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

Arvind, can you please fill in here where this paper will appear?!

ABSTRACT

Commercial off-the-shelf (COTS) middleware is increasingly used to develop distributed real-time and embedded (DRE) systems. DRE systems are themselves increasingly combined using wireless and wireline networks to form "systems of systems" that have diverse quality of service (QoS) requirements. Conventional COTS middleware does not facilitate the separation of QoS policies from application functionality, which makes 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 characteristics of component middleware implementations in the context of DRE systems. This paper therefore provides three contributions to the study of component-based middleware. First, we describe the challenges involved in benchmarking different CORBA Component Model (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. Our preliminary results underscore the importance of applying a range of metrics to quantify 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). DRE systems are increasingly used for a wide range of applications where multiple systems are interconnected using wireless and wireline networks to form system of systems. Such systems possess stringent quality of service (QoS) constraints, such as bandwidth, latency, jitter and dependability requirements. A challenge requirement for these new and planned DRE systems therefore involves supporting a diverse set of QoS properties, such as predictable latency/jitter, throughput guarantees, scalability, 24x7 availability, dependability, and security, that must be satisfied simultaneously in real-time. Conventional distributed object computing (DOC) middleware frameworks, such as DCOM and Java RMI, do not provide capabilities for developers and end-users to specify and enforce these QoS requirements simultaneously in complex DRE systems.

Component middleware [11] 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) [8] is a standard component middleware technology that addresses limitations with earlier versions of CORBA middleware based on the DOC model. 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 a maturing technology base that represents a paradigm shift in the way DRE systems are developed. Several implementations of CCM are now available, including (1) CIAO (Component Integrated ACE ORB) [13], (2) MICO-CCM [4] (MICO's CCM implementation), and (3) Qedo [9]. As the CCM platforms mature and become suitable for DRE it is desirable to devise a standard set of metrics to compare and contrast different implementations in terms of their:

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

Earlier efforts, such as the Open CORBA Benchmarking project [12] and Middleware Comparator [5], have focused on metrics to compare middleware based on the DOC version of CORBA. Our work enhances these efforts by focusing on a previously unexplored dimension: *metrics to compare CCM implementation quality and systematic benchmarking experiments to determine the suitability of those implementations for representative DRE applications*. To quantify these comparisons in a systematic manner we are developing an open-source benchmarking suite called CCMPerf that focuses 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 partitioned into the follow categories:

- *Distribution middleware* tests that quantify the overhead of CCM-based applications relative to applications based on versions of CORBA that do not support CCM capabilities.
- *Common middleware services* tests that quantify the suitability of using different implementations of CORBA services, such as the Real-time Event [7] and Notification Services [6].
- *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 the Boeing Bold Stroke avionics mission computing architecture [10].

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 earlier versions of CORBA middleware that supported a distributed object computing (DOC) model [2]. Figure 1 shows an overview of the layered architecture of the CCM model. *Components* in CCM are the implementation entities that

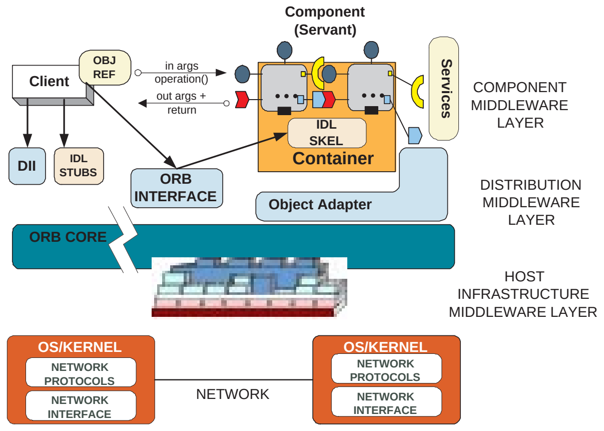

Figure 1: Layered CCM Architecture

export 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* in CCM provides the run-time environment for one or more components that provides various pre-defined hooks and strategies, such as persistence, event notification, transaction, and security, to the component(s) it manages. Each container manages one type of component and is responsible for initializing instances of this component type and connecting them to other components and common middleware services. Developer-specified metadata expressed in XML can be used to instruct the CCM deployment mechanism how to control the lifetime of these containers and the components they manage. In addition to the building blocks outlined above, the CCM also standardizes component implementation, packaging, and deployment mechanisms. The CCM Component Implementation Framework (CIF) helps generate the component implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL).

As shown in Figure 1, CCM is 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). CCM-based applications therefore can incur additional overhead when compared to their CORBA 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 occurs in the critical path of every request/response it may incur non-trivial overhead.

3. OVERVIEW & DESIGN OF CCMPerf

This section describes the challenges involved in developing a benchmarking suite for CCM, outlines how these challenges are addressed, and illustrates the three experimentation categories 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 CORBA implementations that are based on the earlier-generation DOC 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.
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 the following challenges:

- Heterogeneity in implementations
- Difference in implementation quality
- Difference in the domain of application and
- Heterogeneity in hardware & software platforms.

We describe each of these challenges below and then outline how are addressing these challenges in CCMPerf.

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. For example, CCM implementations differ in terms of:

- 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 level of conformance to CCM features, such as automation of component assembly, which are only provided by certain CCM implementations.
- The overall level of maturity of the CCM implementations, *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 can deliver to DRE applications. Evaluating these quality differences necessitates instrumentation of the code within the ORB/CCM implementation, which presents the following benchmarking challenges:

- A thorough understanding of CCM implementations is needed to instrument CCM middleware with probes that measure its performance accurately. However, there is no systematic body of knowledge yet that identifies the critical features within the CCM layer in which 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. ORB-specific configuration options influence the presence/absence of these layers, however, which make it hard to identify the set of steps within each layer for every combination of the configuration options.

3.1.0.3 *Differences in software configuration options.*

CCM implementations differ in the knobs that they provide to maximize performance. For example, the run-time configuration options, such as, number of threads, logging levels, and locks, that can be enabled to fine tune different CCM implementations are implementation-specific, which lead to the following challenges:

- The same set of configuration options many not be available each of the CCM implementations. For example, CIAO allows applications to configure the type of locks used within the ORB, whereas MICO does not provide such a option.
- An implementation could be optimized for a given set of configurations, yet perform poorly for other configurations. For example, MICO is optimized for single-threaded applications and performs poorly in multi-threaded configurations.

Similarly, a change in the underlying hardware configuration (such as processor speed of the computer and/or network bandwidth changes across experiments) prevents valid comparisons of the results.

3.1.0.4 *Differences in the domain of application.*

Each CCM implementation may be tailored for a particular application domain. For example, the CIAO CCM implementation is tailored towards the DRE domain. Conversely, MICO-CCM is targeted for general-purpose distributed computing. These domains of applicability pose the following challenges:

- Use cases may change across domains. For example, some applications in the DRE domain may require that the total startup time be under two seconds. Component middleware catering to this domain may need to be optimized to meet this requirement, whereas middleware for the enterprise domain might not not.
- Certain metrics (such as predictable end-to-end latency and static/dynamic memory footprint) are important in the DRE domain, but often less in the enterprise domain.

CCMPerf addresses 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 CCMPerf tests. These scripts automatically generate CCM platform-specific code and project build files 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 [14] and Lockheed Martin Advanced Technology Lab's (ATL) middleware comparator framework [5]. These testbeds support systematic testing conditions that enable equivalent comparisons of performance differences between CCM implementations. ATL also allows experiment data to be accessed readily from a convenient web interface[1].

3.2 CCMPerf Benchmark Design

The benchmarking tests in CCMPerf focus on the following metrics:

3.2.0.5 *Black-box metrics.*

Black-box metrics do not instrument the software internals when evaluating the performance tests. We benchmarked each CCM implementation end-to-end without knowledge of its internal structure using standard operations published in the CCM interfaces and did not modify or restructure the CCM ORB internals. We outline our black-box performance metrics below:

- *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 (1) the 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.

[1]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 determine whether a CCM implementation incurs excessive buffer copying.
- *Footprint*, which measures the static and dynamic footprint of a CCM implementation to determine whether it is suitable for memory-constrained DRE systems.

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

3.2.0.6 White-box metrics.

White-box metrics are a performance evaluation technique 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. We outline our white-box performance metrics below:

- *Functional path analysis,* which identifies CCM layers that are above the ORB and adds instrumentation points to determine the time spent in those layers. Moreover, we 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 times*, which measure the time required to interrupt the currently running thread and switch to another thread in multi-threaded implementations.

The benchmarking tests in CCMPerf can be categorized into the general areas discussed below.

3.2.1 Distribution Middleware Benchmarks

Each CCM implementation sits on a CORBA ORB, as shown in Figure 1. The ORB manages various network programming tasks, such as connection management, data transfer, (de)marshaling, demultiplexing, and concurrency. Every CCM implementation adds some overhead to the underlying CORBA ORB, as explained in Section 2. CCMPerf's distribution-specific benchmarks employ black-box and white box metrics that measure various aspects of 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 DOC support. Application developers and 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 who are considering a move from DOC middleware to component middleware so they can quantify the pros and cons of such a transition.

3.2.2 Services-specific benchmarks

Service-specific benchmarks help to compare the performance of various implementation choices applied to integrate common middleware services within the 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.

For example, component implementations can specify the event sources and/or sinks through their interface definitions. It is left to a particular CCM implementation, however, to choose the mode of event distribution. Implementations are free to choose between making a direct invocation on the target components or to use publish/subscribe mechanisms to push events. If CCM implementations use a publish-subscribe model, they can use the standard CORBA event channel [7] or use a customized implementation (such as a real-time event channel [3]). To benchmark the type of scenario described above where the container uses the event channel to publish events, CCMPerf measures the overhead introduced by extra (de)marshalling and indirection costs incurred within the container for publishing the events to the all the receivers.

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 **??** to empirically compare and contrast the implementation choices for a particular application domain.

3.2.3 Domain-specific benchmarks

Domain-specific benchmarks include black-box and white-box tests conducted for key use cases that occur in certain domains, such as Boeing's Bold Stroke platform [1] that is 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 component-based architecture has several uses cases with stringent timing constraints, *e.g.,* their architecture requires that total start up time for the entire system of ∼3,000 components be under two seconds. Since Boeing is currently using a proprietary component model to achieve these timing goals we are using the Boeing scenarios to test the suitability of COTS-based CCM implementations to meet the same requirements.

4. CONCLUDING REMARKS

Component middleware in general and QoS-enabled CCM implementations in particular are important R&D topics. Several initiatives are underway to develop both commercial and research implementations of CCM. However, there is not yet a substantial body of knowledge that describes how to develop metrics that systematically measure correctness, suitability, and quality of CCM implementations. This paper discusses the challenges present in benchmarking CCM implementations and describes the design of CCMPerf, which is an open-source benchmarking suite for CCM we are developing. CCMPerf provides distribution, services and scenario-based benchmarks to quantitatively and qualitatively compare CCM implementations using black-box, and white-box, metrics. CCM implementations. The CCMPerf benchmarking suite currently includes experiments for black-box metrics, such as latency, throughput, and jitter, using the CIAO and MICO-CCM implementations of CCM.

Our first step in developing scenario-based benchmarking involved running experiments with CIAO using Boeing's Bold Stroke avoinics mission computing platform 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 rather than distributed. Communication between these components therefore need not incur the overhead of (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 CCMPerf.

Our future work on CCMPerf will focus on completing the white-box and scenario-based benchmarks and enhancing our test suite to include other CCM implementations, such as Qedo and K2. CCMPerf is available for download from `deuce.doc.wustl.edu/Download.html`.

5. REFERENCES

[1] K. R. L. David C. Sharp, Edward Pla and R. J. H. II. Evaluating real-time java for mission-critical large-scale embedded systems. In G. Bollella, editor, *Proceedings of the* 9^{th} *IEEE Real-Time Technology and Applications Symposium*, pages 30–37, Washington D.C, 2003.

[2] 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.

[3] 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.

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

[5] L. M. A. T. Labs. Atl qos home page'. `www.atl.external.lmco.com/projects/QoS/`, 2002.

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

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

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

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

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

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

[12] P. Tuma and A. Buble. Open corba benchmarking. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, 2001.

[13] 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.

[14] 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.