Configuring Real-time Aspects in Component Middleware

Nanbor Wang[1], Christopher Gill[2], Venkita Subramonian[2], and Douglas C. Schmidt[3]

[1] Tech-X Corp, Boulder, CO
nanbor@txcorp.com
[2] Distributed Object Computing Group, Washington University, St. Louis
{cdgill, venkita}@wustl.edu
[3] Institute for Software Integrated Systems, Vanderbilt University, Nashville
schmidt@vanderbilt.edu

Abstract. *This paper makes two contributions to the study of configuring real-time aspects into quality of service (QoS)-enabled component middleware and distributed real-time and embedded (DRE) systems. First, it compares and contrasts the integration of real-time aspects into DRE systems using conventional QoS-enabled distributed object computing (DOC) middleware versus QoS-enabled component middleware. Second, it presents experiments that evaluate several real-time aspects configured in The ACE ORB (TAO) versus in the Component-Integrated ACE ORB (CIAO). Our results show that QoS-enabled component middleware implementations can offer real-time performance that is comparable to DOC middleware, while offering greater flexibility in composing and configuring key DRE system aspects.*

Keywords. Real-time aspects, Component middleware, Real-time CORBA, CORBA Component Model.

1 Introduction

Developers of complex distributed real-time and embedded (DRE) systems need middleware technologies that offer (1) explicit configurability of policies and mechanisms for systemic aspects, such as real-time quality of service (QoS), so that developers can meet the stringent QoS requirements of modern DRE systems and (2) a programming model that explicitly separates systemic aspects from application functionality so developers can untangle code that manages systemic and functional aspects, resulting in systems that are less brittle and costly to develop, maintain, and extend. This section first describes how conventional real-time distributed object computing (DOC) middleware and component middleware technologies each provide one of these requisite capabilities, but not the other. We then describe our approach, which integrates configurability of real-time DOC middleware within a standards-based component middleware programming model.

Limitations with existing middleware technologies. Component middleware [1] technologies are an emerging paradigm that provides mechanisms to configure and control key distributed computing aspects, such as connecting event sources to event sinks and managing transactional behavior, *separate from the functional aspects of the application.* Conventional component middleware platforms, such as the Java 2 Enterprise Edition (J2EE) and the CORBA Component Model (CCM), are designed to address the QoS needs of enterprise application domains (such as workflow processing, inventory management, and accounting systems), which focus largely on scalability and transactional dependability. Other domains, however, require additional constraints to meet application requirements, *e.g.*, over 99% of all microprocessors are now used for DRE systems [2] that control processes and devices in physical, chemical, biological, or defense industries.

Examples of DRE systems include distributed sensor networks, flight avionics systems, naval combat management systems, and financial trading systems, which have stringent QoS requirements. In these types of systems the right answer delivered too late becomes the wrong answer, *i.e.*, failure to meet QoS requirements can lead to catastrophic consequences. Research over the past decade [3–5] has shown that the coordinated management of application and system resources is essential to ensure QoS. Conventional component middleware, however, does not provide adequate abstractions to control the mechanisms for managing these behaviors and thus is not suitable for applications in these domains.

Since the time/effort required to develop and validate DRE systems precludes developers from implementing these systems from scratch, attempts have been made to extend standard middleware specifications so they provide better abstractions for controlling and managing domain-specific aspects. For example, Real-time CORBA 1.x [6] – which is part of the DOC middleware CORBA 2.x specification [7] – introduces QoS-enabled extensions that allow DRE systems to configure and control (1) **processor resources** via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service for real-time applications with fixed priorities, (2) *communication resources* via protocol properties and explicit bindings to server objects using priority bands and private connections, and (3) *memory resources* via buffering requests in queues and bounding the size of thread pools.

Although CORBA 2.x provides mechanisms to configure and control resource allocations of the underlying endsystem to meet real-time requirements, it lacks the flexible higher level abstractions that component middleware provides to separate real-time policy configurations from application functionality. Manually integrating real-time aspects within CORBA 2.x application code is therefore unduly time consuming, tedious, and error-prone [8]. It is therefore hard for developers to configure, validate, modify, and evolve complex DRE systems consistently using QoS-enabled DOC middleware, such as CORBA 2.x and Real-time CORBA.

Solution approach → Integrating real-time QoS aspects into component middleware. To resolve the limitations with the status quo described above, we are integrating (1) *component middleware*, which enables behavioral aspects to be specified declaratively and woven into application functionality automatically – rather than programmed imperatively by hand – with (2) *QoS-enabled DOC middleware*, which supports end-to-end QoS specification and enforcement, to create *QoS-enabled component middleware*. Successful integration of these two approaches requires the resolution of the following challenges:

- The component middleware's configuration infrastructure must be extended to incorporate specification interfaces, tunable policies, and enforcement mechanisms for real-time aspects and
- The performance and predictability of the resulting QoS-enabled component middleware's runtime environment must be validated empirically to ensure it supports the desired real-time properties end-to-end.

This paper extends our prior work [9, 10] by comparing the complexity of *programming* CORBA 2.x DOC middleware features directly in the ACE ORB (TAO) [11] (which implements the CORBA 2.x DOC middleware standard – including Real-time CORBA) versus *configuring* them via component middleware features in the context of the Component-Integrated ACE ORB (CIAO) [10] (which is a QoS-enabled implementation of CORBA 3.x CCM specification implemented atop TAO).[4] It also presents experiments that compare the real-time performance of an example DRE system implemented in TAO and CIAO. Our results show that QoS-enabled component middleware implementations can offer performance and predictability similar to that of real-time DOC ORB middleware, while improving flexibility to compose and configure key DRE system QoS aspects.

Paper organization. The remainder of this paper is organized as follows: Section 2 illustrates how our work on CIAO overcomes limitations with earlier work on component and DOC middleware; Section 3 presents experiments comparing TAO and CIAO's real-time performance; Section 4 surveys related work on component models and integration of real-time aspects in middleware; and Section 5 presents concluding remarks.

2 Composing Real-time Behaviors into DRE Applications

In conventional component middleware, there are multiple software development roles, such as component designers, assemblers, and packagers. QoS-enabled component middleware supports yet another development role – the *Qosketeer* [3] – who is responsible for performing *QoS provisioning*. QoS provisioning involves (pre)allocating CPU resources, reserving network bandwidth/connections, and

[4] TAO, CIAO, and the tests described in this paper are available as open-source from deuce.doc.wustl.edu/Download.html.

monitoring/enforcing the proper use of system resources at runtime to meet or exceed application and system QoS requirements [12].

To improve component reusability and provision resources robustly throughout a QoS-enabled component middleware platform, QoS provisioning specifications should be decoupled from component implementations and specified instead via component composition metadata, such as ... This decoupling enables QoS provisioning specifications to be checked and synthesized via model-based tools [9, 13], which increase the level of abstraction and automation of the DRE system development process. This separation of concerns also makes DRE systems more flexible, easier to maintain, and easier to extend with new QoS capabilities to handle changing operational contexts. As DRE systems grow in scope and criticality, however, a key challenge is to decouple reusable, multi-purpose, off-the-shelf, resource management aspects from aspects that need customization for specific needs of each system. This section describes how CIAO addresses this challenge, presents an example DRE system that motivates our work on CIAO, and then uses this example to compare the development process using conventional DOC middleware technologies versus CIAO.

2.1 Supporting Real-time Aspects in CIAO

QoS provisioning requires component middleware that can meet the QoS requirements of the DRE systems it supports. The interfaces and mechanisms for QoS provisioning in the underlying operating systems and ORBs in conventional component middleware platforms do not provide adequate support for developing and deploying DRE systems with stringent QoS requirements, as follows:

- Since 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.
- Since (1) some resources (such as Real-time CORBA thread pools in CORBA 2.x [6]) can only be provisioned within a broader execution unit (*i.e.*, a component server rather than a component) and (2) component designers often have no *a priori* knowledge about other components, the component itself is not the right place to provision QoS.
- Since (1) some QoS assurance mechanisms (such as checking whether rates of interactions between components violate specified constraints) affect component interconnections and (2) a reusable component implementation may not know how it will be composed with other components, it is not generally possible for components to perform QoS assurance in isolation.
- Since (1) many QoS provisioning policies and mechanisms cannot work properly without installation of customized ORB modules (such as ...) and (2) there are inherent trade-offs between certain QoS requirements (such as high throughput and low latency), it is hard for QoS provisioning mechanisms implemented within components to foresee incompatibilities without knowing the end-to-end QoS requirements *a priori*.

To address the limitations of conventional component middleware in DRE system domains, therefore, it is necessary to make QoS provisioning policies an integral part of component middleware, while also decoupling QoS provisioning policies from component functionality. Over the past several years, we have developed extensions to conventional component middleware that support composing real-time aspects and mechanisms more effectively by:

- Separating the concerns of managing QoS resources from those of component design and development so QoS management code is decoupled from components and system modules that span multiple nodes in a distributed system, and
- Making component implementations more robust and reusable since QoS provisioning via real-time aspects can now be composed with DRE systems transparently to the component implementations.

These extensions have been integrated into the CIAO [10], which is our QoS-enabled CCM implementation that separates the programming and provisioning of QoS concerns as outlined above. Figure 1 depicts the key elements in the CIAO architecture. Key building blocks in CIAO support the three major categories of

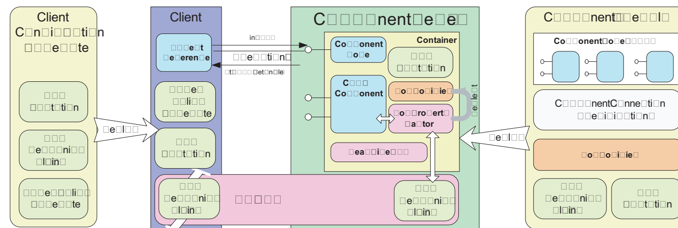

Fig. 1. Key Elements in CIAO

CCM: (1) component implementation, (2) deployment and configuration, and (3) application runtime. CIAO applies a range of aspect-oriented [14] development techniques to support the separation and composition of real-time behaviors and other configuration concerns that must be consistent throughout many parts of the software.

To support the composition of real-time behaviors, CIAO extends the building blocks in the “component implementation” and “application runtime” support categories of CCM to allow developers of DRE systems to specify the required real-time behaviors and to associate them with components in various parts of an application. In particular, application components and the runtime environment managed by CIAO can be configured to support the specified behaviors. For example, CIAO’s real-time component server runtime environment and containers can be configured to enforce different QoS aspects, such as priorities or rates of invocation.

CIAO's real-time enhancements to CCM define a new file format – known as the *real-time component assembly descriptor* (RTCAD) – to the set of XML descriptors that can can be composed into an existing application assembly. An RTCAD file contains definitions of *policy sets* that specify key real-time behaviors and the resources required to enforce the behaviors. The resources and policies defined in CIAO's RTCAD files can be specified for individual component instances. Qosketeers can then use CCM D&C tools to deploy the resulting application assembly onto platforms that support the specified real-time requirements. The remainder of this section present an example that shows how CIAO's real-time CCM enhancements can decouple QoS aspects from DRE system components and enable them to be composed separately.

2.2 An Example DRE System

To illustrate CIAO's support for composing real-time aspects into DRE systems concretely, we first describe an example DRE system from the domain of avionics mission computing [8]. Sections 2.3 and 2.4 then examine the steps required to develop and evolve this example using CORBA 2.x versus CIAO's real-time enhancements to the CORBA 3.x CCM specification. Figure 2 illustrates the

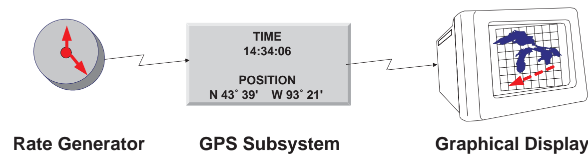

Fig. 2. Example DRE Avionics System

following primary software entities in our example DRE avionics system:

1. A **Rate Generator**, which wraps a hardware timer that triggers the pushing of events at specific periodic rates to event consumers that register for those events.
2. A **GPS Subsystem**, which wraps one or more hardware devices for navigation. Since there is a delay in getting the location reading from the hardware directly, a cached location value is served via the exposed interface to provide immediate response. The cached location value is refreshed when the GPS software receives a triggering event and causes the controlling software to activate the GPS hardware for updated coordinates. A subsequent triggering event is then pushed to registered consumers to notify the availability of a refreshed location value.
3. A **Graphical Display**, which wraps the hardware for a heads-up display device in the cockpit to provide visual information to the pilot. This device displays a cached location value that is updated by querying an interface when the controlling software receives a triggering event.

This example is representative of a class of DRE systems where clusters of closely-interacting components are connected via specialized networking devices, such as VME-bus backplanes. Although the functional characteristics of these systems may differ, they often share the rate-activated computation and display/output QoS constraints illustrated here.

2.3 Comparing DRE System Development using CORBA 2.x versus CIAO

The first step in developing a DRE system with either the CORBA 2.x DOC middleware specification (supported by TAO) or the CORBA 3.x CCM middleware specification (supported by CIAO) involves defining interfaces for the interactions between software entities. For example, to implement our example DRE avionics system using real-time DOC features of CORBA 2.x, a developer must first define the interface for interactions, such as sending the triggering message and querying the GPS for the current location reading. After these interfaces are defined, implementing the avionics example using CORBA 2.x involves the following steps:

1. Develop servant implementations for previously defined interfaces. These implementations are often specific to system hardware.
2. Determine the location of each servant implementation in the network of controllers. Hardware layout often dictates the selection of locations.
3. Based on decisions made in the previous steps, implement each server process as follows: (1) initialize and configure the ORB and hardware devices, (2) initialize and configure POAs to suit the needs of different servant implementations, (3) instantiate servants, register them with POAs, and activate them, (4) if needed, initialize and configure an event delivery mechanism, (5) acquire necessary object references for the system, *i.e.*, connect the referenced objects to this process, and (6) facilitate synchronization with other services and server processes so they are initialized in the right order.
4. Deploy the assembled implementations to the target platforms manually or via proprietary scripts.

Figure 3 presents a CORBA 2.x design for our example DRE avionics system. As the list of steps above indicates, much of the overall system functionality in CORBA 2.x is implemented in the server process and involves complex coordination among configuration and initialization code for specific hardware, ORB, POA, object connections, and initialization. This complexity is inherent to the CORBA 2.x development paradigm and requires careful programming of all objects and server processes involved.

In contrast, the CIAO CCM-based development paradigm provides a more scalable environment for managing key aspects of developing DRE systems. Figure 4 presents a CIAO CCM-based design for our example DRE avionics system, where each hardware device is wrapped within a *component* implementation. After the interfaces defining the interactions between hardware devices are defined, CIAO's development lifecycle involves the following steps:

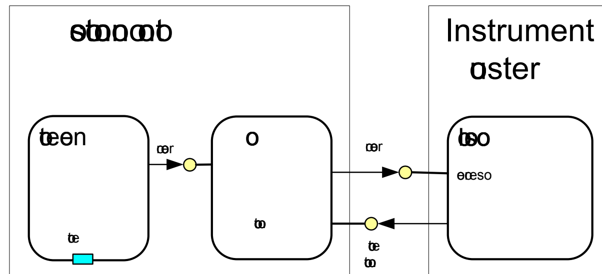

Fig. 3. CORBA 2.x Scenario for the DRE Avionics System

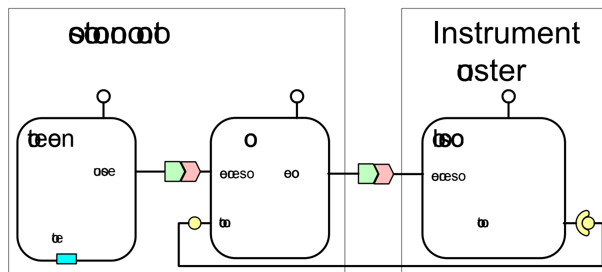

Fig. 4. CIAO Scenario for the DRE Avionics System

1. Identify a unit of installation as a component interface and design how the component interacts with external components by defining the component's ports and attributes. It is straightforward to identify the software component interfaces in this example since they map directly to hardware components.
2. For each type of component, developers create one or more component implementations (*e.g.*, for different hardware or internal algorithms) and bundle them as component packages.
3. An implementation of the DRE system can then be composed by defining a CCM assembly file where developers (1) select the component implementations to use from a pool of available component packages (which need not involve configuring any platform or runtime requirements, such as in the ORB or POA), (2) describe how to instantiate component instances using these component implementations, and (3) specify connections between component instances.
4. Deploy the DRE system onto its runtime platforms using the standard CCM Deployment and Configuration (D&C) framework and tools [15].

Compared with CORBA 2.x, CIAO's CCM-based development paradigm handles much of the complexity for DRE system developers. Developers can therefore focus on the domain problems at each development stage, without being distracted by low-level implementation details of the configuration platform, ORB, POA, and servant activation that are not related directly to application logic. Moreover, CIAO provides many flexible ways to configure a DRE system.

For example, the actual rate for the rate generator component can be specified as a default attribute value in a CCM component package and/or be overwritten in an application assembly by a Qosketeer. In contrast, CORBA 2.x systems require direct modifications to the application code, often by developer responsible for implementing the application functionality.

2.4 Comparing DRE System Evolution Using CORBA 2.x versus CIAO

When an existing DRE system is modified due to changes in DRE system requirements or available hardware, the benefits of CIAO's development paradigm become even clearer. For example, consider how our avionics example could be extended to include a collision warning subsystem to notify the pilot of imminent danger, consisting of a Rate Generator, a Collision Radar, and a Warning Display. Due to the critical nature of the collision warning subsystem, an additional requirement for this extension is that the collision warning subsystem be allocated resources in preference to the navigation subsystem. For example, the collision warning subsystem may run at a slow rate but it would likely always run at a higher priority than that of the navigation subsystem since when a collision alert was sounded, the pilot would perform an evasive maneuver to avoid a collision, rather than worrying about the exact location of the aircraft.

To evolve the solutions described in Section 2.3 to meet these new requirements, developers of DRE systems must first create the new software entities for the collision warning system and then integrate their *functional* aspects into existing applications. In addition, systemic QoS aspects, such as designation of thread pools and assignment of thread pool priorities, must be performed to ensure preferential operation of the collision warning subsystem.

Extending a CORBA 2.x implementation of our example DRE avionics system would require developers to perform the following steps:

1. Create new servant implementations for the Collision Radar, Warning Display, and possibly Rate Generator.
2. Reconfigure ORBs and POAs to accommodate and activate the new interface implementations.
3. Modify code at different points in the two subsystems to assign priorities, allocate thread pools, and set other ORB and POA policies so they interact and synchronize appropriately during system execution.

With CORBA 2.x, moreover, adding the code for resource allocation and real-time policy specifications would require additional intrusive modifications to application code, beyond those needed to integrate the subsystems's *functional* aspects.

Figure 5 illustrates how a CIAO-based implementation can be *configured* rather than *programmed* to support the new extensions outlined above. A key observation is that in CIAO the added components have the same interfaces as

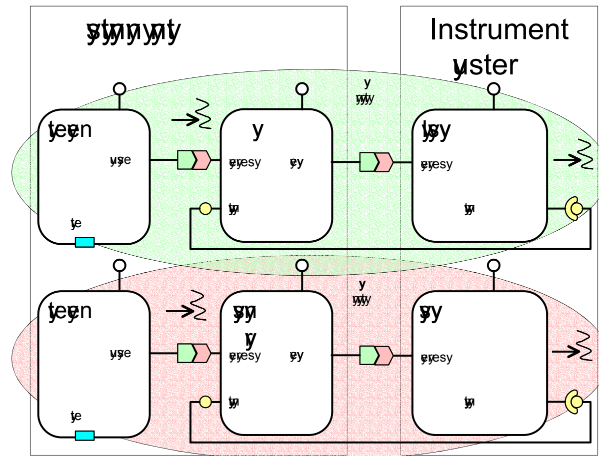

Fig. 5. Extended DRE Avionics System Scenario in CIAO

the original example DRE system, even though they require different implementations to interact with different hardware devices, including a collision radar and the warning light/speaker in the cockpit instrument cluster.

With CIAO's extensions to the CCM development paradigm, extending our example DRE avionics system becomes relatively straightforward, requiring the following steps:

1. Component developers write the new component implementations and package the new implementations with component metadata.
2. Application developers use the new component implementation packages to compose the additional functionality into the new DRE avionics system via the standard CCM D&C assembly format.
3. Qosketeers then define the QoS aspects (*i.e.*, resources and policies) associated with end-to-end real-time behavior, using CIAO's RTCAD format described in Section 2.1.

This aspect-oriented approach for configuring real-time properties supported by CIAO requires neither changes to the component implementations in our example DRE system nor any customized server modifications. Instead, this approach allows the real-time behavior of our example DRE avionics system to be changed simply by composing the new real-time behaviors into its application assembly specification via CIAO's RTCAD format. Creating modifications and variants to DRE systems with different real-time behaviors therefore largely reduces to deploying different application assemblies with CIAO. Moreover, many of these separate development steps can be separated into different development roles, so that developers (such as Qosketeers) can acquire and apply specialized expertise in particular focus areas of the overall DRE system development process.

3 Performance Evaluation

Section 2 presented a qualitative comparison of the steps involved in developing and evolving DRE systems using CORBA 2.x vs. CORBA 3.x. This section presents the design and results of experiments that quantitatively evaluate the effectiveness of CIAO's support for composing systemic QoS aspects to achieve real-time behavior. These experiments illustrate how different real-time behaviors can be composed and configured into existing applications via CIAO's real-time extensions described in Section 2.1. To achieve this goal, all experiments used components whose functional implementation was amenable to – but decoupled from – any real-time aspects.

The components used in the experiments presented in this section consisted of (1) a client component that initiated processing and (2) a worker component that performed a specified workload, akin to the relationship between the Rate Generator and GPS components in Figure 4. Different real-time aspects were then composed with these components in the experiments to model the following two types of tests:

- **TAO real-time tests**, which were presented in earlier work on the real-time features of CORBA 2.x in TAO [16]. In these tests, procedures for different tests were hard-coded into many client and server execution paths which, in turn, depend on complicated logic and scripts to determine the exact tests to perform.
- **CIAO real-time tests**, where different tests were composed – rather than programmed – by selecting and connecting different combinations of components and systemic policies. Fewer component implementations were therefore implemented to perform the tests using CIAO.

In addition to empirically evaluating CIAO and TAO, these two types of tests highlight the benefits of the CIAO development paradigm illustrated in Sections 2.3 and 2.4. In particular, the variations in QoS aspects were managed directly via reuse of CIAO configuration mechanisms and specification formats, rather than through additional manual programming with C++ in the TAO CORBA 2.x approach.

3.1 Testbed Hardware and Software

Two single-CPU 2.8 Ghz Pentium-4 computers with 512 MB of memory and 512 KB of on-chip cache memory served as deployment targets, providing the execution environment for the experiments performed in this chapter. Both machines ran KURT-Linux [17] 2.4.18, which provides a highly predictable experimentation platform. Two other single-CPU 2.53 Ghz Pentium-4 computers with the same OS and memory configuration as the 2.8 Ghz machines were used to deploy the test programs. All four machines were connected via switched 100 Mbps Ethernet.

All test programs, libraries, and tools were based on TAO version 1.3.5 and CIAO version 0.3.5 and compiled using GCC version 3.2 with no embedded

debug information, and with the highest level of optimization (–O3). All component implementations used in the test are implemented using the CCM *session* component category, which provides the functionality most relevant to the DRE systems. To remove spurious variability from the tests, all application processes ran as root in the KURT-Linux real-time scheduling class, using the SCHED_FIFO policy.

The basic interactions in the TAO real-time tests occurred between a test object/component provided by the server and a client invoking an operation on the object/component, thus requesting the server to perform an increment of CPU-intensive work. Different tests were derived from different configurations of the server and client. For example, the number of objects/components handled by the server and their real-time constraints were varied. Clients were also configured with different numbers of threads, each invoking operations in the server with different workloads in different ways, *e.g.*, at a fixed rate vs. continuously.

CIAO real-time tests needed only two basic component types, called *Controller* and *Worker*, to emulate the TAO real-time tests. A Worker component `provides` a common interface that contained an operation that a client invoked with an `in` parameter named `work` to specify the amount of work to perform. The CIAO real-time tests only required one Worker component that performed the specified amount of CPU computation when requested.

A Controller component `uses` a common interface to request that a connected Worker component perform a unit of work. Several Controller implementations were provided for the experiments. Each Controller implemented a particular invocation strategy, such as *continuous* or *rate-based* (at 25, 50, or 75 Hz rates). A Controller component also `supports` an interface for starting and stopping the test operation and outputting the statistic results observed in the controller. Multiple Controllers thus acted as the source of execution threads invoking operations at the server component at different rates. These experiments were based on the component design architecture found in modern avionics mission computing systems, as described in Section 2.2.

3.2 Experiment 1: CIAO vs. TAO Invocation Performance

Experiment goal. This experiment evaluated the performance overhead to componentize DRE systems with the real-time features in CORBA 2.x enabled in the TAO ORB and using CIAO's real-time component server environment described in Section 2.1. Although TAO and CIAO can be configured *without* support for Real-time CORBA features, those features are needed by many types of DRE systems, such as the example avionics application shown in Figure 2. We therefore focused on the performance of TAO and CIAO with Real-time CORBA features enabled.

Experiment design. The implementation of this test in TAO consisted of a pair of servants running on two test machines. The CIAO implementation of this test used two component implementations, where one `provides` the target interface, while the other component `uses` the same interface to invoke the benchmarking

operation. The CIAO test was built by using standard OMG tools to deploy the two components to the same two machines used for the TAO test.

Experiment 1 measured and compared performance by invoking a simple operation repeatedly in each test using either TAO or CIAO. We measured the latency of each call and the number of calls made per second. We then computed statistics to quantify the variability and average performance of each implementation, *i.e.*, in terms of average throughput and latency, maximum latency of all calls, maximum latency of the lower 99% of the calls, and the standard deviation in the latency of all calls.

Compared to TAO, an operation invocation on a CIAO component incurred an additional virtual method call when a generated servant forwards the invocation to the executor. Likewise, when a component invokes an operation on a receptacle interface, it must first retrieve the object reference stored in the container before invoking the operation on it. The cost for both the virtual method call and the retrieval of an object reference should ideally be predictable and small. This experiment therefore selected an operation signature with a small message payload, which made the overhead of CIAO stand out in comparison and offers an approximation of the worst-case CIAO performance difference for non-trivial operation invocations. Not including the length of other protocol headers, an 8 byte message payload was sent over the network, which reduced the time spent marshaling the data and sending the message over the network.

Experiment results. Figure 6 shows the throughput results measured in this test, which were 8,420 and 8,107 calls/sec for TAO and CIAO, respectively. These results show that CIAO incurs a 3.7% reduction in average throughput compared to TAO when CORBA 2.x real-time features are enabled in both ORBs. The latency and variability results from this experiment are shown in Figure 7. This figure shows the average latencies of TAO and CIAO calls were 118.9μsec and 122.9 μsec, respectively, indicating an increase of 4 μsec ($\sim$3.4%) in average latency. This result is consistent with the real-time ORB and CIAO real-time component server throughput results, and shows that the overhead imposed by CIAO's implementation when using CORBA 2.x real-time features is relatively small.

The other graphs in Figure 7 show the standard deviations, 99% latency bounds, and maximum measured latencies for TAO and CIAO with CORBA 2.x real-time features enabled. The standard deviations were again both small, *i.e.*, less than 2 μsec for both TAO and CIAO real-time tests. TAO's measured real-time latency had 99% of all samples under 123 μsec, and 99% of the measurements for CIAO fell within 127 μsec. In both cases, 99% of all samples fell within $\sim$4 μsec above their average latencies. The maximum latency results for TAO and CIAO tests were also comparable, at 182 μsec for TAO and 219 μsec for CIAO.

Analysis of results. The results depicted in Figure 7 show that with real-time features enabled the average- and worst-case performance for CIAO was slightly worse than for TAO, but was reasonably close overall. The results demonstrate

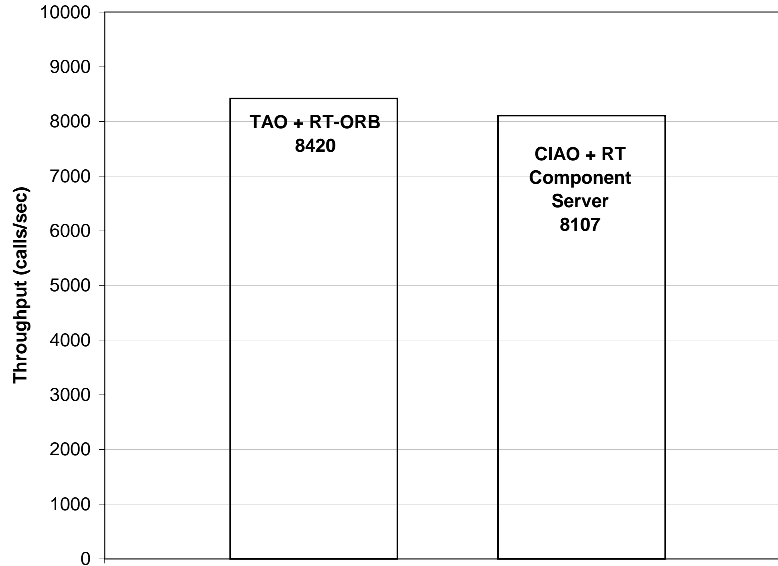

Fig. 6. Throughput Comparison between TAO and CIAO

that CIAO incurs only a small amount of overhead for supporting various CCM mechanisms and interfaces. As the payload size increases, moreover, CIAO's relative performance overhead in terms of throughput and latency will diminish accordingly. In addition, the results show that CIAO does not greatly affect jitter relative to TAO. In general, these results demonstrate the suitability of CIAO in the DRE application domain.

3.3 Experiment 2: Multiple Fixed-rate Controller/Worker Pairs

Experiment goal. This experiment evaluates the behavior of a server handling multiple worker threads over a range of workload without any real-time scheduling, priority assignment or admission control at the middleware level.

Experiment design. This experiment consisted of three Controller/Worker pairs running concurrently. Each controller made requests to the worker at its respective fixed rate of 25, 50, and 75 Hz (*i.e.*, at 25, 50, and 75 times per second, respectively). The worker handles requests from each individual Controller by doing the specified amount of work in separate threads. When invoking an operation on the Worker, a Controller will block until the invocation returns from the Worker. If a Worker can not handle the request at the given rate, therefore, a Controller will not be able to invoke operations at its designed rate.

This experiment measures the rate at which each of the three Controllers can make requests to the Worker components under different workloads. The total work performed by all three worker threads eventually exceeds the amount

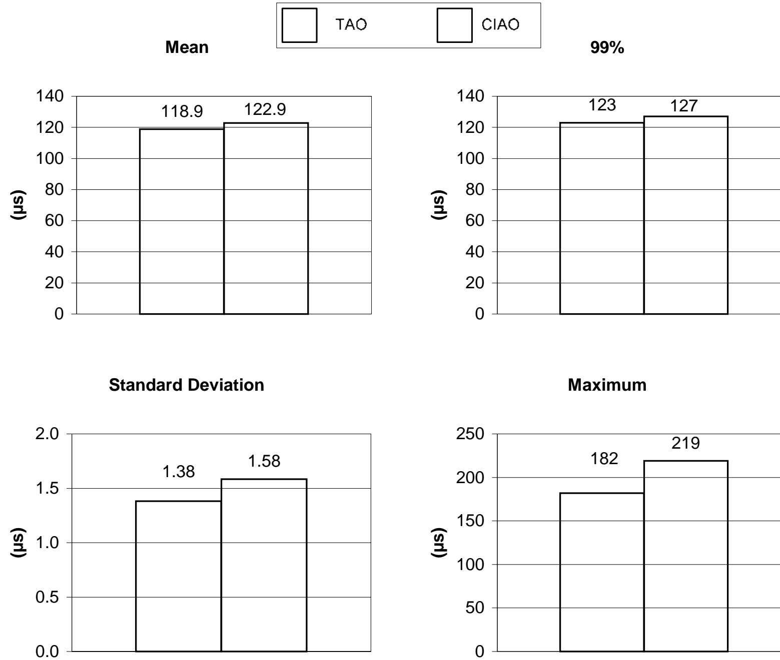

Fig. 7. Latency Comparison between TAO and CIAO

of work the server can handle since the server's capability to perform work is limited by the speed of its CPU. Since all three worker threads perform the same amount of work, we expect one or more worker threads will not be able to maintain the designated rate when the workload increases to the point where the server can no longer complete the 150 requested invocations per second.

Experiment results. Figure 8 shows the measured results from the experiment. This figure shows that when the workload increases initially from 20, all three Controllers can achieve their designed rates. After the workload increases to above 110, however, the 75 Hz Controller starts falling short of its target rate. Similarly, above a workload of 130 repetitions the 50 Hz Controller also falls short of its target rate. Above 210 repetitions all three Controllers fail to perform at their designed rates of invocation.

Analysis of results. As expected, there is only a bounded amount of computation power and the rate at which Worker components can handle requests eventually decreases as workload increases. Indiscriminately failing to meet the invocation rate of controllers, however, is often not acceptable for DRE systems. Instead, certain tasks must meet their execution rates even if there are not enough computation resources to enable all tasks to run at the specified rates.

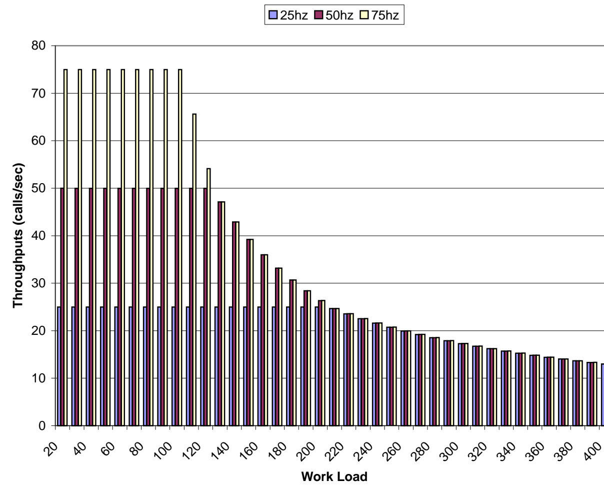

Fig. 8. Achievable Rates vs. Workload

3.4 Experiment 3: Prioritized Workers using Threadpools with Lanes

Experiment goal. This experiment evaluates the effectiveness of composing real-time behaviors into an application using the CORBA 2.x *threadpools with lanes* strategy to manage the reservation of processing resources. In a thread pool model *without lanes*, all threads in a thread pool have the same assigned priority and the priority is changed to match the priority of a client making the request. In the thread pool model *with lanes*, conversely, threads in a pool are divided into lanes that are assigned different priorities. The priority of the threads does not change once assigned.

Experiment design. This experiment allocates a single thread pool with multiple lanes for different priorities. Although the CIAO deployment tools still create multiple containers for host component of different priorities, they all share the same thread pool using this approach. We first assigned priorities according to an "increase rate, increase priority" (IRIP) strategy, also known as the Rate Monotonic [18] assignment of priorities. The same experiment is also performed with the anti-RMS "increase rate, decrease priority" (IRDP) strategy, to demonstrate CIAO's ability to configure a wide range of strategies for priority assignment and other real-time aspects.

Experiment results. The result of using threadpool with lanes with the IRIP strategy is shown in Figure 9. This approach yields the same result as that of using RMS with individual threadpools. Similarly, the result of composing the anti-RMS real-time behaviors with threadpool with lanes in Figure 10 shows that alternative priority assignments, such as IRDP, can be enforced effectively for thread pools with lanes.

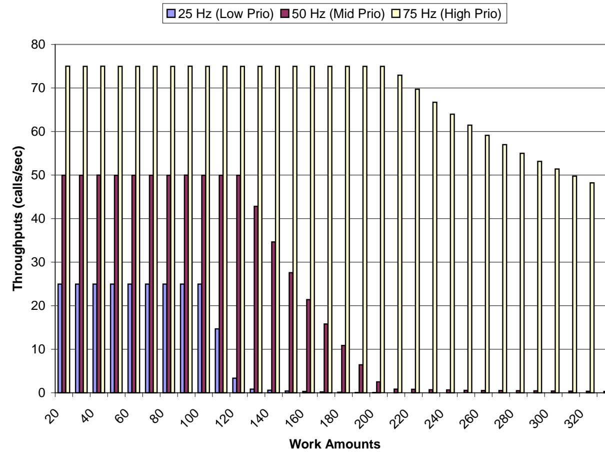

Fig. 9. IRIP with Threadpool Lanes

Analysis of results. The results in Figure 9 and 10 show that the composed real-time behaviors successfully added the desired real-time aspects, *i.e.*, prioritizing task handling. In the experiments, real-time aspects were composed at different stages, *i.e.*, real-time CORBA policies and resources at the component assembly stage and certain real-time ORB configurations at the deployment stage. Moreover, the applied RTCAD file utilized CIAO's support for composing real-time aspects at different granularities in an application, *i.e.*, threadpool configurations at the per-ORB level and sets of real-time policies at the container level.

The benefit of the CIAO development paradigm is evident when comparing the equivalent experiment programs used in TAO (see [16]) and those used in CIAO that are described here. TAO's real-time experiments require complex logic in both the client and server test programs and the collaboration of complicated script to cover configurations for all real-time behaviors performed. In comparison, the CIAO-based tests are *composed* by using component implementations and XML definitions to specify the test applications and real-time behaviors, which is much easier to manage and maintain. Since XML definitions for key component properties are directly readable – rather than being entangled with application code – systems built using the CIAO development paradigm are also easier to analyze.

3.5 Summary of Results

The experiments described in this section show that CIAO adds only a small amount of overhead, by comparing the performance of a CIAO application to an equivalent one based on TAO. Moreover, the proportion of overhead is expected to diminish with any increase in the size of an operation payload.

The current implementation of CIAO does, however, demand more secondary storage and primary memory for running CIAO applications, due to the addi-

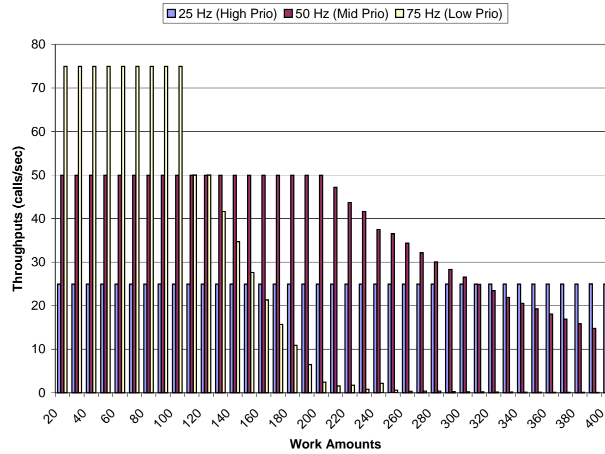

Fig. 10. IRDP with Threadpool Lanes

tional code required to provide a common CCM runtime environment consisting of containers and component servers. For simple test applications, this overhead may be relatively large, though for large-scale DRE systems the additional storage requirement is expected to be less significant compared to the storage requirements of the applications themselves. The extra footprint and storage space required by CIAO is also expected to lessen as implementation of CIAO matures and when the dependencies on unused libraries are removed as future work.

This section also showed that CIAO's runtime support for real-time applications imposes only a small amount of overhead to the overall performance and does not adversely affect predictability. Moreover, we have shown how CIAO's real-time extensions – particularly its RTCAD files that define the aspects (*i.e.*, resources and policies) associated with end-to-end real-time behavior – enable the composition of real-time behaviors into an application flexibly and effectively. Since developers can now integrate real-time behaviors throughout an entire application end-to-end, these extensions make developing, maintaining, and validating large-scale DRE systems easier.

The experiments performed in this work were modeled after existing TAO real-time tests that validate TAO's CORBA 2.x Real-time CORBA features [19]. Comparing CIAO's implementations to their TAO-based counterparts, one striking difference is how easy it is to develop and modify CIAO-based tests. Developing TAO test programs requires writing new tests *i.e.*, several specific programs are required to provide different tests of different configurations, as in the case of basic performance tests for TAO with and without a real-time ORB. In contrast, CIAO requires only a single application assembly, where different configurations can be achieved by using different standard tools provided in CIAO, *i.e.*, by changing the deployment environment configuration to use regular component server or real-time component server.

The real-time validation tests shown in Experiment 3 provide an even more obvious contrast. The TAO real-time test programs require elaborate design of test procedures, options, and configuration into the programs themselves and can only be run using proprietary Perl scripts. In contrast, the CIAO test programs use only a limited number of component implementations running on a common runtime environment, where application components are composed using CIAO's standard assembly and deployment tools. Different test configurations can be selected by simply deploying an assembly with different combination of application composition and real-time behaviors.

4 Related Work

This section reviews work in the areas of Qos-enabled DOC middleware and QoS-enabled component middleware efforts that are related to our research on CIAO.

QoS-enabled DOC middleware. Middleware can apply the Quality Connector pattern [20] to specify the QoS behaviors and configuring the supporting mechanisms for these QoS behaviors. The *Quality Objects* (QuO) framework [4,3] is an adaptive middleware framework developed by BBN Technologies that allows DRE system developers to use aspect-oriented software development [14] 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 [12] to provide an end-to-end QoS provisioning solution.

In their *dynamicTAO* project, Kon and Campbell [5] 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 is similar to *Qoskets*, in that both provide the mechanisms for realizing *dynamic* QoS provisioning at the middleware level. Qoskets offer a more comprehensive QoS provisioning abstraction, however, whereas Kon and Campbell's work concentrates on configuring *middleware* capabilities.

Moreover, although the dynamicTAO approach can provide QoS adaptation behavior by dynamically (re)configuring the middleware framework, it may not be suitable for some DRE systems, since dynamic loading and unloading of ORB components can incur significant and unpredictable overheads and thus prevent the ORB from meeting application deadlines. Our work on CIAO allows Model Driven Architecture (MDA) tools [21] to analyze the required ORB components and their configurations. This approach ensures the ORB in a component server contains only the required components, without compromising end-to-end predictability.

QoS-enabled component middleware. The container architecture in component-based middleware frameworks provides a vehicle for applying meta-programming techniques for QoS assurance control in component middleware. Containers can also help apply aspect-oriented software development [14] techniques to plug in different systemic behaviors [22]. These projects are similar to CIAO in that they provide mechanisms to inject aspects into applications statically at the middleware level.

Miguel de Miguel further develops the state of the art in QoS-enabled containers by extending a QoS EJB container interface to support a `QoSContext` interface that allows the exchange of QoS-related information among component instances [23]. To take advantage of the QoS-container, a component must implement `QoSBean` and `QoSNegotiation` interfaces. This requirement, however, adds an unnecessary dependency to component implementations.

The QoS Enabled Distributed Objects (Qedo) project [24] is another ongoing effort to make QoS support an integral part of CCM. Qedo targets applications in the telecommunication domain. It defines a metamodel with multiple categories 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 the QoS models [25].

Similar to QuO's Contract Definition Language (CDL), Qedo's contract metamodel provides mechanisms to formalize and abstract QoS requirements. QuO (Qosket) contracts are more versatile, however, because they not only provide high levels of abstraction for QoS status, but also define actions that should take place when state transitions occur in contracts. Qedo's extensions to container interfaces and 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 tightly couples the QoS provisioning and adaptation behaviors into the component implementation, and thus hampers the reusability of component. In comparison, our approach in CIAO explicitly avoids this coupling and tries to compose the QoS provision behaviors into the component systems.

Compared with other QoS-enabled component middleware technologies, CIAO aims to *compose* systemic aspects into component-based applications while these other technologies depend on specialized QoS-enabled container interfaces. While some applications do require direct interaction with QoS assurance mechanisms to adapt to system conditions, this approach forces QoS-aware component implementations to be tightly coupled to the type of container and thus reduces their reusability. In contrast, CIAO supports the composition of real-time behaviors with components, so that component implementations need not be tied to a specific container implementation.

5 Concluding Remarks

This paper describes how the Component-Integrated ACE ORB (CIAO) combines standards-based CORBA Component Model (CCM) middleware with Real-time CORBA distributed object computing (DOC) middleware features. Compared to using CORBA 2.x to develop an application, more steps are seemingly required to develop the same application using CIAO. This paper illustrates, however, that the additional steps needed for CIAO simplify codify (via standard CCM processes) activities that are performed in an *ad hoc* manner using CORBA 2.x. Moreover, CIAO alleviates many *accidental complexities* that can arise with CORBA 2.x by offering greater flexibility in composing and configuring key DRE system aspects *declaratively*, resulting in systems that easier to develop, maintain, and extend. Integrating real-time CORBA features with the CCM development model therefore offers developers of complex DRE systems

- Explicit configurability of policies and mechanisms for systemic aspects, such as real-time QoS, and
- A programming model that separates those systemic aspects from the application functionality.

The results of our experiments indicate that CIAO improves the flexibility of DRE systems without significantly affecting their quality of service, *i.e.*, its performance is comparable to that of the TAO real-time CORBA DOC middleware on which it is based.

References

1. G. T. Heineman and B. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Reading, Massachusetts: Addison-Wesley, 2001.
2. Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages, 3rd Edition*. Addison Wesley Longman, Mar. 2001.
3. 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.
4. 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), pp. 375–385, IEEE/IFIP, April/May 2002.
5. F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," *Communications ACM*, vol. 45, pp. 33–38, June 2002.
6. Object Management Group, *Real-time CORBA Specification*, OMG Document formal/02-08-02 ed., Aug. 2002.
7. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0.2 ed., Dec. 2002.
8. D. C. Sharp and W. 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. A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications," *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2004.
10. N. Wang and C. Gill, "Improving Real-Time System Configuration via a QoS-aware CORBA Component Model," in *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, (Honolulu, HW), HICSS, Jan. 2003.
11. D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.
12. 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.
13. J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems," in *Proceedings of the 25th International Conference on Software Engineering*, (Portland, OR), May 2003.
14. 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.
15. Object Management Group, *Deployment and Configuration Adopted Submission*, OMG Document ptc/03-07-08 ed., July 2003.
16. I. Pyarali, D. C. Schmidt, and R. Cytron, "Techniques for Enhancing Real-time CORBA Quality of Service," *IEEE Proceedings Special Issue on Real-time Systems*, vol. 91, July 2003.
17. Douglas Niehaus, *et al.*, "Kansas University Real-Time (KURT) Linux." www.ittc.ukans.edu/kurt/, 2004.
18. C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, Jan. 1973.
19. I. Pyarali, D. C. Schmidt, and R. Cytron, "Achieving End-to-End Predictability of the TAO Real-time CORBA ORB," in 8th *IEEE Real-Time Technology and Applications Symposium*, (San Jose), IEEE, Sept. 2002.
20. J. K. Cross and D. C. Schmidt, "Applying the Quality Connector Pattern to Optimize Distributed Real-time and Embedded Middleware," in *Patterns and Skeletons for Distributed and Parallel Computing* (F. Rabhi and S. Gorlatch, eds.), Springer Verlag, 2002.
21. A. Gokhale, D. C. Schmidt, B. Natarajan, J. Gray, and N. Wang, "Model Driven Middleware," in *Middleware for Communications* (Q. Mahmoud, ed.), New York: Wiley and Sons, 2003.
22. D. Conan, E. Putrycz, N. Farcet, and M. DeMiguel, "Integration of Non-Functional Properties in Containers," *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001.
23. M. A. de Miguel, "QoS-Aware Component Frameworks," in *The* 10th *International Workshop on Quality of Service (IWQoS 2002)*, (Miami Beach, Florida), May 2002.
24. FOKUS, "Qedo Project Homepage." http://qedo.berlios.de/.
25. 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.