Optimizing the CORBA Component Model for High-performance and Real-time Applications

Nanbor Wang and David Levine
{nanbor,levine}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA

Douglas C. Schmidt
schmidt@uci.edu
Electrical and Computer Engineering Dept.
University of California, Irvine, CA 92697*

This work-in-progress paper has been submitted to the Middleware 2000 Conference, IFIP/ACM, Palisades, New York, April 3-7, 2000.

Abstract

With the recent adoption of the CORBA component model (CCM), application programmers now have a standard way to implement, manage, configure, and deploy components that implement and integrate CORBA services. The CCM standard not only enables greater software reuse for servers, it also provides greater flexibility for dynamic configuration of CORBA applications. Thus, CCM appears to be well-suited for general-purpose client/server applications.

Due to the complexity of the standard and relative immaturity of implementations, however, CCM is not yet appropriate for mission-critical applications with high-performance and real-time quality-of-server (QoS) requirements. Therefore, we have begun a project to identify, prototype, benchmark, optimize, and deploy the key patterns and framework components necessary to mature the CCM standard so it can be applied successfully to mission-critical applications with stringent QoS requirements.

There are two contributions of our research project. First, we are identifying the performance bottlenecks and other obstacles that impede the use of CCM for high-performance and real-time applications. Second, we are demonstrating the effectiveness of our methodology of applying optimization principle patterns to alleviate these obstacles.

1 Introduction

Research background: The demand for distributed object computing (DOC) middleware, such as OMG's Common Object Request Broker Architecture (CORBA) [1], is growing rapidly as deregulation and global competition makes it increasingly hard to develop and maintain complex middleware from scratch. CORBA allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [2]. Other common DOC middleware technologies include Microsoft's Component Object Model (COM) [3] and Sun JavaSoft's Jini [4], Java RMI [5], and Enterprise Java Beans (EJB) [6].[1]

The use of CORBA as a flexible infrastructure for distributed client/server applications has grown rapidly over the past five years [7]. Until recently, however, there were no CORBA ORBs that targeted high-performance and real-time systems, which meant there CORBA was not used in these domains. In general, CORBA was perceived as inappropriate for systems that possessed stringent deterministic and statistical real-time quality of service (QoS) requirements [8].

Over the past two years, however, CORBA has become increasingly used in many embedded and real-time systems in aerospace, telecommunications, medical systems, and distributed interactive simulations. The increasing acceptance of CORBA for these types of high-performance and real-time systems stems largely from the maturations of standards [9, 10, 11], patterns [12, 13], and QoS-enabled framework components [14, 15].

Future trends and current limitations: The standardization and advanced R&D efforts on high-performance and real-time CORBA mentioned above have aided the adoption of CORBA middleware in real-time system domains. However, CORBA servers for these domains have historically been implemented in an *ad-hoc* manner due to the lack of a *component* model in the CORBA specification. This omission has prompted the OMG to specify a standard CORBA Component Model (CCM) [16], which will be incorporated into the forthcoming CORBA 3.0 specification [17].

With the advent of JavaSoft's EJB [6] and Microsoft's ActiveX [18], components are becoming the preferred way to de-

*This work was funded in part by Boeing, NSF grant NCR-9628218, DARPA contract 9701516, and Siemens.

[1] We focus on CORBA because it is an open standard. However, the patterns resulting from this project will largely generalize from CORBA to other DOC middleware component technologies, such as COM and EJB.

velop and deploy reusable core building blocks for business applications and services. In theory, the adoption of CCM will make it possible to integrate components needed to implement services and applications with less effort and greater portability. In addition, CCM will simplify the reconfiguration and replacement of existing application services by standardizing the interconnection among components and interfaces.

In practice, however, the CCM standard and implementations are as immature today as the underlying CORBA standard and ORBs were three to four years ago. Moreover, the CCM vendor community is largely focusing on the requirements of e-commerce, workflow, report generation, and other general-purpose business applications. The middleware requirements for these applications generally focus on functional interoperability, with little emphasis on assurance of or control over mission-critical QoS aspects, such as timeliness, precision, dependability, or minimal footprint [19]. As a result, it is not feasible to use off-the-shelf CCM implementations for high-performance and real-time systems.

Solution approach and expected results: To address these shortcomings, we are conducting a research project to *identify, prototype, benchmark, optimize, and deploy the key patterns and QoS-enabled framework components necessary to mature the standard CORBA Component Model so it can be applied successfully to high-performance and real-time applications*. This project focuses on aspects of the CCM specification that are critical to these types of applications. The goals of this research are to leverage our previous experience with QoS-enabled middleware [20, 19], add optimized high-performance and real-time support to TAO's CCM implementation, and transfer the results to the CORBA standardization effort.

2 Technical Rationale

2.1 Overview of CORBA Component Model

2.1.1 Background and Existing Limitations

Historically, the CORBA specification [1] has concentrated on defining *interfaces*, which define contracts between clients and servers. An interface defines how clients *view* and *access* object services provided by a server. Although this model has certain virtues, such as location transparency, it has the following limitations:

Lack of standardized servant interaction model: The CORBA specification has made little effort to define how to implement servants. Although the Portable Object Adapter (POA) specification first introduced in CORBA 2.2 [21] standardized the interactions between servants and ORBs, server developers are still responsible for determining how servants are implemented and how they interact. As a result, the lack of a standardized servant interaction model has yielded tightly coupled, *ad-hoc* servant implementations, which increase the complexity of software upgrades and reduce the reusability and flexibility of CORBA-based software.

Increased time and space overhead: The original CORBA object model treats all interfaces as client/server contracts. To maintain the interface contract and increase reusability, therefore, developers must still model a servant using a general CORBA interface, even if the service will only be used internally within a server. This constraint incurs unnecessary large memory footprints and programming complexity. In addition, it may incur unnecessary communication overhead for ORBs that do not implement collocation optimizations [22].

2.1.2 OMG Solution → the CORBA Component Model

The OMG has addressed the limitations outlined above by defining the CORBA Component Model (CCM). Figure 1 shows an overview of the run-time architecture of the CCM model. This section gives a brief overview of the CCM architecture. A more detailed overview can be found in Appendix A.

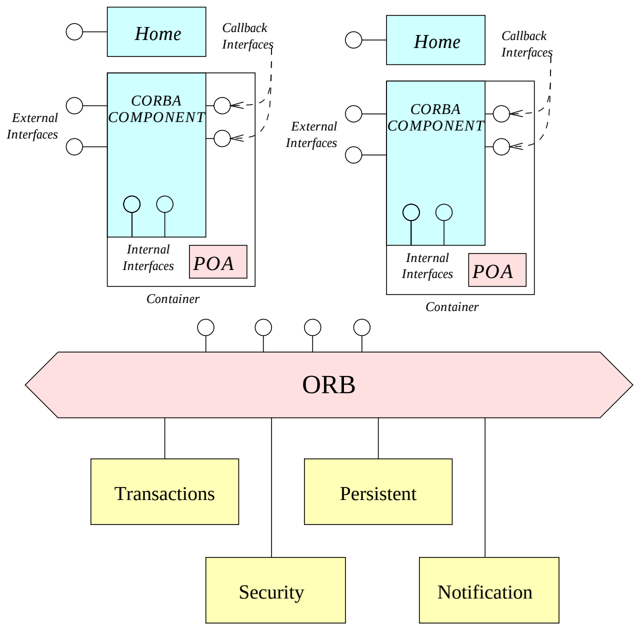

Figure 1: Overview of the CCM Run-time Architecture

Components are the implementation entities that export a set of interfaces to clients. Components support predefined interfaces called *ports* that the CCM uses to facilitate interaction among component and other ORB resources. *Ports* include *Facets*, *Receptacles*, *Event Sources/Sinks*, and *Attributes*. In addition, components can inherit from one or more *supported interfaces*, which so-called "component-unaware" clients use to access the component. Moreover, components define standard interfaces to support additional navigable interfaces, accept object references for delegating responsibilities, generate and receive events, and support dynamic configuration of components.

Home interfaces provide a factory service that manages the lifecycle for components. Moreover, a home implementation may choose to index component instances with a *primary key*, which are user-implemented classes that identify component instances and derive from `Component::PrimaryKeyBase`.

A container provides the run-time environment for a component. A container's run-time environment consists of various pre-defined hooks that provide strategies, such as persistence, event notification, transaction, and security, to the managed component. Each container manages one component and is responsible for initializing the managed component and connecting it to other components and ORB services. The CCM deployment mechanism implements the containers through developer-specified *metadata*, which instructs the CCM deployment mechanism on how to create these containers.

In addition to the building blocks outlined above, the CCM standardizes component implementation, packaging, and deployment. A Component Implementation Framework (CIF) is defined to automate the component implementation and persistent state management in a language independent way. CIF uses the Component Implementation Definition Language (CIDL) to generate component implementation skeletons. The CCM also extends the Open Software Description (OSD), which is a vocabulary of XML defined by W3C, to address component packaging and deployment requirements.

2.2 Implementing and Optimizing the CCM

Section 2.1.2 outlines the key features defined in the CCM, focusing on the overall model and how certain features interact. This section outlines how we plan to implement and optimize the CCM for high-performance and real-time applications. Section 2.2.1 outlines key optimization principle patterns [23] that can be applied to resolve design challenges arising from targeting CCM for high-performance and real-time applications. Optimization principle patterns document rules for avoiding common design and implementation problems that degrade the efficiency, scalability, and predictability of complex systems. Section 2.2.2 illustrates how these optimization principle patterns can be applied to improve key QoS aspects of specific CCM features.

2.2.1 Resolving Key CCM Design Challenges

As shown in Section 2.1.2, the CCM requires a significant number of new classes and interfaces to support its specified features. These requirements may cause problems for high-performance and real-time applications due to unnecessary time and space overhead incurred when components are collocated within the same process or machine. To build a robust CCM implementation and remove these overheads, therefore, we are applying optimization principle patterns gleaned from our previous experience [24, 12] optimizing TAO for high-performance and real-time applications. Some important optimization principle patterns include *optimizing for the common case*, *avoiding gratuitous waste*, *shifting computation in time via precomputing*, *replacing general-purpose functions with special-purpose ones*, *passing hints between layers*, *storing redundant state to speed up expensive computation*, and *using efficient data structures*.

Challenge: Enhancing component transparency:

• **Context:** A key benefit of the CCM is *component transparency*. The following transparencies are important for high-performance and real-time applications:

- *Location transparency* – References to components and objects can be passed among processes that may be distributed or collocated in different system configurations. CCM applications should not be concerned whether components are distributed or are collocated on the same process or host. In particular, CCM applications should not and cannot detect the location of an object reference and should invoke an operation using the same method regardless of where the object resides. In particular, component location should not affect the semantics of operations, such as location forwarding, concurrency and dispatch policies, `Current` state, and interceptor invocations.
- *Performance transparency* – An ORB supporting the CCM should provide different mechanisms to optimize operation invocations based on the location of the objects. However, CCM applications should be able to invoke an operation using the most efficient method available for each configuration. This must be done automatically, *i.e.*, without developer intervention, so that applications can obtain optimal performance and avoid unnecessary overhead without being reprogrammed.
- *Predictability transparency* – No matter where the object resides and what mechanism is used to invoke an operation, a CCM implementation must not incur overhead that degrades latency and increases jitter. For instance,

synchronization contention should be minimized, unnecessary dynamic memory management should be eliminated in the critical path, priority-based queuing should be used for all communication mechanisms, and priority inversion should be minimized or eliminated.

• **Problem:** When a reference is passed back to the same process, or to the same machine where the originating object resides, many ORBs still use remote stubs to invoke operations via the reference. Thus, collocated references lack performance transparency. Although it is sometimes possible to use proprietary mechanisms to override this behavior, these mechanisms hinder the locality transparency of collocated object references. Some ORBs use short-circuited object references to forward in-process collocated invocation directly to servants. However, this approach impedes the predictability transparency of collocated references because direct forwarding deviates from the semantics of the CORBA object model [22].

• **Solution $\rightarrow$ Collocation optimizations:** To improve the performance and predictability of collocated component communication, we will apply the following optimizations to TAO's CCM implementation:

- *Process-collocation* – Process-collocation improves the performance and predictability transparencies for objects that reside in the same address space with the servant, while maintaining locality transparency. To implement process-collocation, the ORB must identify the location of the component's reference without explicit application programmer intervention and without violating the policies specified by POAs and the component containers. Once the ORB determines that this reference is collocated in the same process, all operation invocations can be forwarded to a special collocation stub's method. The goal of the process-collocation is to ensure the performance of accessing in-process collocated components is comparable to accessing regular C++ components, while still providing predictability transparency in the framework.
- *Host-collocation* – For a host-collocated component, it is also necessary to identify the location of a component reference transparently. Because TAO allows applications to plug in various transport protocols into an ORB [25], we plan to implement a shared memory transport protocol for TAO. This pluggable transport will allow operations host collocated components and objects to be invoked transparently and efficiently via shared memory.

Collocation optimizations illustrate the principle patterns of (1) *avoiding gratuitous waste* by avoiding invoking collocated operations using remote stubs and (2) *replacing inefficient general-purpose operations with optimized special-purpose ones* by creating special collocated stubs for invoking collocated operations. Our previous experience [12, 22] shows that collocation can reduce the overhead of many high-performance and real-time applications significantly, *without* affecting the semantics defined by the CORBA object model.

Challenge: Enhancing component configurability and customizability:

• **Context:** As shown in Section 2.1.2, the CCM is a very large and complex specification. In particular, a substantial number of features must be implemented to support the complete specification.

• **Problem:** Many real-time applications are deployed in systems with very stringent memory limits. Often, they run in relatively stable configurations once they are deployed. Thus, much of middleware required to implement certain CCM features will be largely unused. This rarely used code may consume excessive memory, which is often a limited resource for real-time systems. When applications *do* require certain features, however, it should be possible to configure them flexibly.

• **Solution $\rightarrow$ Dynamic configuration of components:** To reduce unnecessary memory usage, while still allowing applications to use certain features when necessary, we are improving the dynamic configurability of TAO. Previous research [26] demonstrates that ORB middleware can be dynamically reconfigured at run-time by dynamically linking in the necessary components. We will take this work to the next level by addressing the following topics:

- *Configurable ORB-level middleware* – We will explore how to further partition TAO's internal structure to maximize its configurability. In particular, we will examine different configurable communication mechanisms, such as shared memory and high-speed backplanes, by extending TAO's existing pluggable protocols framework [25] to support dynamically linkable protocols and marshaling/demarshaling.
- *Configurable component and component infrastructure* – We will enable the stub and skeleton code generated by TAO's IDL compiler to be linked dynamically from shared libraries. Other CCM mandated interfaces that support the semantics of component implementations can be linked dynamically, as well. For example, a component may support several *supported interfaces*, even though only a subset will be used in certain configurations.

  In CCM, a component can be used to support another collocated component. The interfaces for these supporting component require no remote stubs and skeletons. This finer partitioning will allow CCM applications to store system implementation components in secondary storage. Thus, only those portions that are required for the proper functioning of a particular configuration will be linked dynamically, *without* losing the generality of the standard CORBA Component Model.

The use of dynamic configuration outlined above applies the optimization principle pattern of *avoid gratuitous waste* by avoiding loading in software that is not needed at run-time.

2.2.2 Implementing the CCM for High-performance and Real-time Systems

This section outlines the major areas that we will apply the optimization principle patterns and techniques identified in Section 2.2.1 to our real-time, high-performance CCM implementation.

ORB extensions: Three major extensions are required to the current ORB specification. Of them, the addition of locality constrained interfaces has the most impact on the overall performance of CORBA applications. This addition defines the `local` keyword to the IDL syntax to support locality constrained object interfaces. The `local` keyword allows programmers to define and use their own locality constrained objects to avoid unnecessary network traffic and marshaling/demarshaling operations. It is an example of the optimization principle pattern of *avoiding gratuitous waste*.

Component model: As mentioned in Section 2.1, the CCM specifies several component APIs that support core component features. These APIs allow application developers to interconnect components and objects together. Although the addition of `local` interface keyword in the CCM specification improves the performance locality constrainted components, there are still cases where a component can be used both locally and remotely. In this case, we will apply the principle patterns *avoiding gratuitous waste* and *replacing inefficient general-purpose operations with optimized special-purpose ones* by invoking methods via a special collocated object reference on a collocated component. In contrast, invoking the operation through the remote interface stub would impose unnecessary performance overhead from parameter marshaling/demarshaling and transport protocol traffic/latency.

Containers: Containers provide interconnections for managed components, as described in Section 2.1. The CCM employs several common ORB services to manage resources within containers. In general, the Object Transactions Service may not be relevant for real-time applications with deterministic QoS requirements due to the overhead associated with this service. Conversely, the Notification Service has been identified as a useful component for real-time telecommunication management applications. Therefore, we are optimizing TAO's existing real-time Events Service [27] to support key Notification Service features, such as event filtering, so it can be integrated into our CCM implementation for high-performance and real-time applications.

Packaging and deployment tools: The packaging the deployment tools defined in CCM allow software to be composed into packages (files) and be deployed using an application server. Most environments for real-time applications have very stringent memory requirements. Therefore, it is important to explore ways to minimize the extra memory footprint incurred by the component model through finer partitioning of libraries and dynamic configuration.

We will support an XML-based packaging and deployment tool and integrate the dynamic configuration support outlined in Section 2.2 with this tool. The use of XML simplifies the maintenance of packaging and deployment descriptions because it is human-readable. Moreover, because XML is extensible and supports namespaces, the standard component APIs can be extended without violating the CCM specification.

3 Related Work

The following work on middleware and component technologies is related to our project.

EJB: The Enterprise Java Bean [6] is Java's solution to the component model. It supports binary program compatibility within various Java run-time environments. Although the write-once/run-anywhere philosophy of Java simplifies component deployment, having to support all the native OS mechanisms limits Java applications' flexibility and performance. The requirement of automatic garbage collection, which can be started at anytime and run for undetermined amount of time, also makes Java unsuitable for many real-time applications. Moreover, the lack of a Java event demultiplexing mechanism forces developers to use threads to service multiple event sources, which does not scale well [28].

While the CCM is modeled closely on the EJB specification [16], the CCM is arguably more flexible. First, because CCM is based on CORBA, it can work on any platform and language. In contrast, EJB is focused on Java-only systems. Second, CCM developers are not limited to the use of threads to service multiple event services since CORBA defines a wider range of concurrency models, including reactive dispatching and thread pools. Although packaging and deploying CCM components for heterogeneous platforms may be complicated to maintain due to the number of platform/OS combinations, this is rarely a problem for real-time applications, which run on largely homogeneous environments.

Component Object Model (COM+): COM+ is Microsoft's distributed object computing architecture. It is an integration of COM, DCOM, and Microsoft Transaction Server (MTS) [29]. Unlike CORBA, which was developed for distributed object computing, COM+ was originally developed for local object access and later added extra layers for remote accessibility [3]. While Microsoft claims that COM+ is binary compatible on all platforms, it is used primarily on Microsoft platforms, none of which are real-time operating systems.[2]

Reflective ORBs: Kon and Campell [26] demonstrate that TAO can be reconfigured at run-time by dynamically linking in the required components. Although their research provides a proof-of-concept for dynamic configurable middleware framework, their research does not explore performance implications and optimizations related to component-based middleware. We expect the CCM's packaging and deployment framework will supersede their *ComponentConfigurator* and will define standard strategies and patterns for packaging components. Our proposed research on dynamic configuration will concentrate on reducing memory footprint for supporting component model, without compromising the completeness of the model.

COM interceptors: Hunt and Scott [30] described how to implement interceptors in COM. The concept they used to implement interceptors is similar to TAO's collocated stub [22], in that both use alternative stubs to masquerade as operation targets. While this concept is effective, their work was done in the context of COM. Therefore, our research will explore the effects of applying these concepts to CCM.

Active Badge System: Szymaszek, Uszok, and Zieliński built a component-oriented system using Orbix's smart proxy facility in their work on the Active Badge System [31]. In this case study, they used the smart proxies to provide the component framework and stringed the system entities with various ORB services like Event Service, Persistency Service and Security Service. Although the work is not directly related to the CCM, it identifies the key services used by the CCM and the importance of local objects and collocation optimizations that are the focal points in our research.

4 Current Status and Future Work

We have just begun work on the research described in this paper. TAO already implements some basic collocation optimizations [22] and supports dynamic configuration of TAO internal component at application startup time [32]. Many issues are still unresolved, however. The following outlines the future tasks for our research project:

[2]There are UNIX implementations of COM available, but they are not widely used and do not support the latest COM+ specification.

Task 1 – Benchmarking the existing CCM and other component technologies: This task will provide us more information on the limitations of current existing implementation of CCM. It will also help us to identify the sources of performance bottlenecks and nondeterminism. We will also benchmark similar component technologies including Java's EJB and Microsoft's DCOM to study these technologies' limitations and bottlenecks.

Task 2– Implementing the basic component support of CCM in TAO: The CCM introduces many new features to CORBA. This task will implement support for using the basic component model into TAO. It can be further divided into the following three subtasks:

- **Task 2.1:** Modify TAO and the TAO IDL compiler to support and conform with additional modification of CORBA specifications that are not component related. This includes support for local interfaces and some minor interface changes.

- **Task 2.2:** Implement the basic component support in TAO. This task requires modification to TAO IDL to generate component specific code, and to the ORB core to become component-aware.

- **Task 2.3:** Modify and improve some of TAO's ORB services, such as its Real-time Event Service, to become component-aware. This will improve the efficiency later when we add more advanced features of the CCM.

Task 3 – Apply optimizations for real-time applications: This task includes applying the optimizations mentioned in Section 2.2.1 to the modifications and additions in TAO. It involves adapting the current process-collocation optimization for the CCM implementation, implementing a share memory transfer protocol for host-collocation optimization.

Task 4 – Improve dynamic configurability of TAO: This task involves adding the XML-based packaging and deployment tool specified in the CCM specification and integrating it with the dynamic linking of ORB components and IDL compiler generated code.

Task 5 – Identify and implement the suitable concurrency strategies for the CCM for real-time applications: Key concurrency strategies must be identified and implemented to avoid priority inversion when using the CCM programming model in real-time applications.

5 Concluding Remarks

The CORBA object model and conventional ORBs have only recently begun specifying and implementing a standard [16] for composing and deploying "componentizable" services. Simultaneously, developers of high-performance, real-time, mission critical applications have begun employing distributed object computing middleware based on CORBA [26, 33, 34, 35, 36, 37, 38, 24]. Therefore, we believe it is essential to start leveraging experience in designing, optimizing, and tuning QoS-enabled ORB middleware to ensure standard CORBA Component Model (CCM) implementations will be sufficiently mature before they are considered for high-performance and real-time systems.

Reducing the memory footprint required to support the CCM is another area of concern for many real-time applications. Therefore, our research will implement a highly-flexible, just-in-time dynamic linking framework. This framework will reduce the memory footprint of TAO and CCM via dynamic configuration of middleware infrastructure, components, and services.

Based on our past experience on benchmarking, developing, optimizing, and deploying high-performance, real-time ORBs, we will identify the performance bottlenecks in conventional CCM implementations. We will apply optimization principle patterns so that developers in the high-performance and real-time communities can enjoy the advantages provided by the new CCM standard.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[2] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

[3] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[4] Sun Microsystems, "Jini Connection Technology." http://www.sun.com/jini/index.html, 1999.

[5] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[6] Anne Thomas, Patricia Seybold Group, "Enterprise JavaBeans Technology." http://java.sun.com/products/ejb/white_paper.html, Dec. 1998. Prepared for Sun Microsystems, Inc.

[7] The Object Management Group, "OMG's site for CORBA and UML Success Stories." http://www.corba.org/, 1999.

[8] D. C. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar, "A High-Performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

[9] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., August 1998.

[10] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[11] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

[12] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[13] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.

[14] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[15] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[16] BEA Systems, *et al.*, *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 ed., July 1999.

[17] S. Vinoski, "New Features for CORBA 3.0," *Communications of the ACM*, vol. 41, pp. 44–52, October 1998.

[18] C. Egremont, III, *Mr. Bunny's Guide to ActiveX*. Reading, MA: Addison-Wesley, 1999.

[19] Christopher D. Gill and Fred Kuhns and David L. Levine and Douglas C. Schmidt and Bryan S. Doerr and Richard E. Schantz and Alia K. Atlas, "Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems," in *Proceedings of the 1st IEEE International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, Nov. 1999.

[20] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[21] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[22] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, October 1999.

[23] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Principle Patterns to Real-time ORBs," *Concurrency Magazine*, to appear 2000.

[24] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[25] F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware," in *Proceedings of the IFIP* 6^{th} *International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, MA), IFIP, August 1999.

[26] F. Kon and R. H. Campbell, "Supporting Automatic Configuration of Component-Based Distributed Systems," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[27] 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*, (Atlanta, GA), ACM, October 1997.

[28] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.

[29] J. P. Morgenthal, "Microsoft COM+ Will Challenge Application Server Market." http://www.microsoft.com/com/wpaper/complus-appserv.asp, 1999.

[30] G. C. Hunt and M. L. Scott, "Intercepting and Instrumenting COM Application," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[31] J. Szymaszek, A. Uszok, and K. Zieliński, "Building a Scalable and Efficient Component Oriented System using CORBA – Active Badge System Case Study," in *Proceedings of the* 4^{th} *Conference on Object-Oriented Technologies and Systems*, (Santa Fe, NM), USENIX, April 1998. CCM related.

[32] D. C. Schmidt, D. L. Levine, and C. Cleeland, "Architectures and Patterns for Developing High-performance, Real-time ORB Endsystems," in *Advances in Computers*, Academic Press, 1999.

[33] G. Ambrosini, G. Mornacchi, C. Ottavi, and M. Romano, "Performance Evaluation of PowerPC VME Boards Running a Real-Time UNIX System." atddoc.cern.ch/, September 1997.

[34] E. Nett and M. Gergeleit, "Preserving Real-Time Behavior in Dynamic Distributed Systems," in *Proceedings of the IASTED Interational Conference on Intelligent Information Systems 1997*, IEEE, Dec. 1997.

[35] D. C. Schmidt, R. Bector, D. L. Levine, S. Mungee, and G. Parulkar, "An ORB Endsystem Architecture for Statically Scheduled Real-time Applications," in *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*, (San Francisco, CA), IEEE, December 1997.

[36] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the* 4^{th} *IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[37] D. L. Levine, C. D. Gill, and D. C. Schmidt, "Dynamic Scheduling Strategies for Avionics Mission Computing," in *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Nov. 1998.

[38] D. L. Levine, D. C. Schmidt, and S. Flores-Gaitan, "An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers," in *Proceedings of the Multimedia Computing and Networking 2000 (MMCN00) conference*, (San Jose, CA), ACM, Jan. 2000.

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

[40] C. O'Ryan, D. C. Schmidt, and D. Levine, "Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations," in *Proceedings of the* 5^{th} *Workshop on Object-oriented Real-time Dependable Systems*, (Montery, CA), IEEE, Nov. 1999.

[41] Object Management Group, *PIDL & Pseudo-Objects Policy Paper*, OMG Document ab/98-01-02 ed., January 1998.

A Overview of the CORBA Component Model Specification

This section presents a detailed overview of the CCM architecture.

Components: A component is a basic CORBA *meta-type*, *i.e.*, it can be referenced by multiple object references of different types. Each component has a set of *supported interfaces* that it inherits from other interfaces or components. A component encapsulates a design entity and is referenced by a *component reference*. For "component-unaware" clients, component references behave just like regular object references, *i.e.*, clients can invoke operations defined in supported interfaces. As shown in Figure 2, components interact with external entities, such as ORB services or other components, through the following *port* mechanisms:

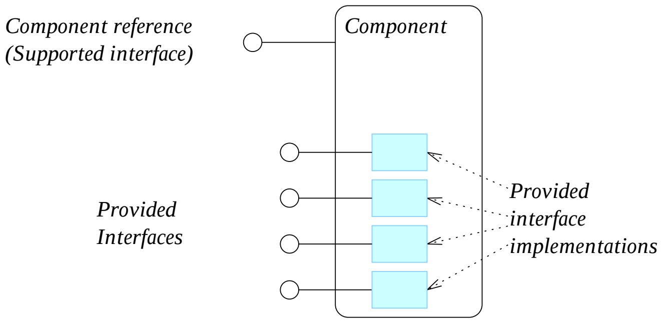

Figure 2: Architecture of a CCM Component

• **Facets:** A *facet*, also called a *provided interface*, is an interface contract exposed by a component. Facets are similar to COM's component model [3] in that they allow a component to support *unrelated interfaces*. Unrelated interfaces exposed through facets need not be related through inheritance to the component's supported interfaces.

The CCM's component model allows clients to *navigate* among provided interfaces and the equivalent interface defined by a component. In contrast, regular CORBA objects only allow clients to traverse interfaces in their inheritance trees. Clients that use components need not be component-aware. Only component-aware clients, however, can use the CCM navigation mechanism to traverse through the interfaces offered by a component.

• **Receptacles:** Components may *connect* to other objects and invoke methods upon those objects. *Receptacles* are used to specify the object connections among components and objects. Receptacles also provide a generic way to connect certain types of objects to a component. A receptacle can have single or multiple connections.

• **Event Sources/Sinks:** Components can express their interests to each other by supplying and consuming events through *event sources* and *event sinks*. The CCM event interfaces defines a subset of the CORBA Notification Service [39] as its event delivery mechanism, although the use of CORBA Notification Service is not required and component developers may choose to implement their own notification mechanism. The ability to connect various parts of a system using the notification mechanism is a common use-case in event-driven real-time systems [27, 40].

• **Attributes:** A component can use *attributes* to represent the states of an entire component. Component attributes differ from interfaces attributes, which describe the internal states of individual interfaces. Component attributes provide a standard mechanism for setting component states and are intended to be used by the CCM framework to configure component.

For instance, a `configure_complete` operation is defined in each component interface to transition a component from configuration phase to operation phase. Component implementors may deactivate access to certain attributes during configuration phase or operation phase. The specification does not limit the access of component attribute interfaces to the configuration phase, however, and clients can still gain access to the attribute interfaces. Therefore, if an implementor chooses to do so, attributes can be used by clients as regular interfaces to access state in a component.

The distinction between configuration phase and operating phase allows component developers to enforce data encapsulation without losing the flexibility of dynamic configuration.

• **Components Home:** A new keyword, `home`, is introduced to support *component homes*. A component home is the factory interface for a component. Each component home manages exactly one type of component. Home interfaces can optionally use a *key* to manage instances of the managed component. The key, if one exists, maps to an instance of the component. For a *keyless* home interface, invoking the factory method simply creates a new instance of the managed component type.

Component Implementation Framework (CIF): The CORBA *Component Implementation Framework* (CIF) defines the programming model for managing components' persistent states and constructing component implementations. The CCM specification defines a declarative language, *Component Implementation Definition Language* (CIDL), to describe implementations and persistent states for components and component homes. As shown in Figure 3, the CIF uses the CIDL descriptions to generate programming skeletons that automate basic behaviors of components, such as navigation, identity inquiries, activation, state management, lifecycle management, transactions, and security.

Containers: The CCM container programming model defines a set of APIs that simplify the task of developing and/or configuring CORBA applications. A container encapsulates a component implementation and uses these APIs to provide a run-time environment for the component that it manages. Figure 1 on page 2 shows the architecture of the container programming model.

Each container manages a component implementation defined by the CIF. A container creates its own POA for all the interfaces it manages. These interfaces can be decomposed as follows:

External APIs: These are the interfaces defined by the component including the *supported interfaces*, *provided interfaces*, and the component *home* interface. External APIs are available to clients.

Container APIs: These include the *internal interfaces* that the component can invoke to access to the services provided by the container and the *callback interfaces* that the container can invoke on the component.

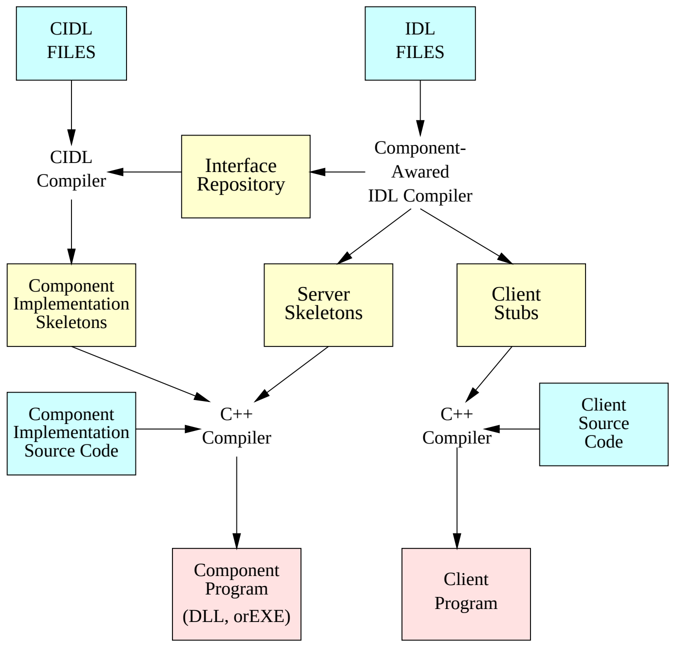

Figure 3: Using IDL and CIDL for component implementation

Through the collaboration of these interfaces, a container provides its managed component access to its POA and the services supported by the ORB.

There are two types of container interfaces: (1) *session* container interfaces for transient components and (2) *entity* container interfaces for persistent components. A *CORBA Usage Model* specifies the required interaction pattern between the container and the POA and the CORBA Services by specifying the interfaces' transientness/persistency and cardinality of servant to OID mapping.

The *component category* defines the legal combinations of the container API types and the CORBA usage models. By specifying a container's component category along with other policies, component developers can specify a wide range of configuration options in the CIF. The CIF then generates the component implementation with proper strategies for persistence, event notification, transaction, security, etc.

When combined with OMG's Real-time CORBA [10] and Messaging [11] specifications, the container programming model provides application developers with a model for creating, specifying, and partitioning various run-time characteristics, such as end-to-end priority and connection bandwidth utilization, for components in real-time systems.

Packaging and Deployment: The CCM defines standard techniques and patterns for packaging and deploying components. The CCM uses the *Open Software Description* (OSD), which is an XML Document Type Definition (DTD) defined by W3C to describe software packages and their dependencies. The deployment mechanism allows remote installation and activation of new or modified components. The OSD feature is useful for certain real-time applications that require dynamic configuration or off-site software maintenance, such as upgrading software packages on-board space vehicles in-flight.

ORB extension $\rightarrow$ locality constrainted interfaces: Locality constrained interfaces have historically been limited to ORB-defined types, such as `CORBA::NVList`, `CORBA::Request`, and `CORBA::TypeCode`, and were often defined using so-called pseudo-IDL (PIDL) [41]. To support the component model efficiently, and to eliminate the need for PIDL, the CCM specifies a new IDL keyword, called `local`, which standardizes the definition of *locality constrained* interfaces. As its name implies, a local interface is (1) only valid in the process in which it is instantiated and (2) cannot be externalized to or invoked from other processes. Adding standard support for locality constrained interfaces to CORBA is particularly important for server-only components because it helps improve performance and minimize memory footprint.