CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations

Arvind S. Krishna, Balachandran Natarajan, Aniruddha Gokhale and Douglas C. Schmidt

Electrical Engineering and Computer Science, Vanderbilt University, Nashville TN, USA

{arvindk, bala, gokhale, schmidt}@dre.vanderbilt.edu

Nanbor Wang

Computer Science Department

Washington University, St. Louis, MO, USA

nanbor@cs.wustl.edu

Gautam Thaker

Lockheed Martin Advanced Technology Labs

Cherry Hill, NJ, USA

gthaker@atl.lmco.com

Abstract

Commercial off-the-shelf (COTS) middleware is now widely used to develop distributed real-time and embedded (DRE) systems. DRE systems are themselves increasingly combined to form "systems of systems" that have diverse quality of service (QoS) requirements. Earlier generations of COTS middleware, such as Object Request Brokers (ORBs) based on the CORBA 2.x standard, did not facilitate the separation of QoS policies from application functionality, which made it hard to configure and validate complex DRE applications. The new generation of component middleware, such as the CORBA Component Model (CCM) based on the CORBA 3.0 standard, addresses the limitations of earlier generation middleware by establishing standards for implementing, packaging, assembling, and deploying component implementations.

There has been little systematic empirical study of the performance characteristics of component middleware implementations in the context of DRE systems. This paper therefore provides four contributions to the study of CCM for DRE systems. First, we describe the challenges involved in benchmarking different CCM implementations. Second, we describe key criteria for comparing different CCM implementations using key black-box and white-box metrics. Third, we describe the design of our CCMPerf *benchmarking suite to illustrate test categories that evaluate aspects of CCM implementation to determine their suitability for the DRE domain. Fourth, we use* CCMPerf *to benchmark CIAO implementation of CCM and analyze the results. These results show that the CIAO implementation based on the more sophisticated CORBA 3.0 standard has comparable DRE performance to that of the TAO implementation based on the earlier CORBA 2.x standard.*

Keywords: CCM, Benchmarking, CCMPerf, white-box metrics, black-box metrics.

1. Introduction

Emerging trends. Distributed real-time and embedded (DRE) systems are becoming more widespread and important. Common DRE systems include telecommunication networks (*e.g.*, wireless phone services), tele-medicine (*e.g.*, robotic surgery), and defense applications (*e.g.*, total ship computing environments). These DRE systems are increasingly used for a range of applications where multiple systems are interconnected to form system of systems that possess stringent quality of service (QoS) constraints, such as bandwidth, latency, jitter and dependability requirements. A challenging requirement for these systems involves supporting a diverse set of QoS properties, such as predictable latency/jitter, throughput guarantees, scalability, and 24x7 availability, dependability, and security, that must be satisfied simultaneously in real-time. Conventional distributed object computing (DOC) middleware frameworks (such as DCOM, Java RMI, and earlier versions of the CORBA 2.x standard) do not provide capabilities for developers and end-users to specify and enforce these QoS requirements simultaneously in complex DRE systems.

Component middleware [25] is a class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly. The CORBA Component Model (CCM) [15] is a standard component middleware technology that addresses limitations with earlier versions of CORBA 2.x middleware based on the DOC model. In particular, the CCM standard defined

by the CORBA 3.x specification extends the CORBA 2.x object model to support the concept of components and establishes standards for specifying, implementing, packaging, assembling, and deploying components.

Empirically evaluating CCM implementations. Component middleware in general – and CCM in particular – are a maturing technology base that represents a paradigm shift in the way complex DRE systems have been developed traditionally. For example, component middleware provides higher-level capabilities for developers and end-users to specify and enforce QoS requirements in complex DRE systems. Several implementations of CCM are now available, including the Component Integrated ACE ORB (CIAO) [28], Mico-CCM [12], Qedo [18], and Star-CCM [23]. As CCM platforms mature and become suitable for DRE systems it is desirable to devise a standard set of metrics to compare and contrast different CCM implementations in terms of their:

- *Suitability*, *e.g.*, how suitable is the CCM implementation for DRE applications in a particular domain, such as avionics, total ship computing, or telecom systems?

- *Quality of service*, *e.g.*, does a CCM implementation for the DRE domain provide predictable performance and consume minimal time/space resources?

- *Conformance*, *e.g.*, does a CCM implementation conform to OMG standards by meeting the portability and interoperability requirements defined by the CCM specification?

Earlier efforts, such as the Open CORBA Benchmarking [26] and Middleware Comparator [10] projects, have focused on metrics to compare middleware based on the DOC middleware standard defined by the CORBA 2.x specifications. Our work enhances these efforts by focusing on a previously unexplored topic: *designing a benchmarking framework to compare CCM implementation quality by developing metrics that evaluate the suitability of those implementations for representative DRE applications*. To quantify these comparisons systematically we developed CCMPerf, which is an an open-source[1] benchmarking suite that focuses on *black-box* and *white-box* metrics, using criteria such as latency, throughput, and footprint measures. These metrics can be partitioned into the follow categories:

- *Distribution middleware* tests that quantify the overhead of CCM-based applications relative to applications based on earlier versions of the CORBA 2.x standard that do not support component run-time, configuration, and deployment capabilities.

- *Common middleware services* tests that quantify the suitability of using different implementations of CORBA services, such as Real-time Event [14] and Notification Services [13].

- *Domain-specific middleware* tests that quantify the suitability of CCM implementations to meet the QoS requirements of a particular DRE application domain, such as static linking and deployment of components in an avionics mission computing architecture [21].

This paper provides the following contributions to the study of component middleware implemented in accordance with the OMG CCM standard by describing:

1. The challenges involved in benchmarking different CCM implementations,

2. The criteria for comparing different CCM implementations using key black-box and white-box metrics, and

3. The design of our CCMPerf benchmarking suite that evaluates aspects of CCM implementations to determine their suitability for the DRE domain.

The vehicle used to test, obtain and analyze our results from CCMPerf is the *Component Integrated ACE ORB* (CIAO) [28], which is an open-source[2] implementation of CCM built upon the Real-time CORBA infrastructure of *The ACE ORB* (TAO) [20]. This paper shows how CCMPerf can be used to collect metrics and evaluate CCM implementations in the DRE domain. Our results show that CIAO and its more sophisticated CORBA 3.x CCM capabilities do not add appreciable overhead relative to its TAO CORBA 2.x foundation.

Paper organization. The remainder of this paper is organized as follows: Section 2 provides an overview of the elements in CCM; Section 3 discusses the design of CCMPerf, focusing on the performance experiments it supports; Section 4 analyzes quantitative results obtained by benchmarking CIAO using CCMPerf; Section 5 compares our work with other middleware benchmarking efforts; and Section 6 presents concluding remarks.

2. Overview of CCM

The CORBA Component Model (CCM) forms a key part of the CORBA 3.0 standard. CCM is designed to address the limitations with earlier versions of CORBA 2.x middleware that supported a distributed object computing (DOC) model [5]. Figure 1 depicts the key elements in the architecture of CCM. The remainder of this section describes each of these CCM elements.

---

1 CCMPerf is available for download from `deuce.doc.wustl.edu/Download.html`.

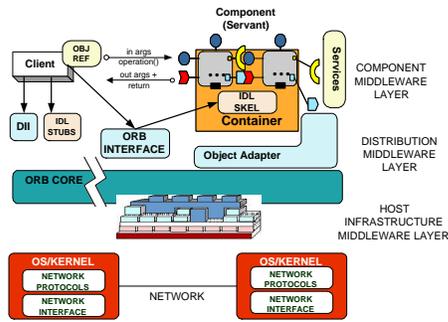2 CIAO is also available for download from `deuce.doc.wustl.edu/Download.html`.

Figure 1. Elements in the CCM Architecture

Components. *Components* in CCM are implementation entities that collaborate with each other via *ports*. CCM supports several types of ports, including (1) *facets*, which define an interface that accepts point-to-point method invocations from other components, (2) *receptacles*, which indicate a dependency on point-to-point method interface provided by another component, and (3) *event sources/sinks*, which indicate a willingness to exchange typed messages with one or more components.

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

Component Assembly. In a distributed system, a component may need to be configured differently depending on the context in which it is used. As the number of component configuration parameters and options increase, it can become tedious and error-prone to configure applications consisting of many individual components. To address this problem, the CCM defines an *assembly* entity to group components and characterize the meta-data that describes these components in an assembly. Each component's meta-data in turn describes the features available in it (*e.g.*, its properties) or the features that it requires (*e.g.*, its dependencies).

CCM assemblies are defined using XML Schema templates, which provide an implementation-independent mechanism for describing component properties and generating default configurations for CCM components. These assembly configurations can preserve the required QoS properties [28] and establish the necessary
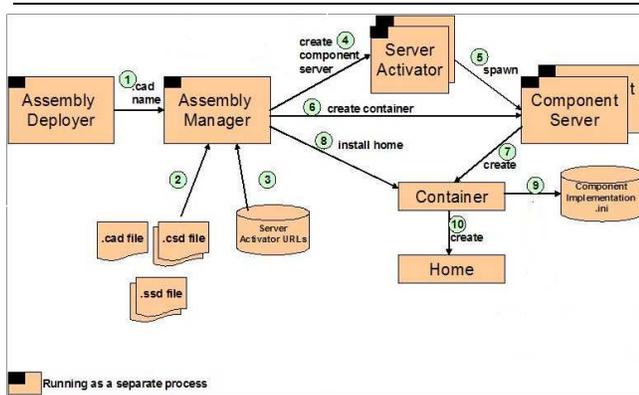
configuration and interconnections among groups of components.

Component server. A *component server* is an abstraction that is responsible for aggregating *physical* entities (*i.e.*, implementations of component instances) into *logical* entities (*i.e.*, distributed application services and subsystems). A CCM component server is a singleton [3] that plays the role of a factory to create containers and standardizes the role of a server process in the CORBA 2.x object model. Each component server is typically assigned a particular set of capabilities within a distributed system.

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

implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL).

Summary. Figure 2 depicts the interaction between the various CCM elements discussed in this section. As shown

Figure 2. Interaction between CCM entities

in this figure, a deployment application creates an assembly manager that is responsible for creating component assemblies from configuration files. Each of these assemblies are hosted in a component server that plays the role of a factory to create containers, which provide the execution environment for the components. A component home is a factory that manages the lifecycle of one type of component.

Figure 1 illustrates how CCM is a layer residing atop an ORB that leverages ORB functionality (such as connection management, data transfer, (de) marshaling of messages, and management and data transfer) event/message demultiplexing) and higher-level CORBA services (such as and higher-level CORBA services (such as Load Balancing, Transaction, Security, and Persistence). CCM applications may therefore incur additional overhead compared to their CORBA 2.x counterparts in the form of additional processing in the code-path (*i.e.*, additional function calls) and data-path (*i.e.*, parameter passing between the underlying ORB and the CCM layers). Since this processing can occur in the critical path of every request/response the overhead may be non-trivial. The remainder of this paper presents key criteria and empirical results that compare CCM implementations and presents empirical results that quantify the overheads added by the CIAO CCM implementation.

3. Overview & Design of CCMPerf

The goals of CCMPerf are to create comprehensive benchmarks that allow users and CCM developers to:

1. Evaluate the overhead CORBA 3.x CCM implementations impose above and beyond CORBA 2.x implementations that are based on the earlier-generation DOC model.

2. Devise and apply benchmarks that systematically identify performance bottlenecks in popular CCM implementations.

3. Compare different CCM implementations in terms of key metrics, such as latency, throughput, and other performance criteria.

4. Develop a framework that automates benchmark tests and facilitates seamless integration of new benchmarks.

This section describes the key challenges involved in developing a benchmarking suite for CCM to address the goals outlined above and shows how these challenges were addressed by CCMPerf. We also illustrate the three experimentation categories in CCMPerf and present a sample of empirical results obtained from applying CCMPerf to CIAO CCM middleware.

3.1. CCM Benchmarking Challenges and Their Resolutions

During the design of CCMPerf we encountered a number of challenges, including (1) heterogeneity in CCM implementations, (2) differences in quality of CCM implementations, (3) differences in application domains, and (4) heterogeneity in hardware and software platforms. We describe each of these challenges below and discuss how we resolve them in CCMPerf.

3.1.1. Heterogeneity in CCM Implementations

Context. CCM implementations use different tools and mechanisms to develop and configure applications, *e.g.*:

- CCM header files are not standardized by the OMG. Moreover, the process of obtaining the generated files (*e.g.*, the compilation chain for the different descriptor files explained in Section 2) used by CCM is specific to each ORB and its CCM implementation.

- Conformance to CCM features, such as automation of component assembly, is inconsistent across CCM implementations.

Problem. A benchmarking framework should encapsulate implementation heterogeneity to ensure its tests are (1) *representative*, *i.e.*, test equivalent configurations and (2) *repeatable*, *i.e.*, be amenable to continuous benchmarking. Of course, these challenges are a microcosm of the issues that CCM application developers must address to ensure portability across heterogeneous CCM implementations.

Solution. To shield CCMPerf from CCM implementation heterogeneity we developed a set of scripts to configure and run its benchmarking tests. These scripts automatically generate CCM platform-specific code and project build files for each implementation. The scripts are similar to the CORBAConf project [19] that provides autoconf support for CORBA 2.x ORBs.

3.1.2. Difference in Quality of CCM Implementations

Context. CCM implementations differ in the data structures and algorithms they use internally, which affects the QoS they can deliver to DRE applications.

Problem. Evaluating these differences requires instrumenting the code within the ORB/CCM implementation, which presents the following challenges:

- A thorough understanding of CCM implementations is needed to instrument CCM middleware with probes that measure performance accurately. No systematic body of knowledge yet exists, however, that identifies the critical features within CCM where instrumentation points should be added.

- CCM implementations are layered architectures, which makes it necessary to isolate each layer to measure its influence on overall end-to-end application performance. Since ORB-specific configuration options influence the presence/absence of these layers it is hard to identify the set of steps within each layer for every combination of configuration options.

Solution. As discussed in Section 3.2, CCMPerf provides benchmarks that use a combination of white-box and black-box metrics to evaluate CCM quality of implementation issues.

3.1.3. Differences in CCM Configuration Options

Context. CCM implementations differ in the configurable parameters they provide to tune performance, *e.g.*, run-time configuration options (such as the number of threads, logging levels, and locks) that can be enabled to fine tune different CCM implementations.

Problem. The presence of implementation-specific CCM configuration options yields the following challenges:

- The same set of configuration options many not be supported by all CCM implementations, *e.g*, CIAO allows applications to configure the type of locks used within the ORB, whereas Mico-CCM does not support this feature.

- An implementation can be optimized for a given set of configurations, yet perform poorly for other configurations, *e.g.*, Mico-CCM is optimized for single-threaded applications and performs poorly in multi-threaded configurations.

Solution. To ensure equivalent configurations, CCMPerf provides automated scripts to configure and run each test. The scripts capture the options that are used in different implementations to obtain equivalent CCM configurations. To ensure consistent hardware and OS configurations CCMPerf tests are run using EMULab [30] and Lockheed Martin Advanced Technology Lab's (ATL) Middleware Comparator framework [10]. These testbeds support systematic testing conditions that enable equivalent comparisons of performance differences between CCM implementations. ATL's Middleware Comparator framework also allows experiment data to be accessed via a convenient web interface (www.atl.external.lmco.com/projects/QoS/).

3.1.4. Differences in Application Domain

Context. Each CCM implementation can be tailored for a particular application domain, *e.g.*, the CIAO CCM implementation is tailored for the DRE domain, whereas Mico-CCM is targeted for the general-purpose distributed computing domain.

Problem. Different domains of applicability pose the following challenges:

- *Use cases may change across domains*, *e.g.*, some DRE applications require that total startup time be performed in under two seconds [22]. Component middleware catering to the DRE domain often needs to be optimized to meet this requirement, whereas middleware for general-purpose distributed computing might not require such optimizations.

- *QoS requirements may change across domains.* Certain metrics (such as predictable end-to-end latency and static/dynamic memory footprint) are important in the DRE domain, but are often less important in other domains, such as enterprise and desktop computing.

Solution. To evaluate domain-specific suitability, we provide scenario-based tests and/or enactments of specific use cases deemed important in a given domain, such as the DRE domain. In this context, we are evaluating CCM implementations using the scenarios present in the Boeing Bold Stroke Prism component model described in Section 3.2.

3.2. CCMPerf Benchmark Design

We now describe the design of CCMPerf, focusing on its three experimentation categories and the metrics collected in each of the categories. The benchmarking tests in CCMPerf focus on black-box and white-box metrics, as discussed below.

Black-box metrics. Black-box metrics are performance evaluation techniques that do not require instrumentation of software internals to select and analyze benchmark

data. CCMPerf can be used to benchmark CCM implementations without knowledge of their internal structure by using standard operations published in the CCM interfaces and without modifying CCM ORB internals. The black-box performance metrics supported by CCMPerf include:

- *Round-trip latency*, which measures the response time for a twoway operation with a single type of parameter, such as an array of CORBA::Long.

- *Throughput*, which compares the (1) number of events per second processed at the component server and (2) number of requests per second at the client.

- *Jitter*, which measures the variance in round-trip latency for a series of requests.

- *Collocation performance*, which measures response time and throughput when a client and server are in the same process vs. across processes on the same and different machines.

- *Data copying overhead*, which compares the variation in response time with an increase in request size to determine whether a CCM implementation incurs excessive buffer copying relative to a CORBA 2.x-based ORB.

- *Footprint*, which measures the static and dynamic footprint of a CCM implementation to determine whether it is suitable for memory-constrained DRE systems.

CCMPerf can measure each of these metrics in (1) single-threaded and (2) multi-threaded configurations on servers and clients.

White-box metrics. White-box metrics are performance evaluation techniques that employ explicit knowledge of software internals to select and analyze benchmark data. Unlike black-box metrics, white-box metrics evaluate performance by instrumenting the software internals with probes. The white-box performance metrics supported by CCMPerf include:

- *Functional path analysis*, which identifies CCM layers above the ORB and adds instrumentation points to determine the time spent in these layers. CCMPerf can analyze jitter by measuring the variation in the time spent in each layer.

- *Lookup-time analysis*, which measures the variation in lookup-time for certain operations, such as finding component homes, obtaining facets, and obtaining a component instance reference given its key.

- *Context switch overhead*, which measure the time required to interrupt the currently running thread and switch to another thread in multi-threaded configurations.

The benchmarking tests in CCMPerf can be categorized into the general areas discussed below. Each area then uses a range of black-box and white-box metrics to compare CCM implementations.

Distribution middleware benchmarks. These CCMPerf benchmarks employ black-box and white box metrics that measure various aspects of distribution middleware performance overhead, *e.g.*, for a given ORB and its CCM implementation the round-trip metric measures the increase in response time incurred by the CCM implementation beyond the CORBA 2.x DOC model support. Each CCM implementation resides atop a CORBA ORB. The ORB manages various network programming tasks, such as connection management, data transfer, (de)marshaling, demultiplexing, and concurrency. CCM implementations may add additional overhead to the underlying CORBA ORB, as explained in Section 2. Application developers and end-users can apply CCMPerf's distribution middleware benchmarks to evaluate how well CCM implementations meet their end-to-end QoS requirements. These benchmarks can also benefit users who are considering moving from DOC middleware to component middleware so they can quantify the pros and cons of such a transition.

Common middleware services benchmarks. These CCMPerf benchmarks quantify the performance of various implementation choices associated with integrating common middleware services within CCM containers. CCM leverages many standard services and features, as described in Section 2. CCM implementations can either use the standard CORBA service specifications or they can use customized implementations of these services. If CCM implementations use a publish/subscribe model, they can use the standard CORBA Notification and/or Event Services [14] or use a customized implementation (such as the TAO Real-time Event Service [6] or the OMG Data Distribution Service [16].

To benchmark the scenario where a container uses an event channel to publish events, CCMPerf measures the overhead introduced by extra (de)marshaling and indirection costs incurred within the container for publishing the events to the all the receivers. Black-box and white-box metrics defined in the Section 3.2 are also used to empirically compare and contrast the implementation choices for a particular application domain.

Domain-specific middleware benchmarks. The characteristics of an application domain often influence the selection and suitability of a particular service and/or its implementation. We therefore designed the CCMPerf benchmarking test suites to use the black-box and white-box metrics defined in the Section 3.2 to empirically compare and contrast the implementation choices for a particular application domain. These CCMPerf benchmarks include black-

box and white-box tests tailored for key domain-specific middleware use cases that occur in certain domains, such as Boeing's Bold Stroke Prism platform [22] that supports avionics mission computing in the DRE domain.

The purpose of these tests is to identify whether a given CCM implementation can meet the QoS requirements for a particular domain, *e.g.*, an organization might have a large number of components that need to be deployed within a certain amount of time. In the DRE domain, for instance, Boeing's Bold Stroke Prism architecture has several use cases with stringent timing constraints.

This category of benchmarks also include tests that analyze domain-specific CORBA implementations (such as Real-time CORBA) and protocols with real-time properties (such as the Stream Control Transmission Protocol [4]) standardized by the OMG. Although the CCM specification itself does yet not explicitly standardize real-time extensions, CCM implementations such as CIAO that target the DRE domain support the integration of Real-time CORBA and SCTP with CCM.

3.3. Summary

Benchmarking feature-rich component middleware implementations, such as CCM, poses several challenges. This section described how the design of CCMPerf (1) addresses the heterogeneity of CCM implementations, such as differences in configuration options, implementation quality, and domain of application, (2) provides black-box and white-box metrics to compare and contrast CCM implementations, and (3) consolidates tests into categories that clarify the structure of the benchmarks and facilitate the integration of new benchmark tests.

4. Empirically evaluating CIAO using CCM-Perf

This section presents distribution middleware and domain-specific benchmarks for CIAO. These experiments evaluate many of the black-box (round-trip latency, throughput, jitter, and collocation latency, throughput and jitter) and white-box (functional-path analysis) metrics described in Section 3. We also analyze the empirical results to evaluate the suitability of CIAO for DRE applications.

The following IDL interface was used for all the experiments:

```
module Test {
  interface RoundTrip {
    // Use a timestamp to measure the
    // roundtrip delay
    typedef unsigned long long Timestamp;

    Timestamp test_method (in Timestamp send_time);
  };
}
```

As shown in the RoundTrip interface, the communication entities exchange timestamps set at the client and propagated to the server. This design enables experimenters to set/unset time-probes at various points in the call-path and propagate them via this interface. For example, to compute the round-trip latency, experimenters compute the difference between the current time and the one encoded in the time-stamp.

Although both CIAO and TAO support a variety of configuration options [11], we make the following assumptions for this options, we make the following assumptions for this analysis:

1. Native exception handling was enabled

2. Servants are normal CORBA servants that inherit from PortableServer::ServantBase, *i.e.*, we do not consider the CORBA dynamic invocation interface (DII) and dynamic skeleton interface (DSI), which are inappropriate for most DRE systems

3. Logging was disabled

4. The ORB was configured to run in single-threaded mode and

5. No proprietary policies were associated with the ORB.

These assumptions are representative of a common class of DRE applications that apply ORB middleware.

4.1. Distribution Middleware Benchmarks

This section presents analysis of the *back-box* and *white-box* tests that quantify the overhead of CIAO over the base TAO ORB. All experiments presented were performed on an Intel Pentium IV 2.0 Ghz processor with 512 MB of main memory. For these experiments, TAO version 1.3.5 and CIAO 0.3.5 were compiled using the Timesys GNU g++ compiler version 3.2.2 and executed using on Linux 2.4.21-timesys-4.1.147 kernel. Each experiment was run in the Timesys Linux real-time scheduling class and a sample size of 250,000 data points was used for the resulting analysis.

4.1.1. Black-box Analysis In the *black-box* experiments, a client issued two-way operations at the fastest possible rate, while the server performed a minimal but non-trivial operation of cubing the data it received. Several black-box metrics were collected including, round-trip latency, throughput, jitter, collocation performance and data copying overhead analysis. These experiments quantify the overhead of CCM for normal CORBA operations.

Experiment description. The experiments consider the following usage scenarios in which an end-user may use CCM:
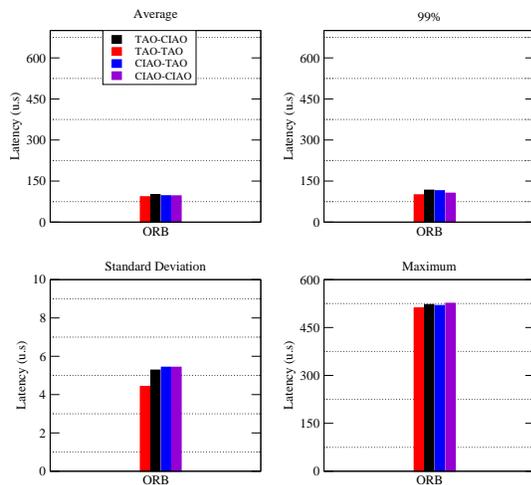
1. A CORBA 2.0 Server interacting with a CORBA 2.0 client,

2. A CORBA 2.0 Server interacting with a CCM Component (playing the role of the client) and

3. A CCM component (playing role of server) interacting with a CORBA client

4. A CCM component interacting with another CCM component (playing both client and server roles)

These four use cases represent all the possibilities of mixing and matching a CCM component with CORBA servers/clients. For each of the above interaction scenarios, we examined the following four combinations:

1. **TAO-TAO** – a TAO server interacting with a TAO client,

2. **CIAO-TAO** – a CIAO component interacting with a TAO client,

3. **TAO-CIAO** – a TAO server interacting with a CIAO component and

4. **CIAO-CIAO** – a CIAO component(server) interacting with another CIAO component (client).

The TAO-TAO interaction servers as the baseline to compute the overhead added by other combinations.

Round-trip analysis. This section analyzes the results of benchmarks that measure the average latency, 99% bound, the dispersion and worst-case behavior.
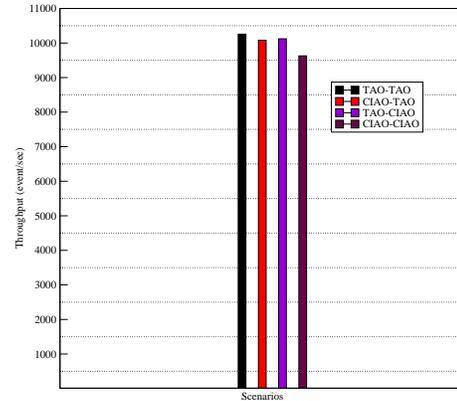

Figure 3. Round Trip Latency Analysis

- *Average measures* – Figure 3 shows that average latency for all the four case is nearly the same with

TAO-TAO scenario having the minimum average latency of ∼93.07 μsecs and CIAO-CIAO scenario having maximum latency of ∼100.54 μsecs. Using the average-case, therefore, the overhead added by CCM over CORBA is ∼8 μsecs.

- *Dispersion measures* – The term predictability has different connotations in different disciplines. For example, in real-time scheduling theory, a predictable system means that each task always meets its deadline. For these experiments, we define *predictability* as the measure of standard deviation of the data points. As seen from the above figure the dispersion measures for all the four cases are comparable to that of the baseline TAO-TAO case of ∼4.5. These measures show that the use of CIAO does not degrade predictability by increasing jitter.

- *Maximum measures* – The 99% and worst-case measures for all the four scenarios display behavior similar to the average measures. The 99% values are very close to the average indicating predictable latency. However, all the four scenarios do incur high worst-case measures. The worst-case measures for CIAO-CIAO scenario was marginally higher than all the cases. These results show that CIAO has worst-case measures similar to that of TAO.

Figure 4. Throughput Analysis

Figure 4, shows the throughput, *i.e.*, number of requests issued at the client side/second. As shown in the figure, the use of CIAO in all scenarios does not unduly degrade throughput. The throughput for TAO-TAO case was the most, ∼10,267 events/sec, while CIAO-CIAO case was the least, ∼9676 events/sec. Using the CIAO-CIAO interaction as the worst-case scenario and TAO-TAO interaction as the best case, the overhead added by CCM is ∼5.6%.

Collocation analysis. This section analyzes the results of benchmarks that measure the average latency, 99% bound,

the dispersion and worst-case behavior for round trip operations in collocated mode, *i.e.*, within the same address space. In the case of CIAO, the interacting collocated components reside within a single component server process. For this experiment, the only possible interaction scenarios are the TAO-TAO and CIAO-CIAO cases. For our experiments, we consider only the *thru_poa* collocation strategy [29] since the *direct* collocation strategy optimization is not permissible for CIAO. Direct collocation bypasses the POA to make invocations directly on the servant. CCM uses a "glue servant" registered with a POA to decouple the actual implementation from the POA, which requires all calls to go through the POA and making it impossible to use direct collocation. In addition, CIAO uses the POA to load the servant implementation on-demand, necessitating the use of a POA.
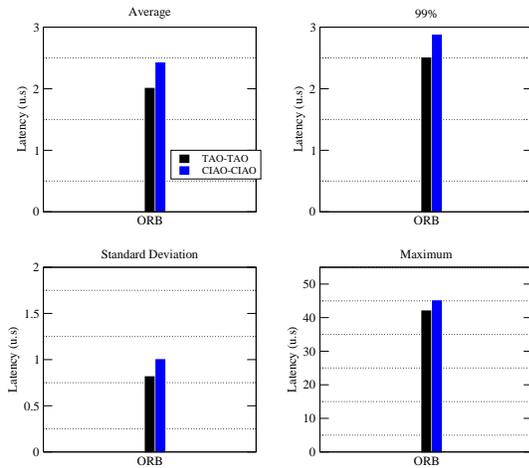

Figure 5. Collocation Analysis

- *Average measures* – Figure 5 depicts the average round-trip latency in collocated mode. As shown in the figure, the average latencies for CIAO-CIAO scenario are $\sim$2 μsecs. The overhead imparted by CIAO over TAO is thus $\sim$0.4 μsecs.

- *Dispersion measures* – The dispersion measures reveal that CIAO-CIAO case is comparable to that of TAO-TAO case. This result shows that the use of CIAO does not degrade predictability in the collocated mode relative to TAO.

- *Maximum measures* – The 99% bound for both the cases are $\sim$2 μsecs. Thus, for both the cases 99% of the observed samples are below $\sim$2 μsecs. The worst-case measures for CIAO are again comparable to that of TAO-TAO case, which illustrates that CIAO has worst-cases measures similar to that of TAO in collocated mode.

ORB	Throughput (events/sec)
TAO-TAO	385,370
CIAO-CIAO	323,832

Table 1. Collocation Performance: Throughput

Table 1 tabulates the collocation throughput, *i.e.*, number of client-requests/sec. As shown in the table, CIAO adds $\sim$15% overhead on top of TAO's collocation mechanism. This overhead stems from the additional call path that has to be traversed in CIAO. The additional steps required include (1) operation on the servant glue code and (2) actual method invocation on the executor. The white-box experiments discussed in Section 4.1.2 analyzes the functional path within CIAO to pinpoint where the overhead arises.

Data copying overhead analysis. These experiments measure roundtrip latency for exchanging n bytes of octets for TAO-TAO and CIAO-CIAO scenarios. The number of octets are varied from 4 to 64 K.B by powers of 2. The variation in round-trip latency with increase in message size is tabulated. The motivation of this analysis is to quantify additional data copying overhead incurred when request traverses various middleware layers the along the data-path. For larger message sizes, any additional overhead incurred is easily revealed. We now analyze the results of benchmarks that measure the average latency, 99% bound, the dispersion, and worst-case behavior for data copying overhead.
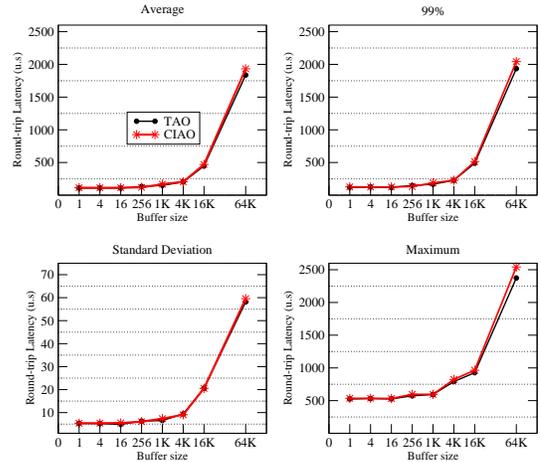

Figure 6. Data Copying Overhead Analysis

- *Average measures.* – Figure 6 illustrates the average round-trip latency values for TAO-TAO and CIAO-CIAO. As shown in the figure, for both TAO and CIAO, latency increases with an increase in request

size, the increase gradual up to message size of 4 K.B, after which there is a sharp increase. The results show that latency measure for CIAO are comparable and in many cases the same as that of TAO, *e.g.*, for message size of 4K.B, latency for CIAO ($\sim$205 μsecs) is nearly the same as TAO ($\sim$202 μsecs). This indicates that CIAO does not incur any additional data copying overhead along the critical request processing path.
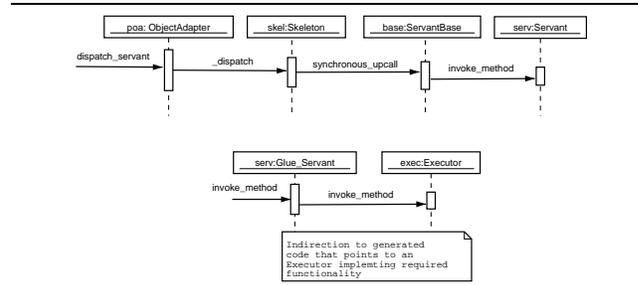
- *Dispersion measures.* – The dispersion measures for both TAO and CIAO display a trend similar to the average. For smaller messages sizes, both CIAO and TAO values are flat, indicating high predictability. For very large message sizes, the predictability degrades considerably. The increase in dispersion is not specific to TAO, *i.e.*, observed in other ORBs[3] and is influenced by external factors, such as cache misses for very large payload sizes. The similarity in the dispersion measures for both CIAO and TAO show that CIAO is as predictable as TAO.

- *Maximum measures.* – The 99% bound for both CIAO and TAO is close to the average and display trend similar to average-case. The worst-case measures for CIAO are comparable to that of TAO, *e.g.*, for request size of 16 K.B, the worst-case measure for CIAO ($\sim$927 μsecs) is nearly same as that of TAO ($\sim$923 μsecs). These results indicate that the worst-case behavior of CIAO is comparable to TAO.

4.1.2. White-box Analysis In the *white-box* experiments, a client issued two-way operations at the fastest possible rate, while the server performed a minimal but non-trivial operation of cubing the sent data. Functional-path analysis metrics are collected that quantify the overhead of the additional code path traversed by a CCM implementation such as CIAO, for processing a remote client request.
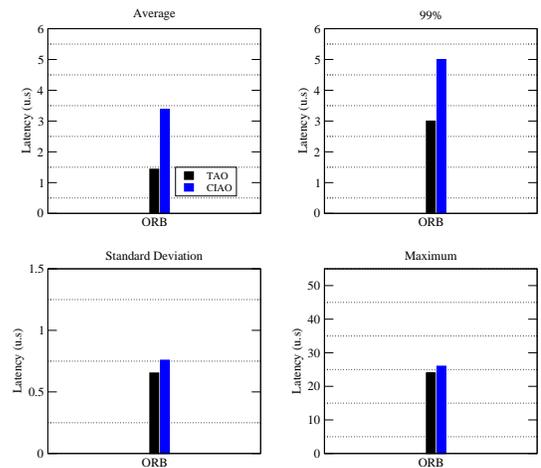
Functional path analysis. Figure 7, depicts the critical code path traversed by a remote request for TAO-TAO and CIAO-CIAO scenarios. As shown in the figure, CIAO incurs two additional method calls (1) to the generated servant from the POA and (2) to the executor that implements the functionality. The motivation for this experiment is to quantify this overhead imposed by CCM.

Result analysis. We now analyze the results of benchmarks that measure the average latency, 99% bound, the dispersion, and worst-case behavior for functional path analysis.

- *Average measures* – Figure 8 illustrates the results for both TAO and CIAO. As shown in the figure, the TAO results represent latency for a normal servant upcall,

Figure 7. Critical Code Path for TAO-TAO & CIAO-CIAO Scenarios

Figure 8. Functional-path Analysis

while the CIAO results denote the latency corresponding to TAO, plus the additional code path needed for CCM. The results indicate that this extra code path leads to an overhead of $\sim$1.5 μsecs, which shows that CIAO adds processing overhead in the critical code path. This overhead stems from conformance to the CCM specification (in particular the need to use a generated "glue servant") that necessitates the additional indirection.
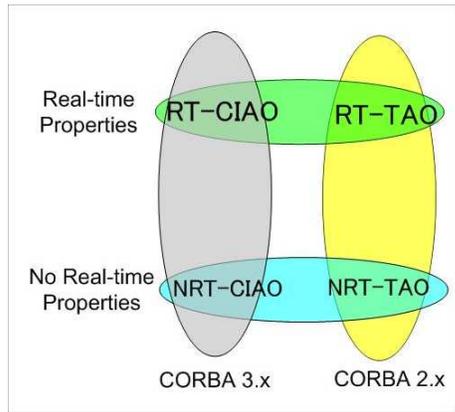
- *Dispersion measures* – The dispersion measures reveal that results of CIAO and comparable to that of TAO $\sim$0.75 μsecs. These results show that use of CIAO does not sacrifice predictability in the critical code path of component based systems.

- *Maximum measures* – The 99% bound for both TAO and CIAO are comparable to the average measures indicating predictable behavior. The worst-case measures for both TAO and CIAO show trend similar to the average- and worst-case measures, though worst-case measures for CIAO are marginally higher.

4.2. Domain-specific Benchmarks

The goal of domain-specific analysis is to evaluate the impact of supporting domain-specific requirements in CCM. These experiments are similar to the black-box experiments in distribution benchmarks. The hardware platform contains 2 Intel Pentium IV 2.8 Ghz processor with 512 MB of main memory running KURT real-time Linux [2] 2.4.18, developed by the Kansas State University, connected with a 100MB Ethernet switch.

Due to the difference between the hardware configuration and OS, the TAO-TAO and CIAO-CIAO tests described in Section 4.1 are performed to relate the domain-specific experiments to the conventional tests. These tests are then repeated again using implementations with domain-specific support. Figure 9 shows the following four scenarios measured in this experiment:

Figure 9. Domain-specific Benchmark Scenarios

- **NRT-TAO** – an interacting TAO object-based client and server that are configured *without* Real-time CORBA policies.

- **NRT-CIAO** – an interacting CIAO component-based client and server that are configured *without* Real-time CORBA policies.

- **RT-TAO** – an interacting TAO object-based client and server that are configured *with* Real-time CORBA policies.

- **RT-CIAO** – an interacting CIAO component-based client and server that are configured *with* Real-time CORBA policies.

The RT-TAO configured used a real-time ORB and Portable Object Adapter (POA). Likewies, the RT-CIAO configuration used a real-time component server was used. For each experiment, however, no Real-time CORBA policies (such as priority and protocol properties) were explicitly set on the objects/components.
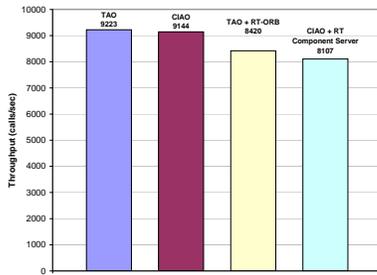
Figure 10 shows the latency results of comparing the following scenarios, which are similar to the TAO-TAO, TAO-CIAO, CIAO-TAO and CIAO-CIAO scenarios compared in the Distribution middleware benchmarks compared in Section 4.1.

Figure 10. Round Trip Latency Analysis for RT-CIAO

- *Average measures* – As shown in the figure, the average latency for all the four case is relatively close, with NRT-TAO scenario having the average latency of ∼108.43 μsecs and NRT-CIAO scenario having average latency of ∼109.05 μsecs. In comparison, the average latency of RT-TAO scenario is ∼118.91 μsecs and the average latency of RT-CIAO scenario is ∼122.87 μsecs. In either case, using NRT-CIAO does add minimal overheads of 0.62 μsecs and 3.96 μsecs to the overall latencies.

- *Dispersion measures* – As seen from Figure 10, the dispersion measures for all the four cases are comparable to that of the NRT-TAO case of ∼4 μsecs. These measures show that the use of NRT-CIAO or RT-CIAO does not degrade predictability by increasing jitter.

- *Maximum measures* – The 99% and worst-case measures for all the four scenarios display behavior similar to the average measures. The 99% values are very close to the average indicating predictable latency. All four scenarios do incur high worst-case measures, however, because no real-time priorities were explicitly associated with the components for the experiment.

Figure 11 shows the throughput of all for scenarios. This figure shows that the use of NRT-CIAO does not degrade

Figure 11. Throughput Analysis of RT-CIAO

throughput significantly, which is consistent with the conventional black-box results. RT-CIAO, however, does incur more overhead ($\sim$3.7%) compared to the non-RT CIAO ($\sim$0.9%).

4.3. Summary

This section presented distribution middleware and domain-specific benchmarks for various combinations CIAO and TAO. The round-trip latency measures for CIAO were comparable to that of TAO, indicating negligible overhead. Similarly, throughput and jitter results revealed that CIAO does not degrade performance significantly, while ensuring predictability. The data copying overhead measures showed that CIAO does not suffer from additional data copying along the critical request processing. The domain-specific benchmarks show that CIAO's real-time extensions help ensure predictability required for DRE applications.

5. Related Work

This section summarizes other benchmarking efforts that relate to our work on component middleware in general and CCM in particular. We decompose middleware into layers and describe the representative benchmarking efforts in each of the middleware layers.

Host infrastructural middleware. This layer encapsulates and enhances native OS mechanisms to create reusable event demultiplexing and interprocess communication mechanisms. A benchmarking effort at this layer is RTJPerf [1], which is an open-source benchmarking suite that measures the quality of various Real-Time Specification for Java (RTSJ) implementations. RTJPerf provides benchmarks for most of the RTSJ features that are critical to real-time and embedded systems.

Distribution middleware. Distribution middleware enables clients to program applications by invoking operations on target objects without hard-coding dependencies on their location, programming language and OS platform.

A benchmarking effort at this layer is the Open CORBA Benchmarking project [26], which is a generic benchmarking suite for various ORB implementations. The goal for this effort is to measure commonly used ORB functionality using metrics tailored for both ORB developers and ORB users.

Another CORBA 3.x and CORBA 2.x benchmarking effort [27] compares the performance and ease of use of Real-time CORBA implementation of TAO versus the real-time extensions added in CIAO. The results in this paper revealed that using component middleware enhanced the configuration of real-time policies via XML-based configuration files without sacrificing predictability. These results also showed that using component middleware enables the configuration of real-time policies that conforms to standard XML schemas.

Common middleware services. This layer provides higher-level domain-independent reusable services. A benchmarking effort at this layer is the Lockheed Martin Advanced Technology Lab's (ATL) [10] Middleware Comparator, which evaluates a range of middleware layers, including common middleware services via an easily accessible Web interface. In particular, the ATL tests evaluate the real-time publish subscribe architectures based on CORBA Data Distribution Service (DDS) [16]. ATL's methodology has been to use identical test conditions (*i.e.*, application, hardware, etc.), which permits comparisons that can reveal performance differences between various systems.

6. Concluding Remarks

Component middleware and QoS-enabled CORBA Component Model (CCM) implementations are important emerging technologies for distributed real-time and embedded (DRE) systems. Several initiatives are underway to develop commercial and research implementations of QoS-enabled CCM. There is not yet, however, a systematic body of knowledge that describes how to develop metrics that can systematically evaluate the correctness, suitability, and quality of CCM implementations for DRE systems.

Empirically evaluating feature-rich component middleware implementations, such as CCM, poses several challenges. This paper described how our CCMPerf benchmarking framework (1) addresses the heterogeneity of CCM implementations, such as differences in configuration options, implementation quality, and domain of application, (2) provides black-box and white-box metrics to compare and contrast CCM implementations at multiple middleware layers (*i.e.*, distribution middleware, common middleware services, and domain-specific middleware), and (3) consolidates tests into categories that

clarify the structure of the benchmarks and facilitate the integration of new benchmark tests. Our empirical results in Section 4 show how CCMPerf can be used to quantify metrics, such as overhead (*i.e.*, increases in the mean), that the CIAO CORBA 3.x CCM implementation incurs above and beyond its underlying TAO CORBA 2.x implementation. Our future work on CCMPerf will focus on benchmarking other open-source CCM implementations (such as Mico-CCM, Qedo, and StarCCM), as well as completing the white-box and scenario-based benchmarks and enhancing CCMPerf's testsuite.

Our work on CCMPerf has also underscored the importance of automating benchmarking experiments from higher level models. For example, to conduct a simple experiment requires developers to write (1) the header files and source benchmarking code that measures QoS, such as roundtrip latency and throughput, (2) IDL files that describes the contract between the client and the server, (3) the configuration and script files that tune the underlying middleware and automate the task of running tests and output generation, and (3) project build files (*e.g.*, makefiles) required to generate the executable code. Writing these files repeatedly for each experiment is tedious and error-prone. Further, in a hand-crafted approach, changing the configuration would entail re-writing the benchmarking code. In a model-based approach, however, the only change will be in the model and the necessary experimentation code will be automatically generated. A model-based approach also provides an effective abstraction to visualize and analyze the overall planning phase, rather than inspecting the source code manually.

To alleviate the shortcomings described above, we are developing the Benchmark Generation Modeling Language (BGML) [8, 9], which automates the generation of benchmarking experiments from high-level models. BGML has been integrated with CoSMIC [7], which is an integrated toolsuite for modeling design and runtime aspects of QoS-enabled component middleware. CoSMIC's model-based [24] approach to benchmark synthesis enables quality assurance engineers and testers to configure components, model test configurations, and generate benchmarking code automatically.

References

[1] A. Corsaro and D. C. Schmidt. Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems. In *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, San Jose, Sept. 2002. IEEE.

[2] Douglas Niehaus, *et al.*. Kansas University Real-Time (KURT) Linux. www.ittc.ukans.edu/kurt/, 2004.

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

[4] Gautam Thaker et. al. Implementation Experience with OMG's SCIOP Mapping. In *Proceedings of the 5th International Symposium on Distributed Objects and Applications*, Nov. 2003.

[5] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. Wang. Applying Model-Integrated Computing to Component Middleware and Enterprise Applications. *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*, 45(10), Oct. 2002.

[6] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, Oct. 1997. ACM.

[7] Institute for Software Integrated Systems. Component Synthesis using Model Integrated Computing (CoSMIC). www.dre.vanderbilt.edu/cosmic, Vanderbilt University.

[8] A. S. Krishna, D. C. Schmidt, A. Porter, A. Memon, and D. Sevilla-Ruiz. Improving the Quality of Performance-intensive Software via Model-integrated Distributed Continuous Quality Assurance. In *Proceedings of the 8th International Conference on Software Reuse*, Madrid, Spain, July 2004. ACM/IEEE.

[9] A. S. Krishna, C. Yilmaz, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan. Preserving Distributed Systems Critical Properties: a Model-Driven Approach. *IEEE Software special issue on Persistent Software Attributes*, November/December 2004.

[10] L. M. A. T. Labs. ATL QoS Home Page. www.atl.external.lmco.com/projects/QoS/, 2002.

[11] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

[12] MICO. The MICO CORBA Component Project. www.fpx.de/MicoCCM/, 2000.

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

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

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

[16] Object Management Group. *Data Distribution Service for Real-Time Systems Specification*, 1.0 edition, Mar. 2003.

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

[18] Qedo. QoS Enabled Distributed Objects. qedo.berlios.de, 2002.

[19] Ruslan Shevchenko. CORBAConf: A Tool for Providing Autoconf Support for CORBA. corbaconf.kiev.ua/, 2000.

[20] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

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

[22] D. C. Sharp, E. Pla, and K. R. Lueck. Evaluating real-time java for mission-critical large-scale embedded systems. In G. Bollella, editor, *Proceedings of the 9th IEEE Real-Time Technology and Applications Symposium*, pages 30–37, Washington D.C, 2003.

[23] StarCCM. StarCCM. starccm.sourceforge.net, 2003.

[24] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, Apr. 1997.

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

[26] P. Tuma and A. Buble. Open CORBA Benchmarking. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2001.

[27] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian. Configuring Real-time Aspects in Component Middleware. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus, Oct. 2004.

[28] 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*, 27(2):45–54, mar 2003.

[29] N. Wang, D. C. Schmidt, and S. Vinoski. Collocation Optimizations for CORBA. *C++ Report*, 11(10):47–52, November/December 1999.

[30] B. White and J. L. et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.