

# Empirically Evaluating CORBA Component Model Implementations

Arvind S. Krishna,  
Jaiganesh  
Balasubramanian,  
Aniruddha Gokhale and  
Douglas C. Schmidt  
Electrical Engineering and  
Computer Science  
Vanderbilt University  
Nashville, TN, 37235, USA  
{arvindk, jai, gokhale,  
schmidt}@dre.vanderbilt.edu

Diego Sevilla  
Department of Computer  
Engineering  
University of Murcia  
Spain  
dsevilla@ditec.um.es

Gautam Thaker  
Lockheed Martin Advanced  
Technology Labs  
Cherry Hill, NJ 08002, USA  
gthaker@atl.lmco.com

## ABSTRACT

Commercial off-the-shelf (COTS) middleware is increasingly used to develop distributed real-time and embedded (DRE) systems. DRE systems are increasingly combined using wireless and wireline networks to form “systems of systems” having multiple quality of service (QoS) requirements. Conventional COTS middleware does not facilitate the separation of QoS policies from application functionality, making it hard to configure and validate complex DRE applications. Component-based middleware addresses limitations of COTS middleware by establishing standards for implementing, packaging, assembling, and deploying component implementations. There has been little systematic empirical study of the performance aspects of CORBA Component Model (CCM) implementations, however, particularly in the context of DRE systems. This paper therefore provides three contributions to the study of component-based middleware. First, we identify the potential performance bottlenecks in CCM implementations. Second, we describe our benchmarking suite to evaluate the overhead of CCM implementations. Third, we develop criteria to compare different CCM implementations using metrics such as latency, throughput, and other performance overheads. Our preliminary results underscore the importance of applying a range of metrics to quantifying CCM implementations effectively.

## 1. INTRODUCTION

Distributed Real-time and Embedded (DRE) systems are increasingly becoming 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). Increasingly DRE systems are used for a wide range of applications where several systems are in-

terconnected using high-speed networks to form system of systems. Such systems possess stringent quality of service (QoS) constraints, such as bandwidth, latency, jitter and dependability requirements. Hence one of the most challenging requirements for these new and planned DRE systems involves supporting multiple quality of service (QoS) properties, such as predictable latency/jitter, throughput guarantees, scalability, 24/7 availability, dependability, and security, that must be satisfied simultaneously in real-time. Existing distributed object computing (DOC) middleware frameworks, such as CORBA, .NET, and Java RMI, do not provide capabilities for systems developers to connect and provision all these QoS requirements in complex DRE systems.

*Component middleware* [10] 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) [6] is a standard component middleware technology that addresses limitations with earlier generations of DOC middleware. The CCM specification extends the CORBA object model to support the concept of components and establishes standards for implementing, packaging, assembling, and deploying component implementations.

Component middleware in general – and CCM in particular – are maturing technology bases that represent a paradigm shift in the way DRE systems are developed. Several implementations of CCM suited for DRE applications are now available, including (1) CIAO (Component Integrated ACE ORB) [12], (2) MICO-CCM [3] (MICO’s CCM implementation), and (3) Qedo [7]. As CCM platforms mature, it is desirable to devise well-established metrics to compare and contrast different implementations in terms of:

- *Suitability, i.e.*, how suitable is the implementation for DRE applications in a particular domain, such as avionics, total ship computing, or telecom.
- *Quality, e.g.*, how good the implementation is, *e.g.*, in the DRE domain do the implementations provide predictable performance and consume minimal time/space resources.
- *Correctness, i.e.*, does the implementation conform to the specification, *e.g.*, does a CCM implementation meet the portability and interoperability requirements defined by the CCM specification.

Earlier efforts, such as the Open CORBA Benchmarking project [11]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

and Middleware Comparator [4], have focused on metrics to compare CORBA middleware. Our work, enhances these efforts by focusing on a previously unexplored dimension: *designing metrics to compare CCM implementation quality and suitability for DRE applications*. To quantify these comparisons in a systematic manner we are developing an open-source benchmarking suite called **CCMPerf**, focusing on *Black-box* and *white-box* metrics using criteria such as latency, throughput, and footprint measures. These metrics are then used to develop benchmarking experiments that can be categorized into:

- *Infrastructure* specific tests that quantify the overhead of CCM based applications over normal CORBA applications,
- *Services* specific tests that quantify the suitability of using different common CORBA services implementations in component middleware and
- *Domain* specific tests that quantify the suitability of CCM implementations in a particular application domain having specific requirements, such as static linking and deployment of components in the Boeing Bold Stroke avionics mission computing architecture [8].

The remainder of this paper is organized as follows: Section 2 provides an overview of CCM describing its run-time architecture; Section 3 describes the challenges involved in developing a benchmarking suite for CCM, outlines how these challenges are addressed in **CCMPerf** and illustrates the experiments in our benchmarking suite; and Section 4 presents concluding remarks and future work.

## 2. OVERVIEW OF CCM

The CORBA Component Model (CCM) is designed to address the limitations with distributed object computing middleware [1]. Figure 1 shows an overview of the layered architecture of the CCM model. *Components* are the implementation entities that export

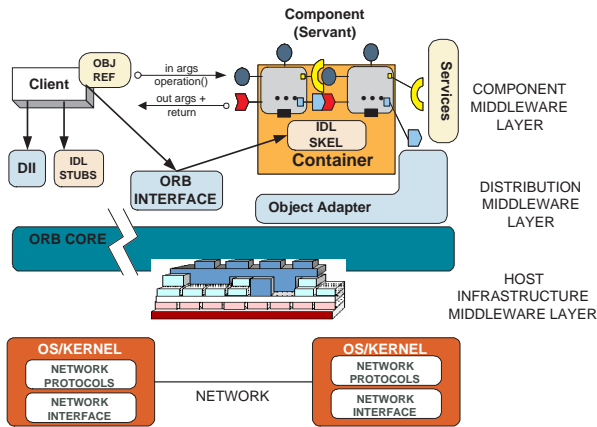


Figure 1: Layered CCM Architecture

a set of interfaces usable by clients. Components can also express their intention to collaborate with other entities by defining interfaces called *ports*. *Ports* express a component’s intent to collaborate with other components. There are several types of ports in CCM, 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. A *container* provides the

run-time environment for a component that provides various pre-defined hooks and strategies, such as persistence, event notification, transaction, and security, to the component it manages. Each container manages one type of component and is responsible for initializing this component and connecting it to other components and ORB services. Developer-specified metadata can be used to instruct the CCM deployment mechanism how to create these containers.

In addition to the building blocks outlined above, the CCM also standardizes component implementation, packaging, and deployment. The CCM Component Implementation Framework (CIF) helps generate the component implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL). CCM can be therefore be visualized as a layer that sits atop an ORB and leverages ORB functionality (such as connection management, data transfer, (de) marshaling of messages, and event/message demultiplexing), as well as higher-level CORBA services (such as Load Balancing, Transaction, Security, and Persistence).

## 3. BENCHMARKING CCM IMPLEMENTATIONS

This section describes the challenges involved in developing a benchmarking suite for CCM, outlines how these challenges are addressed and illustrates the experiments in **CCMPerf**. The goals of **CCMPerf** are to create comprehensive benchmarks that allow users and CCM developers to:

1. Evaluate the overhead CCM implementations impose above and beyond earlier-generation CORBA implementations that are based on a conventional distributed object model
2. Apply our benchmarks to 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, and
4. Develop a framework that will automate benchmark tests and facilitate the seamless integration of new benchmarks.

### 3.1 Benchmarking Challenges and their Resolutions

During the design of **CCMPerf** we encountered several challenges described below.

#### 3.1.0.1 Heterogeneity in implementations.

Our first challenge in developing a CCM benchmarking framework stemmed from the heterogeneity in the tools and mechanisms used in different CCM implementations. In particular, CCM implementations differ in:

- The header files that must be included by each implementation, which are not standardized by the OMG. Moreover, the process of obtaining the generated files (*e.g.*, the compilation chain for the different descriptor files used by the CCM) is often specific to each ORB and its CCM implementation.
- The run-time configuration options, *e.g.*, number of threads, logging levels, and locks, that can be enabled to fine tune different CCM implementations are implementation-specific.
- The level of conformance to CCM features, such as automation of component assembly, which are only provided by certain implementations, and
- The level of maturity of the implementations as a whole, *i.e.*, to what extent does the implementation conform to the CCM specification.

### 3.1.0.2 *Difference in implementation quality.*

CCM implementations differ in the data structures and algorithms they use, which affects the QoS they deliver to DRE applications. Evaluating these quality differences necessitates instrumentation of the code within the ORB/CCM implementation. This presents the following benchmarking challenges:

- A thorough understanding of CCM implementations is needed to instrument CCM middleware with probes that measure performance accurately. However, there is no body of knowledge to identify the critical features within the CCM layer in which instrumentation points need to be added, and
- CCM implementations are layered architectures. Thus, it is necessary to isolate certain layers to independently observe the influence of each layer. However, configuration options influence the presence/absence of layers making it difficult to identify the sequence of steps involved in the benchmark.

In CCMPeef we address each of the challenges outlined above as follows:

- To overcome CCM implementation heterogeneity we are developing a set of scripts to configure and run the benchmark tests. These scripts automatically generate specific code and project build file for each implementation.
- To ensure equivalent configurations, we provide automated scripts to configure and run each test.
- 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 Boeing's custom Component Model described in Section 3.2.3.
- To ensure consistent hardware and OS configurations, our tests are run using EMULab [13] and Lockheed Martin Advanced Technology Lab's (ATL) middleware comparator framework [4]. These platforms support systematic testing conditions that enable apples-to-apples comparisons of performance differences between CCM implementations. ATL also allows experiment data to be accessed readily from a web interface<sup>1</sup>.

## 3.2 Benchmark Design

The benchmarking experiments in CCMPeef focus on the following metrics:

### 3.2.0.3 *Black-box Metrics.*

Black-box metrics do not instrument the software internals when evaluating the performance tests. In our case, each CCM implementation was benchmarked end-to-end without knowledge of its internal structure. Moreover, our black-box benchmarks only use standard operations published in the CCM interfaces and do not modify or restructure the CCM ORB internals. Below, we describe the performance metrics:

- *Round-trip latency*, which measures the response time for a CCM implementation.
- *Throughput*, which compares the number of events per second processed at the component server.
- *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/across processes.

<sup>1</sup>More information is available at <http://www.atl.external.lmco.com/projects/QoS/>

- *Data copying overhead*, which compares the variation in response time with an increase in request size to check if the CCM implementation incurs additional buffer copying.
- *Footprint*, which measures both the static and dynamic footprint of the CCM implementation to determine whether they are suited for memory-constrained DRE systems.

Each of these metrics are measured in (1) single-threaded and (2) multi-threaded mode on both servers and clients.

### 3.2.0.4 *White-box Metrics.*

White-box metrics are a performance evaluation technique that employ explicit knowledge of software internals to select and analyze the benchmark data. Unlike black-box metrics, white-box metrics evaluate performance by instrumenting the software internals with probes. The metrics quantify the following aspects of performance:

- *Functional path analysis*, which identify stages/layers that are above the ORB and add instrumentation points to obtain the time spent in those layers. Moreover, jitter analysis (*i.e.*, measuring the variation in the time spent in the layers) can be achieved by isolating layers.
- *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 times*, which measures the time required for interrupting the currently running thread and switching to another thread in multi-threaded implementations.

The benchmarking experiments in CCMPeef can be categorized into:

### 3.2.1 *Infrastructure-specific benchmarks*

Each CCM implementation sits on a CORBA ORB, as shown in Figure 1, which manages various network programming tasks, such as connection management, data transfer, (de)marshaling, demultiplexing, and concurrency. Every CCM implementation adds some overhead in addition to the underlying CORBA ORB, as explained in Section 2. These benchmarks leverage black-box and white box metrics that measure various aspects of this 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 normal CORBA. End-users can apply these metrics to select CCM implementations that can meet their end-to-end QoS requirements. Moreover, these metrics can also benefit users transitioning from non-component middleware to component-based middleware to contrast measures obtained using component and non-component middleware.

### 3.2.2 *Services-specific benchmarks*

Services-specific benchmarks help in comparing the performance of various implementation choices available for integrating services within the CCM containers.

CCM leverages many standard services and features, as described in Section 2. CCM implementations are free to use the standard CORBA service specifications or use a customized implementation of the service. For example, the component implementations can specify the event sources and/or sinks through their interface definitions, however, it is left to the particular CCM implementation to choose the mode of event distribution. Implementations are free to choose between making a direct invocation on the target components or use publish-subscribe mechanism to push events. If the

CCM implementations use a publish-subscribe model, they could use the standard CORBA Event channel [5] or use a customized implementation like the Real-time Event Channel [2]. Further, the characteristics of the application domain influence the target implementation used. For example, in the DRE domain, using a Real-time Event channel may be more suitable than the standard CORBA event service implementation. Thus, it is necessary to design benchmarking tests, which will use the black-box and white-box experiments to empirically compare and contrast the implementation choices to determine the suitability aspect.

### 3.2.3 Domain-specific benchmarks

Domain-specific benchmarks include black-box and white-box experiments conducted for important use cases that occur in certain domains, such as Boeing's Bold Stroke Architecture [9] for the DRE domain. The purpose of these experiments is to identify if 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 component-based architecture has several use cases with timing constraints, *e.g.*, their architecture requires that total start up time for the system be under 2 seconds. Currently, Boeing is using its own proprietary component model. Our first target is therefore to test the suitability of COTS-based CCM implementations using the Boeing scenarios to help DRE domain experts compare CCM implementations.

Our first step in developing scenario-based benchmarking involved running experiments with CIAO using the Bold Stroke as the target platform. The tests focused on specific requirements in Bold Stroke, such as total start-up time, throughput, and response time in collocated mode. In Bold Stroke, several components are on the same processor board, *i.e.*, they are co-located instead of being distributed. Hence communication between these components need not incur overheads of marshaling/de-marshaling and other network programming tasks. Other domain-specific tests, such as total component assembly time given a topology and component deployment time, can be integrated into our benchmarking framework.

## 4. CONCLUDING REMARKS

Component middleware in general and CCM in particular are emerging fields of study. Several initiatives are underway to develop both commercial and research implementations of CCM. There is not yet substantial body of knowledge, however, that systematically describes how to develop metrics for measuring correctness, suitability, and quality of CCM implementations. This paper discusses the challenges present in benchmarking CCM implementations and describes the design of CCMPeRF, an open-source benchmarking suite for CCM we are developing. CCMPeRF provides black box, white box, and scenario-based benchmarks to qualitatively and quantitatively compare CCM implementations. Currently, our benchmarking suite includes experiments for black box metrics such as latency, throughput and jitter. These experiments can be performed on CIAO and MICO-CCM. Our future work on CCMPeRF would involve completing the white-box and scenario based benchmarks and enhancing our suite to include other CCM implementations such as Qedo. CCMPeRF available for download from [deuce.doc.wustl.edu/Download.html](http://deuce.doc.wustl.edu/Download.html).

## 5. REFERENCES

- [1] 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.
- [2] 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.
- [3] M. is CORBA. The mico corba component project. <http://www.fpx.de/MicoCCM/>, 2000.
- [4] L. M. A. T. Labs. Atl qos home page'. [www.atl.external.lmco.com/projects/QoS/](http://www.atl.external.lmco.com/projects/QoS/), 2002.
- [5] Object Management Group. *Event Service Specification Version 1.1*, OMG Document formal/01-03-01 edition, Mar. 2001.
- [6] Object Management Group. *CORBA Components*, OMG Document formal/2002-06-65 edition, June 2002.
- [7] Qedo. Qos enabled distributed objects. <http://qedo.berlios.de>, 2002.
- [8] D. C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.
- [9] 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 9<sup>th</sup> IEEE Real-Time Technology and Applications Symposium*, pages 30–37, Washington D.C., 2003.
- [10] C. Szyperski. *Component Software—Beyond Object-Oriented Programming*. Addison-Wesley, Santa Fe, NM, 1998.
- [11] P. Tuma and A. Buble. Open corba benchmarking. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2001.
- [12] 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.
- [13] B. White and J. L. et al. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.