be concerned with that issue. This has the overall benefit of simplifying decisions made later in the system lifecycle.

(Re)bind decisions flexibly. Other information, and the results of previous decisions that have relied on it, cross-cut several phases of the system lifecycle. For example, after the sets of available event source and sink rates are ensured to match along component dependences in the assembly phase, the resulting sets of rates and the component dependences are still needed in the system deployment phase. This allows configuration of concerns that cross-cut the architectural boundaries of component, application, and system, as well as the configuration phase for each of these architectural levels.

QoS aspects tend to cross-cut functional boundaries. Notice especially that QoS information is often refined in subsequent lifecycle phases after it is introduced. Functional information, on the other hand, tends to be more fixed once it is specified in a given phase. This reflects a natural point of difference between CIAO and conventional CCM in which functional information tends to compose in a more object-oriented manner, while the "locality of reference" of QoS decisions tends to be organized around aspect modularity that cross-cuts object and even component boundaries. CIAO is designed with the necessary refinement of QoS aspects in mind, and the understanding that decisions can improve with additional information as long as prior decisions can be kept flexible and revisited as needed. Conventional CCM on the other hand is designed more for functional properties that once specified remain stable for all subsequent composition stages.

4 Related Work

Component Middleware: The architectural patterns used in CCM [15] are also used in other popular component middleware technologies, such as J2EE [16, 17] and .NET. Among the existing component middleware technologies, CCM is the most suitable for DRE applications since CORBA is the only standards-based COTS middleware that has made a substantial progress in satisfying the

QoS requirements of DRE systems. Although the CCM specification [7] has recently been finalized by the OMG, it still has not been fully incorporated into the core CORBA specification [18]. In addition to CIAO, a number of CCM implementations are available, including *OpenCCM* by the Universite des Sciences et Technologies de Lille, France [19], *K2 Containers* by iCMG [20], *MicoCCM* by FPX [21], and *Qedo* by Fokus [22].

Quality Objects: QuO [23, 24] is an adaptive middleware framework developed by BBN Technologies that allows the DRE developer to use aspect-oriented software development [13] techniques to separate the concerns of QoS programming from application logic in DRE applications. A *Qosket* is a unit of encapsulation and reuse for QuO systemic behaviors. In comparison to CIAO, Qoskets and QuO emphasize dynamic QoS provisioning where CIAO emphasizes static QoS provisioning and integration of various mechanisms and behaviors during different stages of the development lifecycle. We are collaborating with BBN to integrate Qoskets and CIAO [25] to provide a total QoS provisioning solution.

Reflective Middleware: In their *dynamicTAO* project, Kon and Campbell [26] apply reflective middleware techniques to extend TAO to reconfigure the ORB at runtime by dynamically linking selected modules, according to the features required by the applications. Their work falls into the same category as *Qoskets* in that both provide the mechanisms to realizing QoS provision during runtime.

QoS-aware Component Middleware: de Miguel's work on QoS-enabled containers extends a QoS EJB container interface to support `QoSContext` interface which allows exchange of QoS related information with component instances [27]. To take advantage of the QoS-container, however, a component must implement `QoSBean` and `QoSNegotiation` interfaces. This requirement adds an unnecessary dependency to component implementations, and as we discuss in Section 2.2 tying QoS behavior logic to component implementations has several drawbacks.

The QoS Enabled Distributed Objects (Qedo) project [28] is another ongoing effort to make QoS support an integral part of CCM. Qedo targets applications in the telecommunication domain and has strong support for information streaming. They define a meta-model which is able to define multiple category of QoS requirements for applications. To support the modeled QoS requirements, Qedo defines extensions to CCM's container interface and the Component Implementation Framework (CIF) to realize QoS models [29]. Qedo's extensions to container interfaces and the CIF also require component implementations to interact with the container QoS interface and negotiate the level of QoS contract directly. While this approach is suitable for certain applications where QoS is part of the functional requirements, it inevitably couples the QoS provisioning and adaptation behaviors into the component implementation and thus hampers the reusability of component implementations. In comparison, our approach explicitly avoids this coupling and tries instead to *compose* QoS provisioning behaviors within the component model itself.

Standards Efforts: The OMG has started several standardization efforts that match the goals we are trying to achieve in CIAO. The Light Weight CCM [30] specification aligns with CIAO's goal of reducing the footprint of CCM implementations to make CCM more suitable for DRE applications. This is achieved by removing features, such as persistent state management and transaction support, which are not commonly used in DRE applications, from the existing CCM specification. CIAO is the product of a research project that started before the Light Weight CCM effort and also aims to provide a CCM subset for DRE applications. We have proposed and implemented a similar subset of features to those in the Light Weight CCM submission. Future versions of CIAO will be amended to comform with the Light Weight CCM specification when standardized.

The Deployment and Configuration [31] submission aims to provide a model for deploying and configuring complex component systems. Similar to our approach, it also raises the question of available resources and uses the deployment and configuration stages of a component development lifecycle to ensure the needs of the applications are met. It is currently still in the standardization process and we will be able to apply the specification once it is formally adopted.

The UML Profiles for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms [32] also relate to CIAO's QoS-enabled CCM efforts. That specification is not, however, specifically aimed at component-based applications. Two other efforts that are also related to CIAO are the Quality of Service for CCM Request for Proposal (RFP) [33] and the Streams for CCM RFP [34].

Aspect-Oriented Programming: Aspect-Oriented programming (AOP) [13] provides language-level abstractions to weave different aspects that cross-cut multiple layers of a system. Examples of AOP tools include AspectJ [35] and AspectC++ [36]. Similar to AOP tools, CIAO supports injection of aspects into systems at the middleware level using meta-programming techniques. Both CIAO and AOP tools weave aspects statically, and neither defines abstractions for dynamic QoS provisioning.

5 Conclusions

In this paper we have presented arguments that QoS as well as functional attributes of a DRE application must be supported fully in the overall programming model. In Section 2

we made the case that the QoS metaprogramming capabilities we are building into CIAO are necessary to meet that goal, due to limitations of conventional middleware approaches. In Section 3 we presented a detailed analysis of a simple motivating example, and showed how management of QoS configuration decisions cross-cuts both functional and lifecycle boundaries. We believe that our experiences to date indicate a pivotal role for QoS-aware component middleware such as CIAO in configuring QoS properties of complex large-scale DRE applications, especially when combined with higher-level modeling and analysis tools.

References

[1] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill, "QoS-enabled Middleware," in *Middleware for Communications* (Q. Mahmoud, ed.), New York: Wiley and Sons, 2003.

[2] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[3] Center for Distributed Object Computing, "The ACE ORB (TAO)." www.cs.wustl.edu/∼schmidt/TAO.html, Washington University.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6.1 ed., May 2002.

[5] N. Wang, D. C. Schmidt, and C. O'Ryan, "An Overview of the CORBA Component Model," in *Component-Based Software Engineering* (G. Heineman and B. Councill, eds.), Reading, Massachusetts: Addison-Wesley, 2000.

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

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

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

[9] A. Gokhale, B. Natarajan, D. C. Schmidt, A. Nechypurenko, J. Gray, N. Wang, S. Neema, T. Bapty, and J. Parsons, "CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications," in *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, (Seattle, WA), ACM, Nov. 2002.

[10] V. Subramonian and C. Gill, "A Generative Programming Framework for Adaptive Middleware," in *Hawaii International Conference on System Sciences, Software Technology Track, Adaptive and Evolvable Software Systems Minitrack, HICSS 2003*, (Honolulu, HW), HICSS, Jan. 2003.

[11] C. Gill, D. C. Schmidt, and R. Cytron, "Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing," *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, vol. 91, Jan. 2003.

[12] R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali, "Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware," in *Proceedings of Middleware 2003, 4th International Conference on Distributed Systems Platforms*, (Rio de Janeiro, Brazil), IFIP/ACM/USENIX, June 2003.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[14] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, Jan. 1973.

[15] M. Volter, A. Schmid, and E. Wolff, *Server Component Patterns: Component Infrastructures Illustrated with EJB*. West Sussex, England: Wiley Series in Software Design Patterns, 2002.

[16] F. Marinescu and E. Roman, *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. New York: John Wiley & Sons, 2002.

[17] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

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

[19] F. Universite des Sciences et Technologies de Lille, "The openccm platform." http://corbaweb.lifl.fr/OpenCCM/, 2003.

[20] iCMG, "K2 component server." www.icmgworld.com, 2003.

[21] M. is CORBA, "The mico corba component project." http://www.fpx.de/MicoCCM/, 2000.

[22] Qedo, "Qos enabled distributed objects." http://qedo.berlios.de, 2002.

[23] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging Quality of Service Control Behaviors for Reuse," in *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Crystal City, VA), IEEE/IFIP, April/May 2002.

[24] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[25] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *The Journal of Microprocessors and Microsystems*, vol. 27, pp. 45–54, mar 2003.

[26] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications ACM*, vol. 45, pp. 33–38, June 2002.

[27] M. A. de Miguel, "QoS-Aware Component Frameworks," in *The 10th International Workshop on Quality of Service (IWQoS 2002)*, (Miami Beach, Florida), May 2002.

[28] FOKUS, "Qedo Project Homepage." http://qedo.berlios.de/.

[29] T. Ritter, M. Born, T. Unterschütz, and T. Weis, "A QoS Metamodel and its Realization in a CORBA Component Infrastructure," in *Proceedings of the 36th Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, (Honolulu, HW), HICSS, Jan. 2003.

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

[31] Object Management Group, *Deployment and Configuration Adopted Submission*, OMG Document ptc/03-07-08 ed., July 2003.

[32] Object Management Group, *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Joint Revised Submission*, OMG Document realtime/03-05-02 ed., May 2003.

[33] Object Management Group, *Qualify of Service for CORBA Component RFP*, OMG Document mars/03-06-12 ed., June 2003.

[34] Object Management Group, *Streams for CORBA Component RFP*, OMG Document mars/03-06-11 ed., June 2003.

[35] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," *Lecture Notes in Computer Science*, vol. 2072, pp. 327–355, 2001.

[36] Olaf Spinczyk and Andreas Gal and Wolfgang Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to C++," in *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Feb. 2002.