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, do not facilitate the separation of QoS policies from application functionality, which makes 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 three contributions to the study of CCM for DRE systems. 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. We demonstrate* CCMPerf *by using it to collect metrics from a CCM implementation designed for DRE applications.*

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 [19] 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) [10] is a standard component middleware technology that addresses limitations with earlier versions of CORBA 2.x middleware based on the DOC model. 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. In particular, they provide higher-level capabilities for developers and end-users of DRE systems to specify and enforce QoS requirements in complex DRE systems. Several implementations of CCM are now available, including the Component Integrated ACE ORB (CIAO) [21], MicoCCM [7], Qedo [12], and StarCCM [17]. 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 [20] and Middleware Comparator [6] 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 [9] and Notification Services [8].
- *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 [15].

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) criteria for comparing different CCM implementations using key black-box and white-box metrics, (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) [21], which is an open-source[2] implementation of CCM built upon the Real-time CORBA infrastructure of *The ACE ORB* (TAO) [14]. 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 CORBA 3.x CCM capabilities does 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 some sample 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 [3]. Figure 1 depicts the key elements in the architecture of CCM. The remainder of this section describes each of these CCM elements.

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*,

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

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

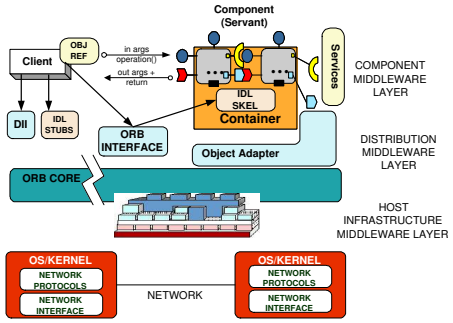

Figure 1. Elements in the CCM Architecture

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*. 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).

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).

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 in this figure, a deployment application creates an Assembly manager in charge of 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

Sidebar 1: Separating Configuration Concerns in CCM

Configuration of components in CCM can be performed at different levels of abstraction and involves different tradeoffs. CCM uses XML-based descriptors to configure components. Each descriptor exposes different aspects of a component-based system. This sidebar describes the different types of descriptors defined by the CCM Deployment and Configuration specification [11] and explains how they help separate component configuration concerns:

- **Component Interface Descriptor (.ccd)**, which describes the interface, ports, and properties of a single component.
- **Implementation Artifact Descriptor (.iad)**, which describes the implementation artifacts (*e.g.*, DLLs and OS platform) associated with a single component.
- **Component Package Descriptor (.cpd)**, which describes multiple alternative implementations of a single component.
- **Component Implementation Descriptor (.cid)**, which describes a specific implementation of a component interface, *i.e.*, if the implementation is monolithic or assembly-based.
- **Component Domain Descriptor (.cdd)**, which describes the composition of domains, *e.g.*, a related set of nodes, inter-connects. and bridges.
- **Component Deployment Plan (.cdp)**, which describes the artifacts (*e.g.*, component implementation and target domain information) for deployment and provides information on how to create component instances from these artifacts.

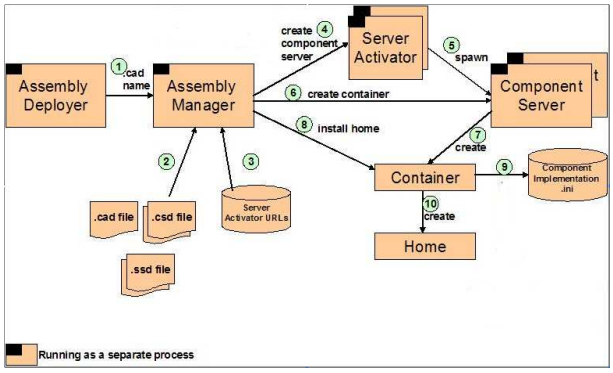

Figure 2. Interaction between CCM entities

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 to compare CCM implementations.

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 CCMPerfto 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 they are resolved 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 [13] 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.
- 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 right options to be used in different implementations to get equivalent CCM configurations. To ensure consistent hardware and OS configurations, CCMPerf tests are run using EMULab [22] and Lockheed Martin Advanced Technology Lab's (ATL) Middleware Comparator framework [6]. 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 [16]. Component middleware catering to the DRE domain often needs to be optimized to meet this requirement, whereas middleware for general-purpose distributed computing may not.
- *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 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 CORBA 3.x CCM implementation beyond the CORBA 2.x DOC model support. 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 Event Service [9] or use a customized implementation (such as a Real-time Event Service [4]).

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 [16] 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 real-time protocols such as the Stream Control Transmission Protocol [2], standardized by the OMG. Although the CCM specification itself does 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.

4. Empirically Evaluating CIAO using CCMPerf

This section presents the results of distribution middleware benchmarks that use *black-box* tests to quantify the overhead of CIAO above and beyond the underlying TAO ORB. The experiment was performed on an Intel Pentium IV 2.0 Ghz processor with 512 MB of main memory using TAO version 1.3.5 and CIAO 0.3.5 compiled with the Timesys `g++` compiler version 3.2.2 and executed on the Linux 2.4.21-timesys-4.1.147 kernel. The experiment was run in the Timesys Linux real-time scheduling class and a sample size of 250,000 data points was used for the result analysis.

Although both CIAO and TAO support a variety of configuration options, we made the following assumptions for this analysis: (1) native exception handling was enabled, (2) logging was disabled, (3) the ORB was config-

ured to run in single-threaded mode, (4) normal CORBA servants were used, *i.e.*, they inherited from `org.omg.PortableServer.Servant`, (5) we did not consider DII and DSI, and (6) no proprietary policies were associated with the ORB. These assumptions are consistent with ORB usage in DRE applications [15, 2].

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), (3) a CCM component (playing role of server) interacting with a CORBA client, and (4) a CCM component interacting with another CCM component (playing both client and server roles). These four use cases represent all combinations of mixing and matching a CCM component with CORBA clients/servers and represent how DRE applications will most likely apply CCM implementations. For each of the above interaction scenarios, we get 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. As shown in

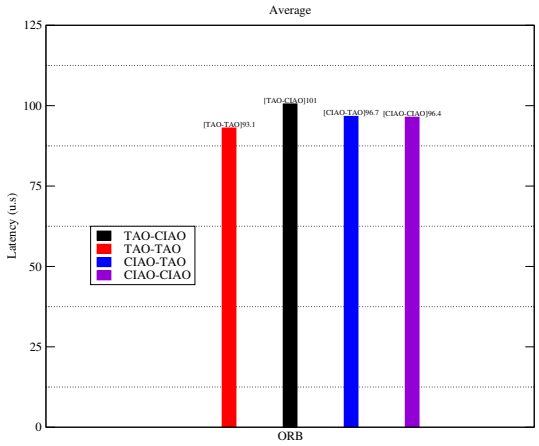

Figure 3. Round Trip latency Analysis

Figure 3, average latency for all the four case is nearly the same with TAO-TAO scenario having the minimum average latency of $\sim$93.07 μsecs and CIAO-CIAO scenario having maximum latency of $\sim$100.54 μsecs. The average case overhead added by CCM over CORBA is thus $\sim$8 μsecs. These results indicate that CIAO's CORBA 3.x CCM capabilities do not add significant overhead above and beyond its underlying TAO CORBA 2.x implementation.

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 [20], 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.

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) [6] Middleware Comparator, which evaluates a range of middleware layers, including common middleware services via an easily accessible Web interface. Their method has been to use identical test conditions (application, hardware, etc.), which permits comparisons that can reveal performance differences between various systems.

6. Concluding Remarks

Component middleware in general and QoS-enabled CORBA Component Model (CCM) implementations in particular are important technologies. 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.

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. We are also integrating our model-based synthesis of benchmarking experiments into CoSMIC [5], which is an integrated toolsuite for modeling design and runtime aspects of QoS-enabled component middleware. CoSMIC's model-based [18] 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* 8^{th} *IEEE Real-Time Technology and Applications Symposium*, San Jose, Sept. 2002. IEEE.

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

[3] 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.

[4] 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.

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

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

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

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

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

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

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

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

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

[14] D. C. Schmidt and et al. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

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

[16] 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^{th} *IEEE Real-Time Technology and Applications Symposium*, pages 30–37, Washington D.C, 2003.

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

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

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

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

[21] 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.

[22] 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.