Report Number: WUCSE-2004-23

2004-05-03

Composing Systemic Aspects into Component-Oriented DOC Middleware

Nanbor Wang

The advent and maturation of component-based middleware frameworks have sim-plified the development of large-scale distributed applications by separating system devel-opment and configuration concerns into different aspects that can be specified and com-posed at various stages of the application development lifecycle. Conventional component middleware technologies, such as J2EE [73] and .NET [34], were designed to meet the quality of service (QoS) requirements of enterprise applications, which focus largely on scalability and reliability. Therefore, conventional component middleware specifications and implementations are not well suited for distributed real-time and embedded (DRE) ap-plications with more stringent QoS requirements, such as low latency/jitter, timeliness,... **Read complete abstract on page 2.**

Composing Systemic Aspects into Component-Oriented DOC Middleware

Nanbor Wang

Complete Abstract:

The advent and maturation of component-based middleware frameworks have sim-plified the development of large-scale distributed applications by separating system devel-opment and configuration concerns into different aspects that can be specified and com-posed at various stages of the application development lifecycle. Conventional component middleware technologies, such as J2EE [73] and .NET [34], were designed to meet the quality of service (QoS) requirements of enterprise applications, which focus largely on scalability and reliability. Therefore, conventional component middleware specifications and implementations are not well suited for distributed real-time and embedded (DRE) ap-plications with more stringent QoS requirements, such as low latency/jitter, timeliness, and online fault recovery. In the DRE system development community, a new generation of enhanced commercial off-the-shelf (COTS) middleware, such as Real-time CORBA 1.0 (RT-CORBA)[39], is increasingly gaining acceptance as (1) the cost and time required to develop and verify DRE applications precludes developers from implementing complex DRE applications from scratch and (2) implementations of standard COTS middleware specifications mature and encompass key QoS properties needed by DRE systems. However, although COTS middleware standardizes mechanisms to configure and control underlying OS support for an application's QoS requirements, it does not yet provide sufficient abstractions to separate QoS policy configurations such as real-time performance requirements, from application functionality. Developers are therefore forced to configure QoS policies in an ad hoc way, and the code to configure these policies is often scattered throughout and tangled with other parts of a DRE system. As a result, it is hard for developers to configure, validate, modify, and evolve complex DRE systems consistently. It is therefore necessary to create a new generation of QoS-enabled component middleware that provides more comprehensive support for addressing QoS-related concerns modularly, so that they can be introduced and configured as separate systemic aspects. By analyzing and identifying the limitations of applying conventional middleware technologies for DRE applications, this dissertation presents a new design and its associated techniques for enhancing conventional component-oriented middleware to provide programmability of DRE relevant real-time QoS concerns. This design is realized in an implementation of the standard CORBA Component Model (CCM) [38], called the Component-Integrated ACE ORB (CIAO). This dissertation also presents both architectural analysis and empirical results that demonstrate the effectiveness of this approach. This dissertation provides three contributions to the state of the art in composing systemic behaviors into component middleware frameworks. First, it illustrates how component middleware can simplify development and evolution of DRE applications while ensuring stringent QoS requirements by composing systemic QoS aspects. Second, it contributes to the design and implementation of QoS-enabled CCM by analyzing and documenting how systemic behaviors can be composed into component middleware. Finally, it presents empirical and analytical results to demonstrate the effectiveness and the advantage of composing systemic behaviors in component middleware. The work in this dissertation has a broader impact beyond the CCM in which it was developed, as it can be applied to other component-base middleware technologies which wish to support DRE applications.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2004-23

Composing Systemic Aspects into Component-Oriented DOC Middleware, Doctoral Dissertation, May 2004

Authors: Nanbor Wang

May 3, 2004

Abstract: The advent and maturation of component-based middleware frameworks have simplified the development of large-scale distributed applications by separating system development and configuration concerns into different aspects that can be specified and composed at various stages of the application development lifecycle. Conventional component middleware technologies, such as J2EE and .NET, were designed to meet the
quality of service (QoS) requirements of enterprise applications, which focus largely on scalability and reliability. Therefore, conventional component middleware specifications and implementations are not well suited for distributed real-time and embedded (DRE) applications
with more stringent QoS requirements, such as low latency/jitter, timeliness, and online fault recovery.

In the DRE system development community, a new generation of enhanced commercial off-the-shelf (COTS) middleware, such as Real-time CORBA 1.0
(RT-CORBA), is increasingly gaining acceptance as (1) the cost and time required to develop and verify DRE applications precludes developers from implementing complex DRE applications from scratch and (2) implementations of standard COTS middleware specifications mature and encompass key QoS properties needed by DRE systems. However, although COTS middleware standardizes mechanisms to configure and

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COMPOSING SYSTEMIC ASPECTS INTO COMPONENT-ORIENTED DOC MIDDLEWARE

by

Nanbor Wang

Prepared under the direction of Dr. Christopher D. Gill and Dr. Douglas C. Schmidt

---

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

May, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

COMPOSING SYSTEMIC ASPECTS INTO COMPONENT-ORIENTED DOC MIDDLEWARE

by Nanbor Wang

ADVISORS: Dr. Christopher D. Gill and Dr. Douglas C. Schmidt

May, 2004
Saint Louis, Missouri

The advent and maturation of component-based middleware frameworks have simplified the development of large-scale distributed applications by separating system development and configuration concerns into different aspects that can be specified and composed at various stages of the application development lifecycle. Conventional component middleware technologies, such as J2EE [73] and .NET [34], were designed to meet the quality of service (QoS) requirements of enterprise applications, which focus largely on scalability and reliability. Therefore, conventional component middleware specifications and implementations are not well suited for distributed real-time and embedded (DRE) applications with more stringent QoS requirements, such as low latency/jitter, timeliness, and online fault recovery.

In the DRE system development community, a new generation of enhanced commercial off-the-shelf (COTS) middleware, such as Real-time CORBA 1.0

(RT-CORBA)[39], is increasingly gaining acceptance as (1) the cost and time required to develop and verify DRE applications precludes developers from implementing complex DRE applications from scratch and (2) implementations of standard COTS middleware specifications mature and encompass key QoS properties needed by DRE systems. However, although COTS middleware standardizes mechanisms to configure and control underlying OS support for an application's QoS requirements, it does not yet provide sufficient abstractions to separate QoS policy configurations such as real-time performance requirements, from application functionality. Developers are therefore forced to configure QoS policies in an *ad hoc* way, and the code to configure these policies is often scattered throughout and tangled with other parts of a DRE system. As a result, it is hard for developers to configure, validate, modify, and evolve complex DRE systems consistently.

It is therefore necessary to create a new generation of QoS-enabled component middleware that provides more comprehensive support for addressing QoS-related concerns modularly, so that they can be introduced and configured as separate systemic aspects. By analyzing and identifying the limitations of applying conventional middleware technologies for DRE applications, this dissertation presents a new design and its associated techniques for enhancing conventional component-oriented middleware to provide programmability of DRE relevant real-time QoS concerns. This design is realized in an implementation of the standard CORBA Component Model (CCM) [38], called the Component-Integrated ACE ORB (CIAO). This dissertation also presents both architectural analysis and empirical results that demonstrate the effectiveness of this approach.

This dissertation provides three contributions to the state of the art in composing systemic behaviors into component middleware frameworks. First, it illustrates how component middleware can simplify development and evolution of DRE applications while ensuring stringent QoS requirements by composing systemic QoS aspects. Second, it contributes to the design and implementation of QoS-enabled CCM by analyzing and documenting how systemic behaviors can be composed into component middleware. Finally, it presents empirical and analytical results to demonstrate the effectiveness and the advantage of composing systemic behaviors in component middleware. The work in this dissertation

has a broader impact beyond the CCM in which it was developed, as it can be applied to other component-base middleware technologies which wish to support DRE applications.

To Pam,

my dear wife and my best friend,

for everything we share together,

in good times and in bad times.

Contents

List of Tables

List of Figures

Acknowledgments

The journey has turned out to be not only an intellectual adventure but also a period of great personal growth and an opportunity to rediscover oneself. Every soul that has crossed my path taught me something in every aspect of life for which I am grateful for. However, this dissertation could not have been completed without the help, support, encouragement and friendship of many people.

First, I would like to express my gratitude to my mentors and advisors. Dr. Douglas C. Schmidt has made this all possible for bringing me to the DOC Group and for his continued support throughout my stay in the DOC Group. He has been heavily participating in the work and research leading to this dissertation, even when he ventured out to Irvine, D.C., and finally, Nashville while working on host of other things. Dr. Ron K. Cytron has been very supportive and offered many challenging questions which I appreciate a lot. Dr. Christopher D. Gill has been of tremendous assistance by lending a helping hand when I needed the most, in my final stage of completing the dissertation. His encouragements and numerous discussions have guided this dissertation greatly. He also chaired my final defense committee.

I am grateful to my dissertation defense committee, Dr. Christopher D. Gill, Dr. Douglas C. Schmidt, Dr. Ron K. Cytron, Mr. David C. Sharp, Dr. Roger D. Chamberlain, and Dr. Ronald S. Indeck, for their time and efforts spent reviewing, commenting on, and making suggestions for improvement to this dissertation. I also want to express my gratitude to Dr. John R. Cary and Dr. Svetlana G. Shasharina of Tech-X Corporation for their confidence in me and their support for me to finish up this dissertation. I am similarly indebted to my many mentors who have helped guiding my research and my growth both in knowledge and character, Dr. Richard Schantz, Dr. Joseph P. Loyall, Dr. Joseph K. Cross, Mr. David C. Sharp, Dr. David L. Levine, Dr. Ebrahim Moshiri, Craig Rodrigues, Gautam H. Thaker, Dennis D. Noll, and Wendy Roll.

Support for this research was provided by NASA under cooperative agreement NCC3-777. Support for this research was also provided by the DARPA PCES program under contracts F33615-00-C-3048 and F33615-03-C-4111.

I could not have done the work without the help of many people in various aspects of the development of CIAO, specifically, Jeff Parsons for lots of help in IDL and CIDL development, Boris Kolpakov for his work on the CIDL compiler, Andrey Nechypurenko for contributing a GUI component which is indispensable for demonstrating CIAO, Craig Rodrigues for his many discussions, critiques and suggestions, Dr. Irfan Pyarali for his help on RT-CORBA, Arvind S. Krishna for developing the benchmarking tests, and also Carol L. Sanders for her help in testing the pre-alpha release and her many suggestions.

I am indebted to my friends for their support and encouragement. They are Darrell E. Brunsch, Angelo Corsaro, Jeff Parsons, Balachandran Natarajan, Yamuna Krishnamurthy, and, Dr. Irfan Pyarali. I am grateful for past and present DOC Group members, for their the friendship and collaboration which make my work and study in the DOC Group a joy, including Venkita Subramonian, Huang-Ming Huang, Dr. Aniruddha Gokhale, Michael Kircher, Kirthika Parameswaran, Dr. Carlos O'Ryan, Pradeep Gore, Sharath Cholleti, Ossama Othman, Tao Lu, Marina Spivak, Jaiganesh Balasubramanian, Mayur Deshpande, Vishal Kachroo, Chris Cleeland, Shawn Hannan, Luther Baker, Joe Hoffert, and Krishnakumar Balasubramanian.

I can not express my gratitude enough to my parents, T.H. and H.J. Wang, for their unconditional love, support and encouragement. Similarly, to my big sister and her husband, Dr. James and Kay Han, and our friends, Heng-Hsin and Hwei-Wen Liao, for their love and support. Last, but not least, to my beloved wife, Pam, if not for your endless patience, love, and support, I could not have finished this challenge.

Nanbor Wang

Washington University in Saint Louis
May 2004

Chapter 1

Introduction

1.1 Motivation

Commercial-off-the-shelf (COTS) middleware technologies, such as The Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) [40], Sun's Java RMI [86], and Microsoft's COM+ [35], have matured considerably in recent years. They are being used to reduce the time and effort required to develop applications in a broad range of information technology (IT) domains. Historically, these middleware technologies have been applied to *enterprise applications* [20] such as online catalog and reservation systems, bank asset management systems, and management planning systems. Recently, however, more and more applications from *different* domains are beginning to take advantage of COTS middleware.

All applications depend at least implicitly on the behavior of underlying platforms, including software and hardware, on which they run. COTS middleware helps simplify software development by abstracting and standardizing common runtime behaviors within middleware *frameworks*. As is the case for operating systems, in order to apply a middleware framework to as many application domains as possible middleware standards try to standardize only the behaviors that are common denominators of all applicable domains. This abstraction of common behaviors simplifies application development and allow developers to design software modules and their interactions easily via well-defined *interfaces*.

Regardless of the domain in which middleware is applied, however, its goal is to help expedite the overall software process by (1) making it easier to integrate parts together and (2) shielding developers from many complexities, such as platform and language heterogeneity, resource management, and access control. Applications from different domains, however, often also have domain-specific behavioral requirements, *e.g.*, most

web commerce applications require security features such as user certificates and message encryption. To extend the applicability of COTS middleware, it is necessary to address domain-specific requirements when they are needed. Although these behavioral requirements are vital for the correctness of applications, they are often hard to capture in an application's functional logic or interface definitions, as they often crosscut multiple software layers and thus require some degree of control over all software modules involved.

This dissertation will use the term "systemic aspects" to denote these requirements. Because of the difficulty in offering an abstraction for the management of systemic aspects, conventional COTS middleware has only gone as far as to standardize the interfaces for manipulating these behaviors, and leaves the logic for managing these aspects to the application developers. Distributed Real-time and Embedded (DRE) applications, for example, have distinctly different requirements on systemic aspects than conventional desktop or back office applications in that *the right answer delivered too late can become the wrong answer*, *i.e.*, failure to meet key Quality of Service (QoS) requirements can lead to catastrophic consequences. In order for the benefits of COTS middleware to be realized fully by DRE applications, domain-specific extensions must be added to middleware standards to give developers appropriate control over middleware frameworks, to satisfy stringent QoS requirements such as predictability, latency, efficiency, scalability, dependability, and security. Because DRE related interfaces reside in the same abstraction layer as the application interfaces, they tend to entangle with the application logic everywhere.

Component middleware [75] is a class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly. In particular, component middleware offers application developers the following reusable capabilities:

- *Connector mechanisms between components*, such as remote method invocations and message passing
- *Horizontal infrastructure services*, such as request brokers, and
- *Vertical models of domain concepts*, such as common semantics for higher-level reusable component services ranging from transaction support to multi-level security.

These abstractions allow application developers to separate systemic aspects from the application components, into first-class entities defined in the component middleware standards. Examples of COTS component middleware include the CORBA Component Model

(CCM) [38], Java 2 Enterprise Edition (J2EE) [73], and the Component Object Model (COM) [5], each of which uses different interfaces, protocols, and component models.

DRE applications, such as industrial controllers for surface-mount hardware pick-and-place machines [76] or total ship computing environments [63], have stringent individual QoS requirements, *each* of which must be satisfied simultaneously. Examples of such QoS systemic requirements include bounded response latency and jitter which require allocating various resources, such as processing resource or network bandwidth, to ensure these QoS requirements can be met. Failure to meet these QoS requirements, *e.g.*, a missed deadline, often leads to serious consequences even if an application may still logically function properly. Conventional component middleware technologies, however, are designed largely for applications with business-oriented QoS requirements, such as data persistence, confidentiality, and transactional support.

The supporting run-time environments in conventional component middleware, therefore, do not support the kind of QoS required by DRE applications. DRE developers will need to embed code to allocate and manage DRE-related QoS and resources into component implementations if they wish to take advantage of conventional component middleware technologies. Programming QoS management and allocation explicitly in conventional component implementations is not sufficient as

1. embedding QoS management code hampers the reusability of component implementations,

2. QoS requirements often require end-to-end enforcement and collaboration of all interaction component implementations where they have no prior knowledge about other components' specific QoS requirements, and

3. QoS resources often are shared by multiple component implementations with compatible QoS requirements and therefore are not appropriate to be allocated in a component implementation.

Moreover, implementing the QoS provisioning logic into component implementations reintroduces problems of tightly coupled and hard to reuse implementations that component middleware tried to solve.

It is not suitable to apply conventional component middleware directly to other application domains such as DRE applications. Rather, component middleware needs to be

extended to make DRE-related QoS requirements an integral part of the middleware framework. Necessary extensions include supporting both mechanisms for managing the resources for added QoS requirements and the application composition descriptions to specify the required QoS requirements and the resources for supporting them. The goal of this dissertation is to extend the applicability of component middleware, using the CORBA Component Model as the basis framework and DRE application domain as the target domain, by composing systemic aspects into DRE applications.

1.2 Design and Implementation Challenges

To ensure that DRE applications can achieve their QoS requirements, various kinds of *QoS provisioning* must be performed to allocate and manage computing and communication resources throughout a distributed system. QoS provisioning can be performed in two main ways:

- *Statically*, where adequate resources required to support a particular degree of QoS are pre-configured into an application. Examples of static QoS provisioning include task and communication bandwidth reservations.
- *Dynamically*, where the resources required are determined and adjusted based on the runtime system status. Examples of dynamic QoS provisioning include dynamic adjustment of video quality to adapt to available network bandwidth and reallocation of CPU cycles to ensure completion of critical tasks at run time.

QoS provisioning in DRE systems crosscuts multiple layers and requires end-to-end enforcement. Conventional component middleware technologies, such as CCM, J2EE, and COM, were designed largely for applications with business-oriented QoS requirements, such as data persistence, confidentiality, and transactional support. Thus, they do not effectively address simultaneously enforcing multiple stringent end-to-end QoS requirements of DRE applications. What is therefore needed is *QoS-enabled component middleware* that preserves existing support for heterogeneity in standard component middleware, yet also provides multiple dimensions of QoS provisioning and enforcement [55] to meet the stringent end-to-end QoS requirements of DRE applications.

Among the existing component middleware technologies, CCM is the most suitable for DRE applications since the current base-level CORBA specification is the only standard COTS middleware that has made substantial progress in satisfying the QoS requirements of DRE systems. For example, the OMG has adopted several DRE-related

specifications, including **Minimum CORBA**, **Real-time CORBA**, **CORBA Messaging**, and **Fault-tolerant CORBA**. These QoS specifications and their accompanying enforcement capabilities are essential for supporting DRE systems. This dissertation therefore selects CCM as the basis for developing QoS-enabled component models that are able to support DRE systems.

A common strategy to allocate resources for satisfying QoS requirements in DRE applications is to specify them *statically* because the allocations are often very effective for most DRE application and are easy to apply and verify. Furthermore, composing static QoS provisioning does not require changing a component implementation to interact with the composed QoS provisioning mechanisms and thus allows greater flexibility to compose applications having different QoS requirements. In contrast, other research looking to provide new QoS support in component middleware tends to require modifications to allow component implementations to interact with QoS support mechanisms, which makes applications less robust. This dissertation, therefore, focuses specifically on addressing *static* QoS provisioning challenges in DRE application development. In order to successfully support developing, composing and deploying DRE applications with properly provisioned resources for enforcing QoS requirements, this research addresses key technical challenges outlined in Table 1.2.

Moreover, the stringent QoS requirements of DRE applications make them sensitive to the performance of the underlying middleware. A key research challenge is to design component middleware technologies that can deliver performance comparable to that of conventional COTS middleware such as Real-Time CORBA.

1.3 Research Contributions

This dissertation makes three major contributions to the state of the art in *composing* systemic behaviors within component middleware frameworks.

The need for QoS-aware component middleware: First, using DRE applications as its target application domain, it illustrates how component middleware can simplify the development of DRE applications, while still meeting stringent cross-cutting performance requirements by composing QoS provisioning policies statically with application components. It reviews the limitations of conventional component middleware in supporting DRE applications. It then shows how these limitations can be addressed by composing systemic

Table 1.1: **Challenges and Solution Techniques Presented in This Research**

Challenges	Solution Techniques	Effects and Results
Multiple scopes for applying systemic policies in component middleware at run-time.	Identify proper binding granularity for applying various QoS policies in various scopes in component middleware as described in Section 4.4.1.	The test programs used to validate real-time support in CIAO demonstrates the effectiveness of the granularity CIAO supports.
Multiple stages for inserting systemic policies in component development lifecycle.	Identify and document proper binding stages in various stages of application development lifecycle for key QoS provisioning policies critical to DRE applications as described in Section 4.4.2.	The real-time validation tests show how CIAO supports various real-time policies can be composed into application at various development stage to control its real-time behaviors flexibly and effectively. Analytical results provide suggestions on how the added information can help improving application performance over the development lifecycle.
Consistent systemic policies must be applied throughout an application end-to-end.	The work in this dissertation presents extensions to application deployment configuration to specify and apply QoS policies consistently throughout an application, regardless of the location of a component instance. The design is described in detail in Section 5.5.	The design of real-time tests demonstrates how real-time policies can be composed into applications flexibly, consistently, and robustly, comparing to traditional DOC middleware.
Resolving platform diversity in supporting mechanisms.	This research develops and documents solutions for managing differences in QoS management mechanisms when composing general QoS policies with platform-specific QoS management mechanisms. Section 5.4.2 details the design CIAO employs.	The performance tests conducted by this research depend on the design to deploy the test applications to different deployment target platform configurations flexibly.

QoS policies into applications. An example implementation called CIAO validates the use of QoS-aware component middleware.

Addressing design and implementation challenges in composing systemic behaviors in component middleware: This dissertation also makes the following contributions to the state of the art in the *design and implementation* of component middleware frameworks to support composable systemic aspects:

1. It analyses and documents the proper binding points in various stages of the component development lifecycle for several key QoS provisioning policies critical to DRE applications.

2. It reviews and documents different granularities for binding QoS provisioning policies, and examines the impact of binding with different granularities.

3. It presents extensions to manage application deployment configuration to minimize inconsistency and to make analyzing deployment information easier.

4. It develops and documents solutions for managing differences in QoS management mechanisms when composing general QoS policies with platform-specific QoS management mechanisms.

Validating the effectiveness and demonstrating the benefits of composing systemic behaviors: Finally, this dissertation presents empirical and analytical comparisons between component middleware and conventional DOC middleware, and shows the effectiveness of composing systemic aspects into DRE applications that is achieved by the approach presented in this dissertation. It first presents performance comparisons showing the small and bounded amount of overhead for adopting component middleware and QoS-aware component middleware. These performance comparison figures provide information helping developers in deciding the cost to adopt component middleware. It then validates the effectiveness of extended features for supporting composition of systemic behaviors. It also demonstrates how a QoS-aware component middleware framework like CIAO makes the development and evolution process for DRE applications much easier.

1.4 Dissertation Organization

This dissertation is organized as follows. Chapter 2 surveys related work in the area of composing systemic behavior into middleware. Relevant topics include applications of meta-programming techniques, QoS control and adaptation techniques, aspect-oriented programming and other middleware technologies. Chapter 3 presents an overview of component middleware, including an introduction to the CORBA Component Model (CCM), and an explanation of how CCM features can help to ease application development. Chapter 4 points out key limitations of conventional middleware such as CCM, and how a new component middleware framework called the Component-Integrated ACE ORB (CIAO) helps address these limitations. Chapter 5 describes key implementation details of the CIAO component middleware framework, including its extensions to the CCM specification. Chapter 6 presents empirical results, comparing the performance of two versions of an experimental application built using TAO and CIAO respectively, and demonstrates how CIAO helps in the development and verification of real-time applications. Finally, Chapter 7 presents concluding remarks about these findings, discusses impacts of the research on composing systemic behaviors in component middleware, and outlines future research directions.

Chapter 2

Survey of Related Work

Abstraction has always been a focal point in the evolution of computer science. New programming languages and paradigms come to life to help programmers solve problems and manage complexity by providing higher levels of abstraction and better separation of concerns.

This chapter first discusses previous efforts focused on raising levels of abstraction and improving separation of concerns, such as meta-programming and aspect-oriented programming techniques. It then surveys efforts that integrate QoS support into programming environments, in particular for representative DOC middleware. Finally, it discusses other prior and on-going research efforts in integrating QoS support with component-based middleware.

This section reviews work on QoS provisioning mechanisms using the taxonomy shown in Figure 2.1. One dimension depicted in Figure 2.1 is *when QoS provisioning is performed, i.e.,* static versus dynamic QoS provisioning, as described in Section 1.2. Some enabling mechanisms allow static QoS provisioning before the startup of a system, whereas and others provide abstractions to define dynamic QoS provisioning behaviors during runtime based on resources available at the time. The other dimension depicted in Figure 2.1 is the *level of abstraction*. Both middleware-based approaches shown in the figure, *i.e.,* CIAO and BBN's QuO Qoskets, offer higher levels of abstraction for QoS provisioning specification and modeling. Conversely, the programming language-based approach offers meta-programming mechanisms for injecting QoS provisioning behaviors. We review previous research in the area of QoS provisioning mechanisms along these two dimensions.

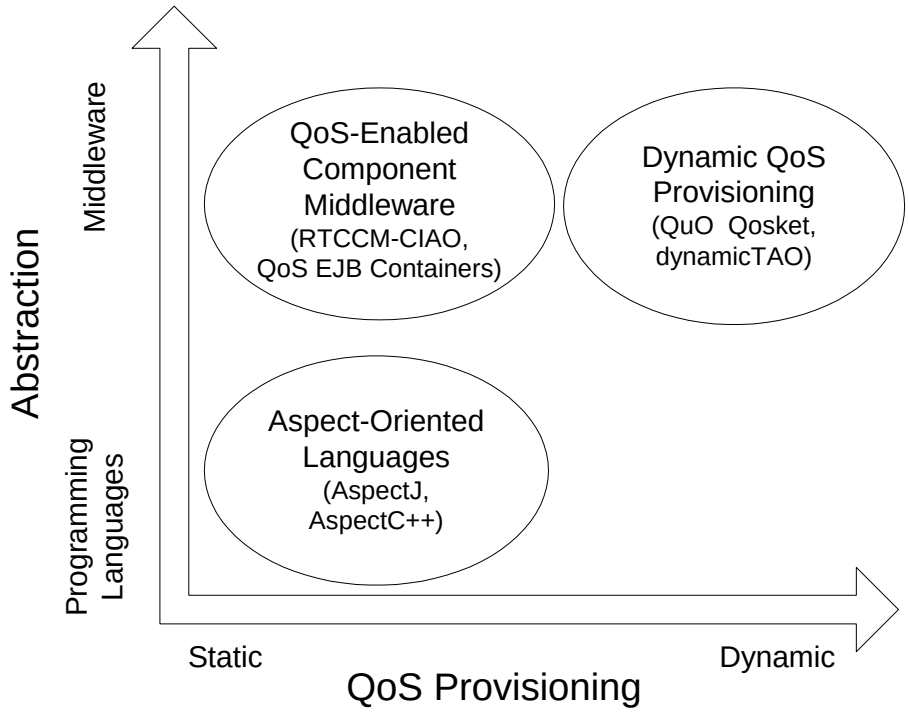

Figure 2.1: **Taxonomy of QoS Provisioning Enabling Mechanisms**

2.1 Aspect-oriented Programming

Aspects are programming concerns that are not directly related to the main purpose of a program but are nonetheless important factors for the program's functioning. For example, how a communication framework allocates and manages the buffer memory among different layers can significantly affect the behaviors of the framework. Aspects, however, are hard to decompose from a software system and to be encapsulated in software module using traditional object-oriented or procedure-oriented programming models because they often cross-cut multiple layers of software modules and require management in unison. Meta-programming provides techniques to improve software adaptability by abstracting the management of certain aspects from a software module through an interface implementing the abstraction called a "meta-protocol." Aspect-oriented programming, which evolved from meta-programming, provides not only techniques to separate concerns from software modules but also mechanisms to weave application logic with aspects.

Aspect-Oriented programming (AOP) [27] provides language-level abstractions to weave different aspects that cross-cut multiple layers of a system. Examples of AOP tools include AspectJ [26] and AspectC++ [47]. Similar to AOP tools, CIAO supports injection of aspects into systems at the middleware level throughout an application. Both CIAO

and AOP tools weave aspects statically, and neither defines abstractions for dynamic QoS provisioning. In contrast, more dynamic QoS provisioning technologies covered later in this Chapter use AOP techniques to organize and connect the various dimensions of the QoS management abstractions with each other and with the program to which the behavior is being attached. Compared to CIAO, AOP supports aspect weaving only within a single program whereas CIAO aims to provide composition of systemic behaviors *across* multiple potentially heterogeneous programs in a distributed application.

2.2 QoS-enabled Middleware

The *Quality Objects* (QuO) framework [59, 87] is an adaptive middleware framework developed by BBN Technologies that allows DRE system developers to use aspect-oriented software development [27] techniques to separate the concerns of QoS programming from application logic in DRE applications. A *Qosket* is a unit of encapsulation and reuse for QuO systemic behaviors. In comparison to CIAO, Qoskets and QuO emphasize dynamic QoS provisioning where CIAO emphasizes static QoS provisioning and integration of various mechanisms and behaviors during different stages of the development lifecycle. We are collaborating with BBN to integrate Qoskets and CIAO [80] to provide a total QoS provisioning solution.

In their *dynamicTAO* project, Kon and Campbell [29] apply reflective middleware techniques to extend TAO to reconfigure the ORB at runtime by dynamically linking selected modules, according to the features required by the applications. Their work is similar to *Qoskets* (shown in Figure 2.1), in that both provide the mechanisms for realizing *dynamic* QoS provisioning at the middleware level. Qoskets offer a more comprehensive QoS provisioning abstraction, however, whereas Kon and Campbell's work concentrates on configuring *middleware* capabilities.

Moreover, although Kon and Campbell's work can also provide QoS adaptation behavior by dynamically (re)configuring the middleware framework, their research may not be as suitable for DRE applications, since dynamic loading and unloading of ORB components can incur significant and unpredictable overheads and thus prevent the ORB from meeting application deadlines. Our work on CIAO relies upon Model Driven Architecture (MDA) tools [21] to analyze the required ORB components and their configurations. This approach ensures the ORB in a component server contains only the required components, without compromising end-to-end predictability.

2.3 Other QoS-enabled Component Middleware

Middleware can apply the Quality Connector pattern [9] to meta-programming techniques for specifying the QoS behaviors and configuring the supporting mechanisms for these QoS behaviors. The container architecture in component-based middleware frameworks provides a vehicle for applying meta-programming techniques for QoS assurance control in component middleware, as was previously identified in [82]. Containers can also help apply aspect-oriented software development [27] techniques to plug in different systemic behaviors [8]. These projects are similar to CIAO in that they provide mechanisms to inject "aspects" into applications statically at the middleware level.

Miguel de Miguel further develops the state of the art in QoS-enabled containers by extending a QoS EJB container interface to support a `QoSContext` interface that allows the exchange of QoS-related information among component instances [12]. To take advantage of the QoS-container, a component must implement `QoSBean` and `QoSNegotiation` interfaces. This requirement, however, adds an unnecessary dependency to component implementations. Section 4.1 examines the limitations of implementing QoS behavior logic in component implementations.

The QoS Enabled Distributed Objects (Qedo) project [15] is another ongoing effort to make QoS support an integral part of CCM. Qedo targets applications in the telecommunication domain and supports information streaming. It defines a metamodel with multiple categories of QoS requirements for applications. To support the modeled QoS requirements, Qedo defines extensions to CCM's container interface and the Component Implementation Framework (CIF) to realize the QoS models [56].

Similar to QuO's Contract Definition Language (CDL), Qedo's contract metamodel provides mechanisms to formalize and abstract QoS requirements. QuO (Qosket) contracts are more versatile, however, because they not only provide high levels of abstraction for QoS status, but also define actions that should take place when state transitions occur in contracts. Qedo's extensions to container interfaces and CIF also require component implementations to interact with the container QoS interface and negotiate the level of QoS contract directly. While this approach is suitable for certain applications where QoS is part of the functional requirements, it inevitably tightly couples the QoS provisioning and adaptation behaviors into the component implementation, and thus hampers the reusability of component.

In comparison, our approach explicitly avoids this coupling and tries to compose the QoS provision behaviors into the component systems.

Compared to all these other QoS-enabled component middleware technologies, CIAO aims to *compose* systemic aspects into component-based applications while these other technologies each depends on a specialized QoS-enabled container interface. While certain applications do require direct interaction with QoS assurance mechanisms to adapt to system conditions, this approach forces QoS-aware component implementations to be tightly coupled to the type of container and therefore reduces their reusability. In contrast, CIAO will support even the composition of adaptive behaviors for a component so that component implementations need not be tied to a specific container implementation unless their implementation demands it.

2.4 Standardization Efforts

The OMG has started several standardization efforts that match the goals we are trying to achieve in CIAO. The Light Weight CCM [43] specification aligns with CIAO's goal of reducing the footprint of CCM implementations to make CCM more suitable for DRE applications. This is achieved by removing features, such as persistent state management and transaction support, which are not commonly used in DRE applications, from the existing CCM specification. CIAO's research project, which predates the Lightweight CCM specification effort, also aims to provide a CCM subset for DRE applications. We have proposed and implemented a similar subset of features to those in the Light Weight CCM submission. Future versions of CIAO will be amended to conform with the Light Weight CCM specification once that specification is formally adopted.

The Deployment and Configuration [42] (D&C) submission aims to provide a framework for deploying and configuring complex component systems. Similar to our approach, it also raises the question of available resources and uses the deployment and configuration stages of a component development lifecycle to ensure the needs of the applications are met. It is currently still in the standardization process and we will be able to apply the specification once it is formally adopted. Many features of the D&C specification designed to decouple the configuration of deployment platforms from application assembly have been in CIAO since before the D&C specification efforts began. The CIAO team will integrate the work in this dissertation and the implementation of the new D&C specification in CIAO in the near future.

The UML Profiles for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms [46] also relate to CIAO's QoS-enabled CCM efforts. That specification is not, however, specifically aimed at component-based applications. Two other efforts

that are also related to CIAO are the Quality of Service for CCM Request for Proposal (RFP) [44] and the Streams for CCM RFP [45].

Chapter 3

Overview of the CORBA Component Model

This chapter presents an overview of component middleware, using the CORBA Component Model (CCM) as its example. It first motivates the need for component models in Section 3.1 by reviewing the benefits and the limitations of conventional distributed object computing (DOC) middleware upon which component middleware builds. Section 3.2 then introduces CCM and reviews the major building blocks in the CCM architecture.

3.1 Motivation for Component Models

In the early days of computing, each piece of software was developed from scratch to achieve a particular goal on a specific hardware platform. Since computers were themselves much more expensive than the cost to program them, scant attention was paid to systematic software reuse and composition of applications from existing software artifacts. Over the past four decades, however, the following two trends have spurred the transition from hardware-centric to software-centric development paradigms:

- **Economic factors** – Due to advances in VLSI and the commoditization of hardware, most computers are now *much* less expensive than the cost to program them. Significant economic gains can now be realized by improving software development processes.
- **Technological advances** – With the advent of software development technologies such as object-oriented programming languages and distributed object computing

frameworks, it has become easier to develop complex software systems. Consequently, more capabilities and features are then added to software which makes developing and maintaining complex software systems a complicated task.

A common theme underlying the evolution of modern software development paradigms is the desire for reuse, *i.e.*, to compose and customize applications from pre-existing software building blocks [13]. Major modern software development paradigms all aim to achieve this common goal but differ in the kind(s) and granularity of building blocks that form the core of each paradigm. The development and evolution of middleware technologies also follow a similar goal, to capture and reuse *design* information learned previously, within various layers of software.

Middleware is a kind of reusable software that resides between applications and the underlying operating systems, network protocol stacks, and hardware [60]. Middleware's primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure, and to coordinate how parts of applications are connected and how they interoperate. Middleware focuses especially on issues that emerge when such programs are used across physically separated platforms. When developed and deployed properly, middleware frameworks can reduce the cost and risk of developing distributed applications and systems by helping to:

- Simplify the development of distributed applications by providing a consistent set of capabilities that is closer to the set of application design-level abstractions than to the underlying computing and communication mechanisms.
- Provide higher-level abstraction interfaces for managing system resources, such as instantiation and management of interface implementations and provisioning of QoS-related resources.
- Shield application developers from low-level, tedious, and error-prone platform details, such as socket-level network programming idioms.
- Amortize software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable frameworks, rather than rebuilding separate implementations manually for each use.
- Provide a wide array of off-the-shelf developer-oriented services, such as event channel and security services, that have proven necessary to operate effectively in a distributed environment.

- Ease the integration and interoperability of software artifacts developed by multiple technology suppliers, over increasingly diverse, heterogeneous, and geographically separated environments [7].

- Extend the scope of portable software to higher levels of abstraction through common industry-wide standards.

The emergence and rapid growth of the Internet, beginning in the 1970's, brought forth the need for distributed applications. For years, however, these applications were hard to develop due to a paucity of methods, tools, and platforms. Various technologies have emerged over the past 20+ years to alleviate complexities associated with developing software for distributed applications and to provide an advanced software infrastructure to support it. Early milestones included the advent of Internet protocols [49, 50], interprocess communication and message passing architectures [11], micro-kernel architectures [1], and Sun's Remote Procedure Call (RPC) model [72]. The next generation of advances included OSF's Distributed Computing Environment (DCE) [57], CORBA [40], Java RMI [74], and DCOM [5].

The success of middleware technologies has added the middleware development paradigm to the familiar operating system, programming language, networking, and database offerings used by previous generations of software developers. By decoupling application-specific functionality and logic from the accidental complexities inherent in a distributed infrastructure, middleware enables application developers to concentrate on programming application-specific functionality, rather than wrestling repeatedly with lower-level infrastructure challenges.

3.1.1 Limitations of Conventional Middleware

One of the watershed events during the evolution of middleware was the emergence of distributed object computing (DOC) middleware in the late 1980's and early 1990s [61]. DOC middleware represented the confluence of two major areas of software technology: *distributed computing systems* and *object-oriented design and programming*. Techniques for developing distributed systems focus on integrating multiple computers to act as a unified and scalable computational resource. Likewise, techniques for developing object-oriented systems focus on reducing complexity by creating reusable frameworks and components that reify successful patterns and software architectures [6, 16, 64]. DOC middleware therefore uses object-oriented techniques to distribute reusable services and applications

efficiently, flexibly, and robustly over multiple, often heterogeneous, computing and networking elements.

The Object Management Architecture (OMA) described in the CORBA 2.x specification [41] defines an advanced DOC middleware standard for building portable distributed applications. Figure 3.1 shows the architecture of the conventional DOC middleware – CORBA. The CORBA 2.x specification focuses on *interfaces*, which are essentially contracts between clients and servers that define how clients can *view* and *access* object services provided by a server as they would a local object. CORBA also uses the same interface definition syntax to define the interactions between applications and the object request broker's (ORB's) infrastructure.

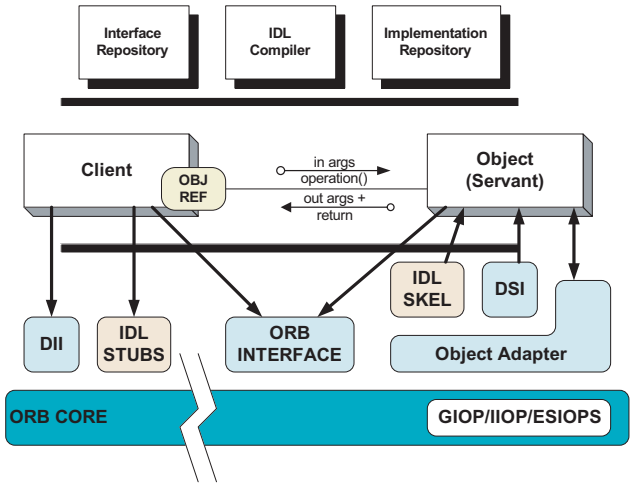

Figure 3.1: **A Conventional DOC Middleware – CORBA 2.x Architecture**

The evolution of conventional DOC middleware has simplified the work of client-side application developers the most. Implementing a non-trivial server application, however, still remains a challenging task with conventional DOC middleware because of a lack of higher-level abstractions for implementing server applications. For example, despite its advanced capabilities, the CORBA 2.x standard does not define sufficiently how to *implement* and *deploy* server objects but only gives a rather simplistic model of the application development lifecycle, as shown in Figure 3.2. The development lifecycle defined by CORBA 2.x specifications essentially divides application development into two stages:

1. **Interface Definition Stage:** a system developer designs the client view of a service, *i.e.*, how external entities should interact with the service, by defining the points of interaction in the CORBA *interface definition language* (IDL).

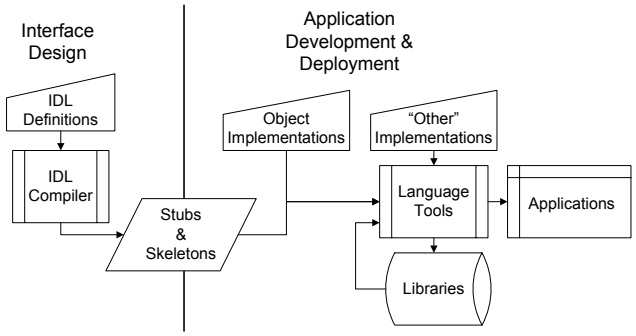

Figure 3.2: **Application Development Lifecycle of CORBA – An Example Conventional Object Middleware**

2. **Implementation Stage:** application developers then use the interface definitions specified by the system developer in Stage 1, and implement

   - clients that access remote objects via those interfaces,
   - object implementations and server processes that provide remote access for the interfaces, or both.

The only tool defined by the CORBA specification to standardize and automate the transition from Stage 1 to Stage 2 is a compiler that translates the IDL definitions into various language mappings for client and server side development in the second stage. The CORBA 2.x specifications concentrate on providing a platform and vendor neutral development environment through a set of standardized interfaces, but stop short of standardizing other higher level elements of the system development process. For example, they do not codify how multiple object implementations can be combined within a server process, how to specify the location of object implementations, or how to establish their inter-connections. As a result, application developers are forced to hard code these aspects into both object implementations and server processes, which hampers the reusability of the resulting software. This under-specification of the application development methodology has led to following limitations [83]:

1. **Lack of generic server standards.** CORBA 2.x does not specify a generic server framework to perform common server configuration and management tasks, such as starting a server process, configuring its QoS policies, providing common services (such as notification or naming services), or managing the runtime environment of each object implementation. Although CORBA 2.x standardizes the interactions between object implementations and object request brokers (ORBs), server developers

must still determine how (1) object implementations are installed in an ORB and (2) ORB and object implementations interact, explicitly by programming them into the server implementations. The lack of a generic component server standard yields tightly coupled, *ad-hoc* server implementations, which increase the complexity of software upgrades and reduce the reusability and flexibility of CORBA-based applications.

2. **Lack of functional boundaries.** The CORBA 2.x object model treats all interfaces as client/server contracts. This object model does not, however, provide standard *assembly* mechanisms to decouple dependencies among collaborating object implementations. For example, objects whose implementations depend on other objects need to discover and connect to those objects explicitly. To build complex distributed applications, therefore, application developers must explicitly program the connections among interdependent services and object interfaces, which requires extra work and which can yield brittle and non-reusable implementations.

3. **Lack of software configuration and deployment standards.** There is no standard way to distribute and start up object implementations remotely in the CORBA 2.x specification. Application administrators must therefore resort to in-house scripts and procedures to deliver software implementations to target machines, configure the target machine and software implementations for execution, and then instantiate software implementations to make them ready for clients. Moreover, software implementations themselves often must be modified to accommodate such *ad hoc* deployment mechanisms. The need of most reusable software implementations to interact with other software implementations and services further aggravates the problem. The lack of higher-level software management standards thus results in systems that are harder to maintain and software component implementations that are much harder to reuse.

3.1.2 Addressing the Limitations of Conventional Middleware

One promising solution that has evolved to address the aforementioned limitations in conventional middleware is component middleware. *Component middleware* [75] is a class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly by

- defining standard runtime mechanisms needed to execute components in generic component servers,
- creating a virtual boundary around larger application component implementations that interact with each other only through well-defined interfaces [4], and
- specifying the infrastructure to assemble, package, and deploy components throughout a distributed environment via well-defined configuration language specifications.

Standardizing these features provides a higher level of abstraction on top of application component interface-level interactions, and enables system developers to specify:

- required properties of their runtime environments,
- the interactions between components, and
- deployment configuration details,

explicitly and declaratively instead of implicitly and programmatically.

The CORBA Component Model (CCM) [38] is an example of component middleware that addresses limitations with earlier generations of DOC middleware. The CCM specification extends the CORBA object model to support the concept of components, and establishes standards for implementing, packaging, assembling, and deploying components. From a client perspective, a CCM component is an extended CORBA object that encapsulates various interaction models via different interfaces and connection operations. From a server perspective, components are units of implementation that can be installed and instantiated independently in standard component server runtime environments stipulated by the CCM specification.

Components are larger building blocks than objects, with more of their interactions managed by the middleware itself. This allows the middleware to simplify and automate key aspects of construction, composition, and configuration of components into applications. By providing these higher level abstractions, CCM enhances CORBA to simplify not only client program development but also server program development. The following section gives a more detailed survey of CCM and summarizes the major mechanisms that facilitate CCM's features.

3.2 Major CCM Building Blocks

A *type* uniquely identifies a group of similar object instances and the operations applicable to these objects. A *metatype* is a type that can be used to describe attributes, including the kind of data and applicable operations, of another type. A fundamental example of a metatype in CORBA is *interface*. The CCM extends and enhances the CORBA Object Model by adding new metatypes, tools and mechanisms. These extensions and enhancements collaborate to address the limitations previously described in Section 3.1.1. The CCM programming paradigm standardizes the application development lifecycle into several stages, notably component implementation, packaging, assembly, and deployment as shown in Figure 3.3, where each stage of the lifecycle adds information pertaining to that stage.

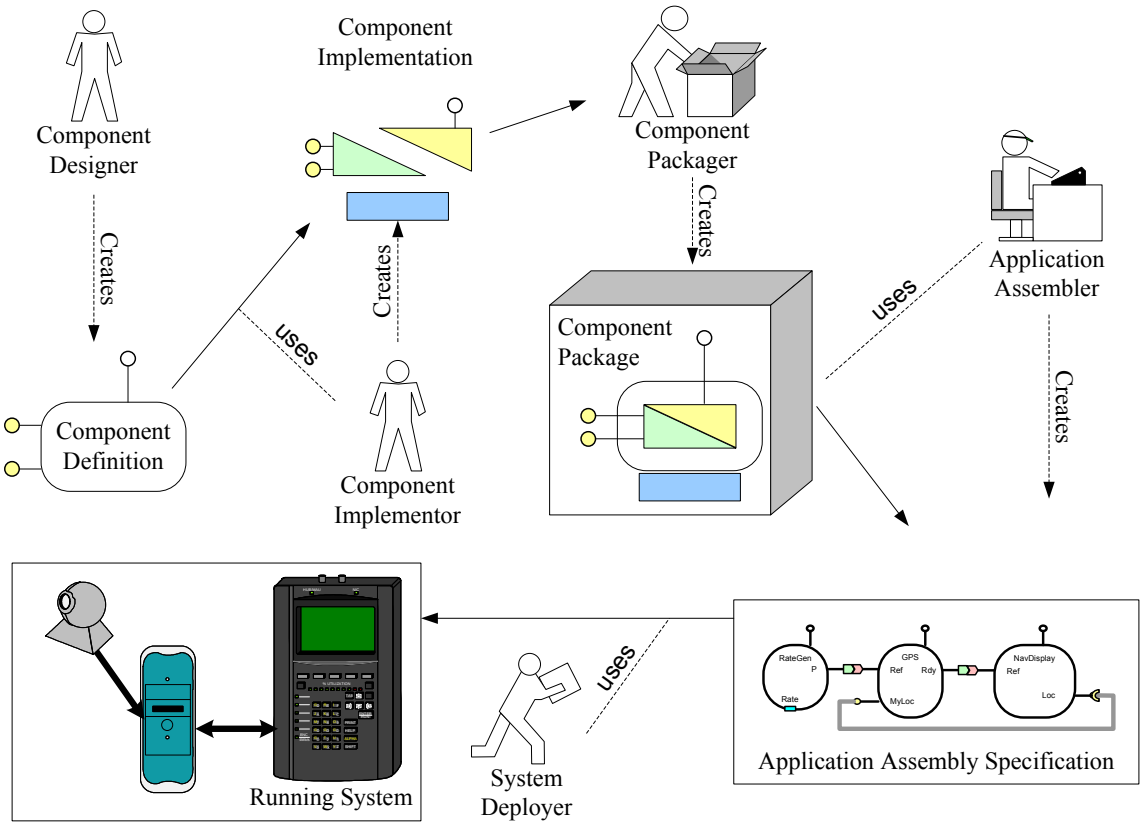

Figure 3.3: **Overview of the CCM Development Lifecycle**

3.2.1 Development Roles

Because of this clear separation, each stage of the component development lifecycle can be handled separately by the following development roles:

- **Component designers** define component features by specifying what each component does and how components collaborate with each other and with their clients. Component designers determine how a component should interact with other entities, including components, clients, or services, by specifying various offered interfaces and required connection points. Component designers, however, do not decide how a component should be implemented to support the features of that component.

- **Component implementors** develop component implementations based on predefined sets of component features. There are also tools to help component implementors generate metadata called *component descriptors* that describe the runtime support needed by each component implementation and the *allowable* component interactions defined by component designers.

- **Component packagers** bundle component implementations with component configuration metadata and component descriptors into *component packages*.

- **Application assemblers** configure applications by selecting component implementations, specifying component instantiation constraints, and connecting component instances via metadata called *assembly descriptors*.

- **System deployers** analyze the runtime resource requirements given in the assembly descriptors, and prepare and deploy required resources where application assemblies can be realized.

This separation of concerns allows developers with different areas of expertise to concentrate on different tasks and thus makes it easier and less error-prone to develop robust server applications. The remainder of this chapter reviews the major building blocks and mechanisms provided by the CCM extensions to facilitate this separation of development roles.

3.2.2 Generic Runtime Support in CCM – Component Servers and Containers

Components are software artifacts that can be distributed, deployed, and instantiated independently. Requiring a component implementation to configure *programmatically* all

the mechanisms that constitute its runtime environment, such as object request brokers (ORBs) and portable object adapters (POAs), limits the reusability of component implementations, as components may very well require conflicting POA or ORB configurations. To separate the concerns of implementing customized server programs, configuring ORBs and POAs, and managing object implementations in those servers, the CCM defines two standard mechanisms called *Component Server* and *Container*.

Component servers are generic server implementations which a CCM framework uses to host components. To provide the proper run-time environments for hosted components, a component server is responsible to configure its ORB to satisfy the requirements of the components. Likewise, containers are the CCM mechanisms for configuring and managing POAs. Together, component servers and containers provide runtime support for component implementations, and thus separate the concerns of programming and configuring servers, ORBs, and POAs from the concerns of component and application development. Figure 3.4 shows the runtime architecture of the CCM container programming model.

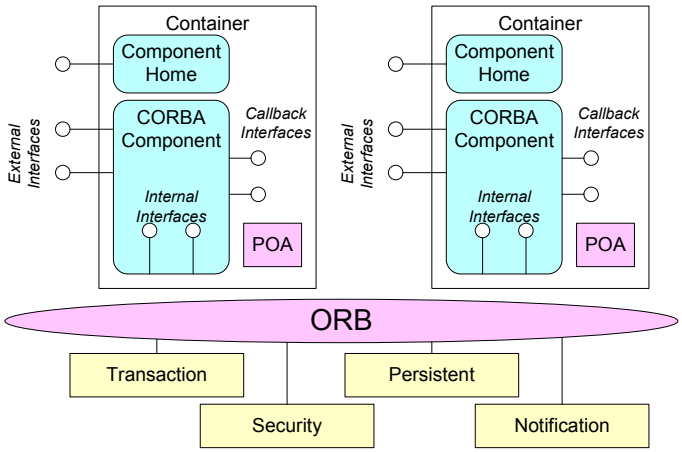

Figure 3.4: **Overview of the CCM Run-time Architecture**

In a conventional CORBA server program, a server process contains an ORB, which in turn creates and manages several POAs under the ORB's RootPOA. Servants, which are object implementations, can then be activated under these POAs. Similar to conventional CORBA servers, a CCM component server contains an ORB and several containers, each of which contains a POA. Component implementations are installed and activated within containers. Each container thus provides an *explicit* encapsulation boundary between each set of servants that implements a component and the services upon which it depends.

With CCM component developers are thus no longer responsible for creating and configuring either ORBs or POAs. Furthermore, there is no longer an easy and standard way for a component implementation to interact directly with the internal object interfaces of the POAs and ORBs, as the container *strongly* encapsulates those details. However, to install and execute the components and component homes dynamically in a standard and generic server process, servers and installed components and homes must know how to interact with each other. Therefore, the CCM defines container interfaces in the "container programming model" to standardize the interactions between components implementations and their containers.

All interactions of a component implementation are handled by its container, including interacting with clients and with the hosting run-time environment. These interactions are handled via 3 kinds of local interfaces that are predefined in the container programming model. These interfaces can be categorized into:

1. **External interfaces:** Client invocations on the component are relayed to the external interfaces provided by the component implementation by its hosting container.

2. **Internal interfaces:** A component implementation interacts with its hosting container through the *internal interface*. These interfaces provide access to container-managed services and context, such as lifecycle management and transactional operations.

3. **Callback interfaces:** Component implementations must implement callback interfaces to allow their hosting containers to inform component instances of important events, such as when a component instance is being made available for service.

3.2.3 Components and Homes

As was mentioned in Section 3.1.2, components are units of implementation that can be installed and instantiated independently in a component server. A component thus encapsulates the implementation of a set of features. The functionality of a component is made available to its clients through a set of interfaces. Likewise, a component may use some external interfaces either to delegate functionality or simply to pass on certain messages.

To capture this encapsulation of implementation, CCM adds a new *component* metatype which extends of the conventional CORBA object interface metatype to represent this encapsulation of implementation. This extension allows components to be manipulated directly in a CORBA framework. The new extension also enables extensions to the IDL

grammar so that component interactions can be standardized and specified in a component's IDL declaration. Component interactions are defined by exposing a set of *ports*. Ports are named interfaces and connection points that define how a component can interact with other entities. The following interfaces and connection points, shown in Figure 3.5, constitute the different kinds of ports:

- **Facets**: Facets allow a client to invoke operations on the component. Specifically, facets define named interfaces that service method invocations from other components synchronously. Facets also allow a component to present different identities and capabilities to different clients. A component uses the keyword *provides* to specify the name and the type of each facet it exposes.

- **Receptacles**: A receptacle stores the object references with which the component can invoke operations. Specifically, receptacles provide named connection points that can accept one or more object references pointing to plain interface implementations, or facets of other components. A component uses the keyword *uses* to specify the names and types of receptacles it exposes. A receptacle can be specified to have either a single connection or multiple connections.

- **Event sources/sinks**: An event sink allows messages to be delivered to the component via invoking a *push* operation. Similarly, an event source stores the object reference(s) to the consumer the component can send messages to by invoking the *push* operation. A component uses event sources and event sinks to indicate its willingness to exchange event messages with other components or other event publishers or consumers asynchronously. Unlike with the Event [48] and Notification [22] CORBA Common Object Services, the event sources and event sinks in CCM components are typed using the native CORBA valuetypes. A component uses the keywords *publishes*, *emits*, and *consumes* to specify the names and event types of event sources and sinks.

These port mechanisms enable component developers to employ the Extension Interface pattern [64] to upgrade and replace component implementations without breaking existing applications that use them.

Another important extension in CCM is the addition of a *home* interface. The home interfaces help to standardize lifecycle management for component instances. Each component instance must be created and managed by a home interface. Developers can use different home interfaces and implementations to support different lifecycle management strategies for a component.

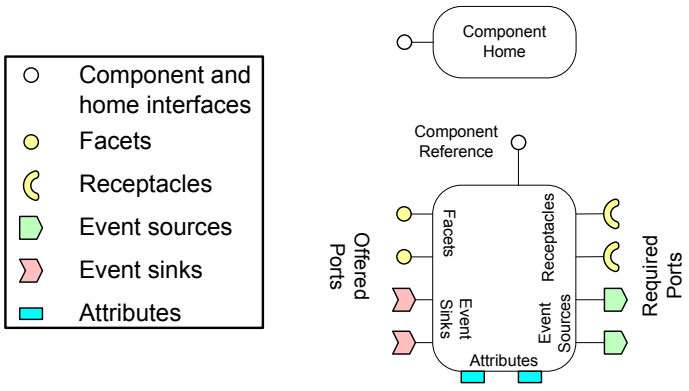

Figure 3.5: **Client View of CCM Components**

Both components and component homes can have *attributes*, which are named parameters like those of regular object interfaces and are intended for component configuration. The CCM deployment mechanisms described in Section 3.2.6 use XML-based component property file descriptors to configure the initial values for attributes in components and component homes.

By standardizing features and interfaces for components, ports, attributes, and homes, component clients and component developers now are able to figure out what component features are available and how to access these features. The CCM standard also defines a set of generic port operations that allows standardized tools to query component capabilities, configure component attributes, acquire port interfaces, and connect to port connection points. These standardized component features are vital to allow components to define crisp implementation boundaries while giving them flexibility to be composed and deployed into running applications using automatic tools.

3.2.4 Component Implementation

Section 3.2.3 alluded to many key features that CCM components must support, including generic, port-specific, and introspective operations. These additional features and operations allow CCM components to be assembled into applications easily with tools. Likewise, as illustrated in Section 3.2.2, a component implementation must support and use several local interfaces to interact with its hosting container and for the component implementation to be installed and instantiated.

Furthermore, some features of components require them to maintain certain context information, such as lists of connected interfaces and event consumers. Implementing

these additional interfaces and operations required by component and container programming models can make implementing components a daunting and error-prone task. Furthermore, some of the context management code and the type of container interfaces a component uses are dependent on how a component should be used, *e.g.*, on whether or not the component implementation is stateful and whether or not a servant instance can be shared by multiple object references [51]. These complexities could greatly hinder CCM's goal to separate concerns and development roles, as component developers would then have to implement code to interact with these additional systemic aspects, such as connection management and interface activations in the container programming model.

To address these issues and thus make implementing components easier, the CCM specifies a component implementation framework (CIF) to standardize and automate the generation of most servant implementations. To achieve this goal, the CIF defines the Component Implementation Definition Language (CIDL) to define how a component should be used, *i.e.*, what constitutes a component implementation type.

A CIDL compiler can then be used to generate automatically the component servant implementation which realizes the majority of the component management functionality but leaves the implementation of application-specific operations to entities called *executors* that are implemented by component developers. The forwarding of operations on facets and event sinks follows a set of executor mapping rules that define the interfaces and their operation signatures of executors. The CIF thus further separates the concerns of the component implementation itself from those of the component management code.

Figure 3.6 shows how the container, generated servant code, and developer-implemented executor(s) interact via various pre-defined container interfaces. When running a component, the generated servant is activated in the POA of the container, much like how a servant can be activated in a POA in a conventional CORBA program. The generated servant in turn contains references to developer-implemented executors, as well as a component-specific context object which is also generated by the CIDL compiler. The component-specific context object implements connection management code for the component implementation. Based on the CIDL definition, a component-specific context interface inherits an *internal interface* defined in the container programming model, and provides access to container managed services.

The CIDL compiler also generates the component-specific executor interface definitions that component developers can use to implement component executors. Like the component-specific context, based on its CIDL definition, a component executor inherits

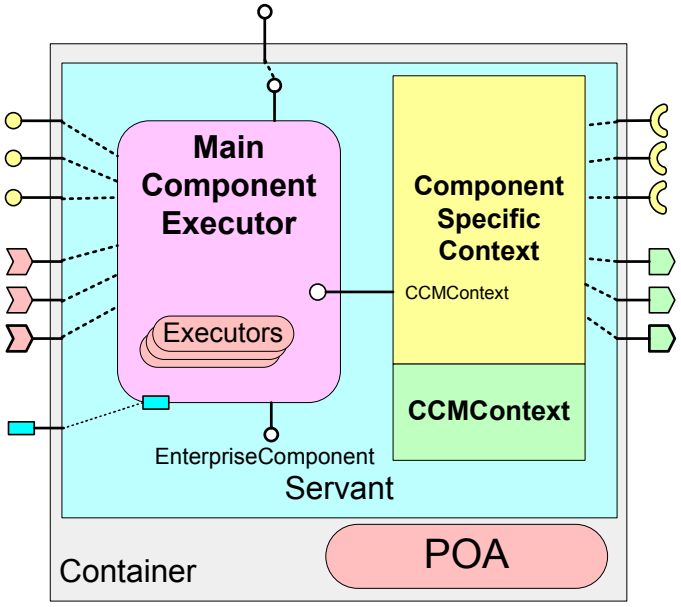

Figure 3.6: **Servant and Context Code Generated by CIDL Compiler**

from a *callback interface* defined in the container programming model to provide an interface the container framework can use to notify executors of important events. In summary, the component management code generated by the CIDL compiler assumes responsibility for the following features of the component implementation:

- managing the lifecycle of object references and executors,
- accepting operation invocations on interfaces and event consumers and forwarding those invocations to the corresponding executors, and
- managing interface and event publisher connections which executors can use to invoke operations.

Such automation simplifies the component implementation process, as component management code can now be generated automatically using simple declarative directives instead of having to iterate through all the interfaces and connection points and implementing the necessary management code manually.

Figure 3.7 shows how the CIDL compiler works with other tools and source files to generate component implementations. As shown in the figure, CIDL also generates the component descriptor files, which are XML-based documents that contain metadata describing component capabilities, *i.e.*, what ports and attributes are available, and runtime

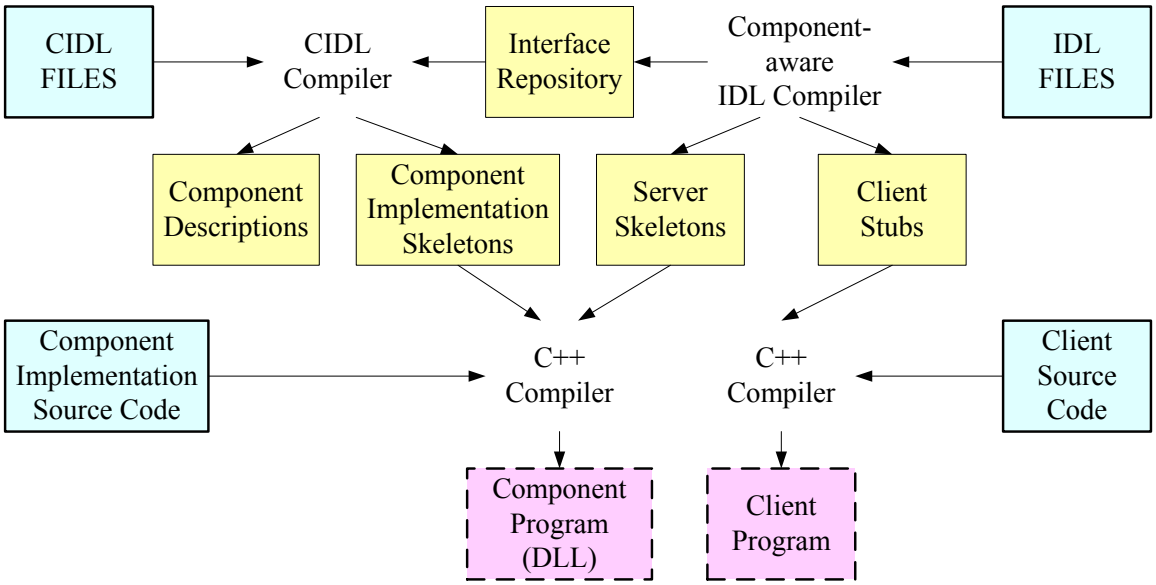

Figure 3.7: **Flow of CIDL and IDL Generated Files**

requirements, *e.g.*, the type of container a component needs. Component descriptors provide information that is vital later to application assembly, packaging and deployment, as is described in Sections 3.2.5 and 3.2.6.

3.2.5 Composing Components into Applications

One major feature of component middleware is the ability to compose component implementations together into applications. The majority of component features illustrated in Section 3.2.3 are designed to facilitate such composition. The CCM specification standardizes the syntax and structure of XML-based metadata, namely in *assembly descriptors* for describing component-based application assemblies. Such standardization allows tool vendors to develop a variety of application assembly tools, as long as they generate conforming assembly specifications.

For example, CIAO facilitates two collaborating projects, both of which involve software modeling tools aiming to generate CCM standard assembly specifications with optional CIAO-specific metadata which will be discussed in more detail in Chapter 5. Of the two projects, the CoSMIC [30] project from the Distributed Object Computing (DOC) Group in the Institute of Software Integrated Systems (ISIS) at Vanderbilt University offers

a GUI-based composition and modeling front end and plans to synthesize CIAO configuration settings. The Cadena [23] project from the Specification, Analysis, and Transformation of Software (SAnToS) Laboratory at Kansas State University offers tools for both GUI-based and table-based composition and analysis. Both of these component assembly tools can obtain component information either from a component-aware Interface Repository or from the component descriptors accompanying component implementations, which were generated by a CIDL compiler.

An assembly descriptor file consists of 3 major XML elements, each describing an important aspects of the assembled systems. These elements are as follows:

1. **Component Implementations** describe the specific component implementations the assembled application requires. An assembled application may even use several different implementations of the same component definition for different instances of the component.

2. **Component Placements** specify location constraints on installing component home instances, and thus on the component instances created by the home instances. Location constraints are specified in form of collocations, *i.e.*, they specify sets of homes that should be installed on the same host or the same process. They can also be in the form of an exact location. Component instances can also be instantiated from homes explicitly.

3. **Component Connections** describe how component and home instances should be connected. Connections are made by passing an object reference, *e.g.*, to a facet, an event consumer, a receptacle or an event source for a component. Other than acquiring object references from component instances created in the same assembly, assembly descriptors also support other mechanisms for locating an object reference. For example, an assembly can also specify attributes needed to acquire an object reference from a Trading service.

CCM's assembly descriptors provide key information about what component homes and components need to be instantiated and how to connect them and other software elements together to form an application. They do not, however, contain information about where on the network, or on what specific hardware platforms, operating systems, or languages, the application should be installed. By generalizing assembly descriptors away from including explicit information related to specific instantiations of the application, CCM enables system deployers to use assembly descriptors to deploy an application on alternative environments and networks, as is described in the next section.

3.2.6 Packaging and Deployment

Standardizing the packaging and deployment of component-based applications is one of the major goals of the CCM development paradigm. Packaging and deployment standards in CCM separate the concerns of distributing, installing and instantiating software applications from the concerns of designing and implementing those applications. The packaging specification provides a standard way to collect, aggregate, and associate disjoint information into integrated entities that can be distributed intact. The deployment mechanisms defined in the CCM specification standardize the interfaces and operations for realizing applications, including distributing, installing, deploying, and connecting applications.

A *software package* in CCM is a compressed file that contains a collection of software implementations, their associated metadata, and an XML-based descriptor that details the contents of the software package. The CCM specification defines two major types of software packages. They are:

- **Component package:** A component package contains the information needed to deploy a specific implementation of a component. The major descriptor for a component package is called its "Software Package Descriptor" which includes pointers to component metadata, *i.e.*, the component descriptor, dynamic loadable libraries of the implementation for different platforms and programming languages, and default property descriptors for components and component homes.

- **Application assembly package:** An application assembly package contains the information needed to deploy an application consisting of multiple interconnected components. The major descriptor for an application assembly package is the "assembly descriptor" mentioned earlier in Section 3.2.5. Other than the embedded information about component implementations, component placements, and connections, assembly descriptors reference component implementations using component packages, and contain pointers to component property descriptors that can be used to overwrite the default property descriptors in components packages.

The deployment mechanisms defined by the CCM specification offer a path to distribute software packages and to install them on target platforms. To interact with the various runtime mechanisms described previously in this section, the CCM specification defines a series of interfaces for these mechanisms, such as `ComponentServer` and `Container`, in order for the deployment tools to interact with them directly. Likewise, *interfaces* of major entities for distributing and deploying applications are defined, instead

of the actual tools themselves. This abstraction of deployment and runtime interfaces allows deployment tools and runtime environments from different sources to inter-operate together.

3.2.7 CCM Development Lifecycle

The conventional CORBA development lifecycle shown in Figure 3.2 simplifies client program development, as that 2-stage development model provides adequate abstractions for building client programs. For server development, however, the conventional CORBA paradigm is insufficient for non-trivial systems that involve integration of multiple aspects in the resulting server programs. By standardizing the tools and the process of component development, the CCM specification greatly enhances the CORBA development lifecycle by separating these concerns into the different stages of the development lifecycle shown in Figure 3.8. These development stages include:

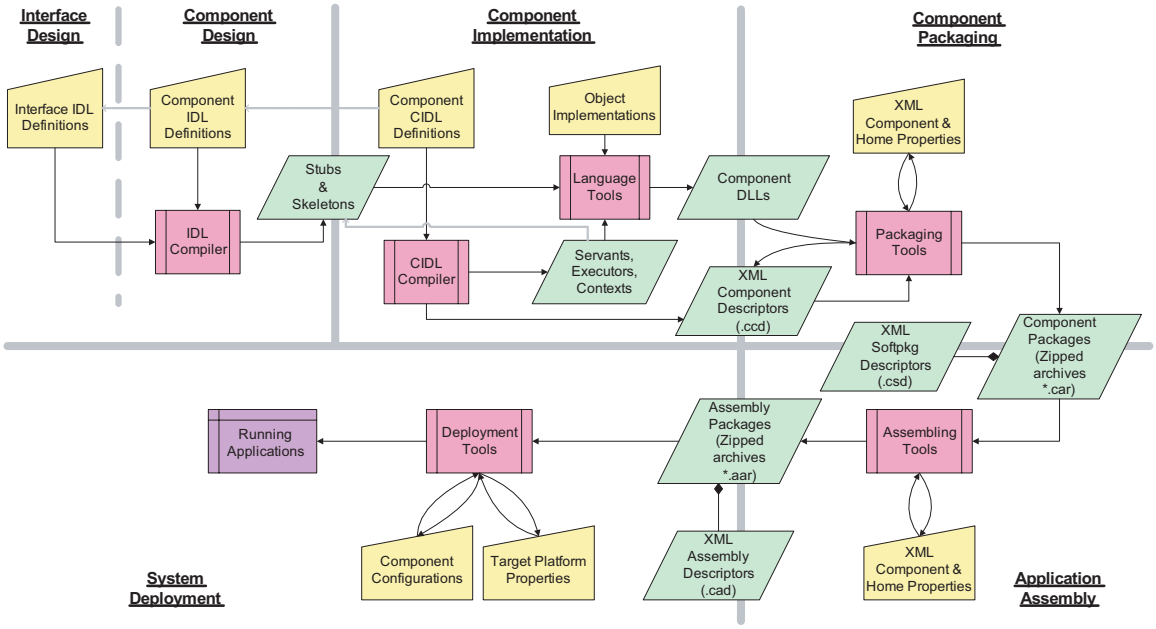

Figure 3.8: **Overview of the CCM Development Lifecycle**

- Interface design
- Component design
- Component implementation

- Component packaging
- Component assembly
- System Deployment

This separation of concerns into different stages allows developers to deal with one aspect at a time. In particular, with this paradigm different developers (possibly with different areas of expertise) can effectively divide a complex development problem into separate aspects and thus address each aspect most effectively.

3.2.8 Status of CCM Development

The CCM specification was finalized recently by the OMG and is in the process of being incorporated into the core CORBA specification.[1] There are, however, many remaining open research issues for the further evolution of the CCM. These research issues falls into following major categories:

1. **Supporting new systemic aspects not currently covered in the CCM specification:** Current CCM specification is designed for enterprise applications. It is necessary to extend its support for other systemic aspects in order to apply CCM in other application domains.

2. **Providing robust and flexible deployment and configuration model:** Existing CCM specification does not adequately define the process to deploy and configure component applications which may lead to implementations that are not flexible or depends proprietary extensions.

3. **Modeling and synthesizing component application assemblies:** Although application assemblies in CCM are defined in the human-readable XML format, they are verbose and non-intuitive to write, to reason and to analyze. It is necessary to define different way to represent these assemblies.

Researchers and practitioners are beginning to address these open research issues by experimenting with several different approaches in their CCM implementations. Some of the approaches have already begun to undergo standardization. Other than the standardization efforts described previously described in Section 2.4, several CCM implementations are now available based on the recently adopted specification [38]. They include:

[1]The CORBA 3.0 specification [40] released by the OMG only includes changes in IDL definition and Interface Repository changes from the Component specification.

- **OpenCCM** [77] developed by the Universite des Sciences et Technologies de Lille, France, is a Java implementation of CCM. It addresses deployment and configuration issues in the context of a Java CCM.

- **K2 Containers** developed by iCMG, is a commercial C++ based CCM implementation which addresses deployment and configuration issues by providing its own interpretation of the specification. It also provides tools for visually creating component application assemblies.

- **MicoCCM** developed by FPX, is also a C++ based CCM implementation. It addresses both deployment and configuration issues by providing deployment tools and the modeling issue by providing a visual application assembly tool.

- **Qedo** [54] by Fokus, is a C++ based CCM implementation that focuses on addressing the issue of providing new systemic aspects to support QoS for applications. Its approach for new systemic aspects focuses on extending the container programming model and requires rewriting component implementations to take advantage of the added QoS control capability. This approach is closer to the forthcoming QoS for CCM standard. Qedo also provides deployment and configuration tools.

- **StarCCM** [70] is a C++ based CCM implementation at an early stage. It currently focuses on the task of integrating transaction behaviors and persistent component state into the infrastructure.

- CIAO developed by the DOC groups at Washington University in St. Louis and the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, is the major vehicle of this research. CIAO concentrates on providing new systemic aspects via composition, which does not require new component implementations to adopt new systemic behaviors. Furthermore, CIAO provides a robust deployment framework which is described later in Section 5.3 of this dissertation.

The architectural patterns used in CCM [78] also appear in other popular component middleware technologies, such as J2EE [32, 3] and .NET. Research conducted in the context of CCM can often be applied to other component-based technologies.

Chapter 4

Composing Systemic Aspects in CIAO

Chapter 3 motivates the need for component-based middleware and presents an overview of a component-based middleware framework, *i.e.*, the CCM. The current generation of component-based middleware frameworks was designed for enterprise applications and therefore does not consider the needs of other application domains. This chapter describes the limitations from this lack of support for other application domains, and motivates the solution presented in this dissertation – the ability to compose arbitrary systemic aspects within the component model itself. To illustrate these limitations, we use distributed real-time embedded (DRE) systems as our motivating application domain.

This chapter begins by noting several limitations of conventional component middleware, in Section 4.1. Section 4.2 gives a brief discussion of real-time middleware aspects, and Section 4.3 then describes the problem of integrating support for those aspects within the CCM programming model. Section 4.4 examines from a programming perspective how real-time aspects can be composed and integrated within the CIAO framework. Finally, Section 4.4.3 illustrates how CCM can be used to implement components and to compose and deploy applications.

Together, the last three sections of this chapter illustrate how in general the approach presented in this dissertation can be applied to overcome the limitations of conventional component middleware by supporting integration of other systemic behaviors and other component middleware frameworks. Chapter 5 then presents the implementation of CIAO in detail.

4.1 Limitations of Conventional Component Middleware

Historically, conventional commercial off-the-shelf (COTS) middleware frameworks, such as CORBA, are designed to address the needs of enterprise application domains like workflow processing, inventory management, and accounting systems. Other application domains, however, usually require additional constraints for applications to be considered to behave "correctly". For example, over 99% of all microprocessors are now used for DRE systems [2] that control processes and devices in physical, chemical, biological, or defense industries. Examples of DRE applications include distributed sensor networks, flight avionics systems, naval combat management systems, and financial trading systems, all of which typically have stringent Quality of Service (QoS) requirements. For example, certain tasks of these applications must operate in a timely manner or else these applications would not be considered to be "working."

Failure to meet these QoS requirements can lead to catastrophic consequences, such as failing to detect an incoming threat in a combat management system. Research in QoS-enabled middleware over the past few years has shown that the coordinated management of application and system resources is an essential dimension for ensuring QoS. Conventional middleware, however, does not provide adequate abstractions to control the mechanisms for managing these behaviors and thus is not suitable for applications in these domains.

As the cost and time required to develop and verify applications from these domains precludes developers from implementing these applications from scratch, communities from these application domains have been trying embrace COTS middleware frameworks by extending the middleware specifications to provide better abstractions for controlling and managing these domain-specific aspects. For example, the Real-time CORBA 1.x specification [39] extends the conventional CORBA specification to provide better abstractions for managing resource access for DRE applications requiring timely response of critical tasks. As COTS middleware evolves and matures in supporting these extensions, it is increasingly gaining acceptance within the DRE systems community.

Although these domain-specific middleware extensions standardize the mechanisms and interfaces for managing and controlling the QoS policies, they still don't provide a complete set of configuration abstractions for application developers. As a result, developers are forced to integrate the code to configure these QoS policies in an *ad hoc* way, often by scattering the configuration code throughout the application code. This limitation is much like the limitation of conventional DOC middleware outlined in Section 3.1.1. Such

limitations in turn result in systems that are harder to maintain and software component implementations that are much harder to reuse.

Furthermore, as the use of COTS middleware becomes more pervasive, DRE applications are increasingly being combined to form distributed systems that are joined together by the Internet and intranets. Examples of these types of DRE applications include *industrial process control systems*, such as hot rolling mill control systems that process molten steel in real-time, and *avionics systems*, such as mission management computers [67, 68] that help pilots with navigation and other key avionics functions as shown in Figure 4.1. The kind of control systems often consist of multiple interacting software components run-

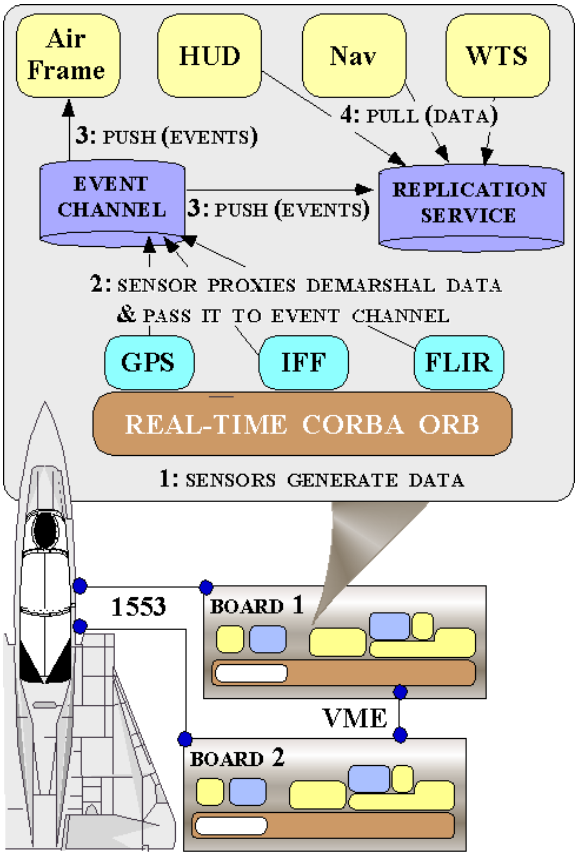

Figure 4.1: **A Typical Avionics System**

ning concurrently at multiple rates. Allocating resources to ensure critical tasks always finish in time is essential to the correct functioning of such "system of systems" for *military command and control systems* that gather and assimilate information from various devices (such as unmanned aerial vehicles and wearable computers), present and analyze the information, and coordinate the deployment of available forces and weaponry. For complex

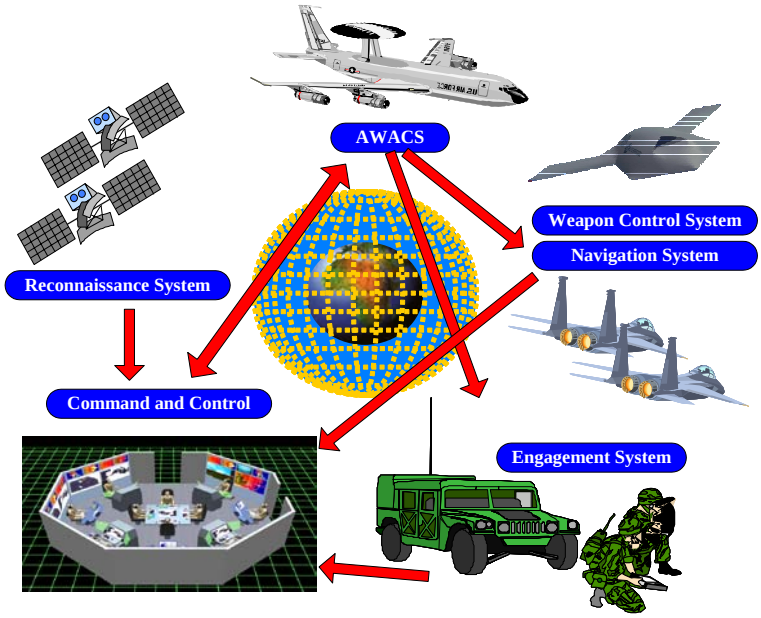

Figure 4.2: **Characteristics of Military Systems of Systems**

systems like this, it is important that the QoS requirements for all the consistituent systems within larger systems can be met systematically. A QoS-aware CCM would make reasoning about and allocating resources for such large-scale complex systems possible.

Many DRE-relevant QoS properties, however, are not considered in the CCM specification. It is certainly possible for component developers to take advantage of middleware or OS features to implement component support for special systemic aspects, *i.e.*, by embedding management support code within a component implementation. Many QoS capabilities, however, cannot be implemented solely within a component due to the limitations listed in Table 4.1.

In general, isolating QoS provisioning functionality into each component prematurely commits every implementation to a specific QoS provisioning scenario. This tight coupling defeats one of the key benefits of component middleware: *separating component functionality from system management*. By creating dependencies between application components and the underlying component framework, component implementations become harder to reuse, particularly for large-scale DRE systems whose components and applications have stringent QoS requirements. Moreover, many resources required for QoS assurance must be provisioned within larger execution units overall, such as a process or a host.

Table 4.1: **Limitations of Conventional Component Middleware's Support for QoS Aspects**

Limitations
QoS provisioning must be done end-to-end, *i.e.*, it needs to be applied to many interacting components. Implementing QoS provisioning logic internally in each component hampers reusability.
Some resources, such as thread pools in Real-time CORBA 1.0 [39], can only be provisioned within a broader execution unit, *i.e.*, a component server rather than a component. Since component developers often have no *a priori* knowledge about other components, the component itself is not the right place to provision QoS.
Some QoS assurance mechanisms, such as checking whether rates of interactions between components violate specified constraints, affect component interconnections. Since a reusable component implementation may not know how it will be composed with other components, it is not generally possible for components to perform such QoS assurance in isolation.
Many QoS provisioning policies and mechanisms require installation of customized ORB modules to work correctly. However, some requirements such as high throughput and low latency, may involve inherent trade-offs. It is hard for QoS provisioning mechanisms implemented within components to foresee incompatibilities without knowing the end-to-end QoS requirements *a priori*.

Based on historical experience with building large-scale research and production DRE systems over the past two decades [63], in the next generation of large-scale DRE systems the middleware – rather than operating systems or networks alone – will likely be responsible for separating QoS systemic properties from functional application properties and coordinating QoS across various DRE system and application resources end-to-end. There is thus a compelling need for extensible component middleware frameworks that can be extended to support different QoS policies that are needed to support applications from different domains.

4.2 Real-time Aspects

As was previously mentioned in Section 4.1, DRE applications often have stringent real-time requirements, such as bounded response latency, that need to be satisfied in order to be considered correct. Example tasks for managing real-time requirements include:

- Pooling of concurrency resources and synchronization of concurrent operations,
- Sensor input and actuator output timing constraints,
- Allocation, scheduling, and priority assignment of computing and communication resources end-to-end,
- Memory management.

The work in this dissertation, therefore, focuses on allocating and managing key real-time resources that are crucial to meet the time constraints in DRE applications. This section provides a brief overview of the need and capability of real-time mechanisms for managing real-time resources in DRE applications. The following Section 4.3 and Section 4.4 will then examine how these capabilities can be integrated into component middleware.

Conventionally, real-time applications for DRE systems have been customized through tedious and error-prone manual development processes to implement code for managing these real-time requirements. This approach fails to address several key challenges facing developers, such as the accidental complexity caused by different hardware and OS configurations and the ever changing system requirements and operating contexts.

Strict control over the scheduling and execution of these resources is essential for many fixed-priority real-time applications. Real-time middleware, such as RT-CORBA, enables client and server applications to (1) determine the priority at which invocations will be processed and (2) allow clients and servers to allocate resources for processing and servicing invocations of different priorities. When used in the appropriate environment, real-time middleware features help application developers and integrators configure heterogeneous systems to preserve priorities end-to-end. Key features of real-time middleware include a portable priority system, and mechanisms for controlling processor resources and inter-process communication as described below [66, 53].

Priority-based Systems

Real-time systems often use priorities to indicate the relative importance of completing certain tasks before others and to allocate resources to process these tasks to ensure their priorities are enforced. Priority inversion is a scheduling hazard that occurs when a thread or request blocks the execution of a higher priority thread or request, and possibly prevents it from completing its task in time. To reduce end-to-end priority inversion, as well as to bound latency and jitter for applications with stringent real-time QoS requirements, it is necessary to provide a consistent view of priorities across heterogeneous platforms. Real-time middleware, such as RT-CORBA, provides platform-independent mechanisms to specify the priority of operation invocations and resource allocations. These mechanisms include the following key features:

- **Priority Type System:** Most real-time operating systems define their own unique priority type systems which have different ranges or priority values, different directions of higher priority, and different base priority levels. It is very hard to assign priority levels in applications running on different real-time operating systems, let alone hoping the priority level will remain valid when propagating over multiple ORB endsystems running on different real-time operating systems. To address this problem, real-time middleware must define a priority type that (1) is portable across different OS-level priorities, and (2) can be interpreted end-to-end across local endsystems. For example RT-CORBA defines the *CORBA priority* type to meet these requirements, as Section 4.2 discusses in detail.

- **Priority Assignment Model:** A real-time middleware should support different strategies for programmers to assign the priority of an execution thread to accommodate various usage scenarios flexibly. For example, a server might want to pre-define the priorities of requests coming in from different clients. Alternatively, a server object may need to service requests coming from clients whose requests need to be handled at different priorities. In this case, the client should define the priority level a request should be handled. The invoked server, in turn, should honor the client's request and handle the invocation at the requested priority level. This approach allows the priority level of the invoking thread to be propagated to many server objects along the invocation path.

Processor Resource Management

Strict control over the scheduling and execution of processor resources is essential for many DRE applications. The priority type system described in this section provides a mechanism for real-time middleware endsystems to schedule tasks consistently. These endsystems, however, also need the mechanisms to allocate processor resources that can be used for executing scheduled requests.

Real-time middleware often pre-allocates CPU resources for executing tasks of a certain priority as thread pools to leverage the benefits of multithreading. Pre-allocating thread pools avoids the extra overhead of dynamic resource allocation and helps prevent priority inversions, as creating threads requires access to global thread management data that is shared among all priority levels. Moreover, real-time middleware can provide mechanisms to set certain thread pool attributes to limit consumption of memory resources, such as thread stack size and request buffering.

Communication Resource Management

Historically, DOC middleware, such as CORBA, provides an abstraction for object connections that hides the nitty-gritty details of connecting and communicating between client and server processes. This abstraction enables location transparency which relieves applications from differentiating the actual location of an object. Treating the underlying OS, network, and/or bus as an encapsulated "black box" allows DOC middleware to provide various optimizations and strategies, such as on-demand connection establishment and connection multiplexing, for minimizing resource utilization without requiring application programmer intervention.

Although this encapsulation is useful for applications with best-effort QoS requirements, it is inadequate for applications with more stringent QoS requirements. For DRE applications that demand predictable behaviors, it is often necessary to manage and control connections, the underlying communication protocols and the endsystem resources to avoid priority inversions when requests at various priorities queue up in the same connection. It is therefore necessary for real-time middleware to provide mechanisms that enable application to manage and control object interconnections when desired. For example, a DRE application may need to use multiple communication channels to avoid priority inversion. An application may request certain connections to be established in the configuration phase before the full system goes into the production phase to avoid unpredictable delay incurred by on-demand connection establishment.

Real-time CORBA Capability

The Real-time CORBA (RT-CORBA) 1.x specification [39] extends the traditional CORBA specifications by defining standard features shown in Figure 4.3 that support end-to-end predictability for operations in *fixed-priority* CORBA applications. RT-CORBA includes standard interfaces and QoS policies that allow applications to configure and control the processor, communication, and memory resources allocated to applications. Major real-time features supported by RT-CORBA include:

- **Priority system**: RT-CORBA support priority aspects to denote the relative importance of tasks. The priority type in RT-CORBA has a fixed range from 0 to 32767 inclusive, with 0 representing the lowest priority. This provides a consistent view of priority levels for all interacting applications in a system. To map a CORBA priority level to a native priority value supported by the local operating system, an application can use one of the mapping strategies predefined by RT-CORBA, or can provide

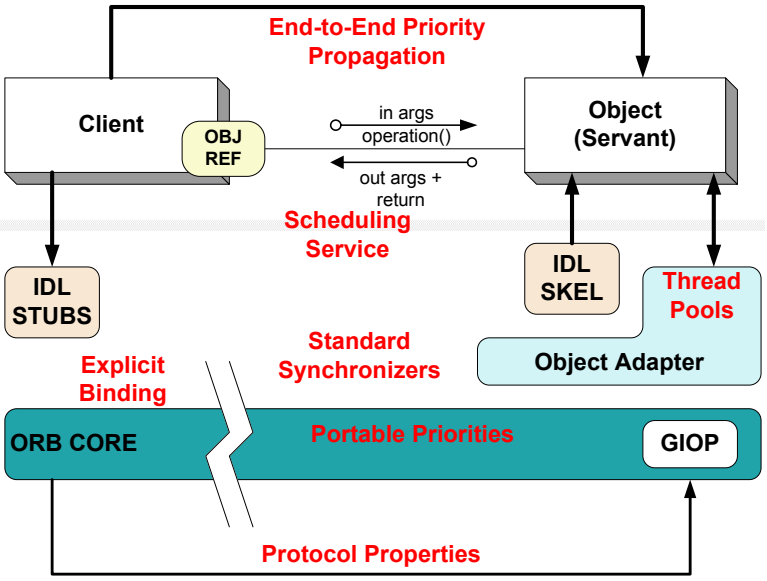

Figure 4.3: **Key Features of RT-CORBA**

its own *custom* priority mapping strategy. RT-CORBA also provides the two priority assignment models for determining operation priorities in an application, *i.e.*, `SERVER_DECLARED` and `CLIENT_PROPAGATED`.

- **Processor Resource Management**: RT-CORBA supports pre-allocation of processor resources in the form of thread pools, for DRE applications to leverage the benefits of multithreading. This mechanism allows server applications to pre-allocate pools of threads and to set certain thread attributes, such as default priority levels, stack size, and buffering policy. It is also possible to bound the priority of ORB threads, which is useful when combining a CORBA server with other real-time threads that may need to run in a particular range of priorities.

- **Communication Resource Management**: The Real-time CORBA specification defines standard interfaces to control the communication resources as described in Section 4.2. These standardized interfaces allow an ORB endsystem to:

  – Establish priority-banded connections to demultiplex traffic at different ranges of priorities into different connections. This prevents messages of different priorities from queuing up in the same connection buffer. When using this policy, a server ORB will create multiple endpoints to accept incoming requests using different connections. A client ORB also will select or establish a connection

for an out going request based on the priority of the request. Combined with the priority type system and the processor resource management features of Real-time CORBA, this ensures that end-to-end priorities are maintained and that key sources of priority inversion are eliminated.

– Select and configure protocol properties of an ORB endsystem. Real-time CORBA defines an interface that applications can use to specify ORB- and transport-specific protocol properties that control various communication protocol features, for example, to reserve connection bandwidth or to set up packet delivery priorities for connections of different priorities.

Other than the aforementioned features that affect both client and server side ORB endsystems, the following features allow a client ORB to:

– Pre-establish connections binding to server objects. Real-time CORBA defines an explicit binding mechanism that uses the `_validate_connection` operation defined in CORBA::Object. Pre-connecting to server objects provides more predictable behavior as connection establishment requests will not be queued and are usually performed before the system is fully configured.

– Establish non-multiplexed private connections to an object. This feature allows the client ORB to establish private connections to server objects. These connections will not be shared for other object invocations.

Using these two features allows the client ORB to manage network connections between client and server ORB endsystems efficiently.

Although real-time middleware provides mechanisms to configure and control underlying OS support for allocating an application's real-time resources and to meet its real-time requirements, it still lacks sufficient higher level abstractions to separate real-time QoS policy configurations from application functionality. Developers using real-time middleware are therefore forced to mix the QoS policy configuration code in the application code. It is thus hard for developers to configure, validate, modify, and evolve complex DRE systems consistently using traditional real-time middleware, such as Real-time CORBA.

4.3 Addressing CCM Limitations - Extending Support for Real-Time Aspects in CIAO

As Section 4.1 alludes, QoS provisioning requires a component model that is aware of applications' QoS requirements. Supporting the interfaces and mechanisms for QoS provisioning in the underlying operating systems and ORB in a component middleware framework does not provide adequate support for developing and deploying applications with these QoS requirements as was summarized in Table 4.1

To address the limitations of conventional component middleware in supporting DRE application domains, it is necessary to make QoS provisioning policies an integral part of component middleware to decouple QoS provisioning policies from component functionality. This dissertation extends conventional component middleware to support composing these systemic aspects and mechanisms into applications. This extension separates the concerns of managing QoS resources from those of component development as it is no longer necessary to tangle QoS management code within multiple components and system modules that can span across multiple programs over the network. This extension also makes component implementations more robust and reusable, as QoS provisioning via systemic aspects can now be composed with applications in ways invisible to the component implementations.

The middleware technologies discussed in this dissertation apply a range of *generative* development techniques [10] to support the separation and composition of QoS systemic behaviors and configuration concerns. Aspect-oriented design techniques in particular are important since code for provisioning and enforcing QoS properties in traditional DRE systems is often spread throughout the software and usually becomes tangled with the application logic. This tangling makes DRE applications brittle, hard to maintain, and hard to extend with new QoS mechanisms and behaviors for changing operational contexts. As Section 4.1 discusses, as DRE systems grow in scope and criticality, a key challenge is to decouple reusable, multi-purpose, off-the-shelf, resource management aspects from aspects that need customization and tailoring to the specific preferences of each application. The solution presented in this dissertation decouples QoS-aspects from application objects and allows them be composed later in the system lifecycle.

The CIAO framework presented in this dissertation is a "QoS-enabled" CCM implementation which separates the programming and provisioning of QoS concerns. In conventional component middleware, such as CCM described in Section 3.2, there are multiple software development roles, such as component designers, assemblers, and packagers.

QoS-enabled component middleware supports yet another development role, *i.e.*, the *Qosketeer* [87] who is responsible for performing QoS provisioning, such as preallocating CPU resources, reserving network bandwidth/connections, and monitoring/enforcing the proper use of system resources at runtime to meet or exceed application and system QoS requirements. The following section provides a more detailed examination of how to support composition of real-time policies and mechanisms, starting with the CCM framework as a blueprint.

4.4 Composing Real-time Behaviors into CIAO Applications

To provision end-to-end QoS robustly throughout a component middleware system and to improve component reusability, QoS provisioning specifications should be decoupled from component implementations and specified instead in component composition meta-data. This section provides solutions to two major design and implementation challenges, *i.e.*, real-time policies can be applied at different levels of granularity and at different stages of the system lifecycle, in composing real-time behaviors into component middleware. It reviews how policies can be applied with different granularities in CIAO during different stages of application development lifecycle. It provides an analysis to show how CIAO composes and applies real-time policies throughout the CCM application development lifecycle. Finally, it illustrates the impact of these features through a simple yet illustrative example application with both functional and real-time composition requirements.

4.4.1 Challenge 1 – Granularity in Policy-based Configuration Mechanisms

Within a real-time middleware endsystem, resources for enforcing real-time behaviors are allocated using various policy objects standardized by real-time middleware specifications such as RT-CORBA. These policies can be applied with different granularities, *i.e.*, affecting different scope of objects. Depending on the purpose for which a policy is designed, it can be applied (1) to the client-side, *i.e.*, to affect how the ORB invokes remote operations, or (2) to the server-side, *i.e.*, to affect how the ORB handles incoming operation invocations, or (3) to both the client and server sides, *i.e.*, to control common mechanisms and strategies.

Figure 4.4 shows the different scopes and granularities at which a policy can be applied, according to the `Policy` management framework introduced by the CORBA Messaging specification [36, 65]. On the client-side, a policy can be applied with the

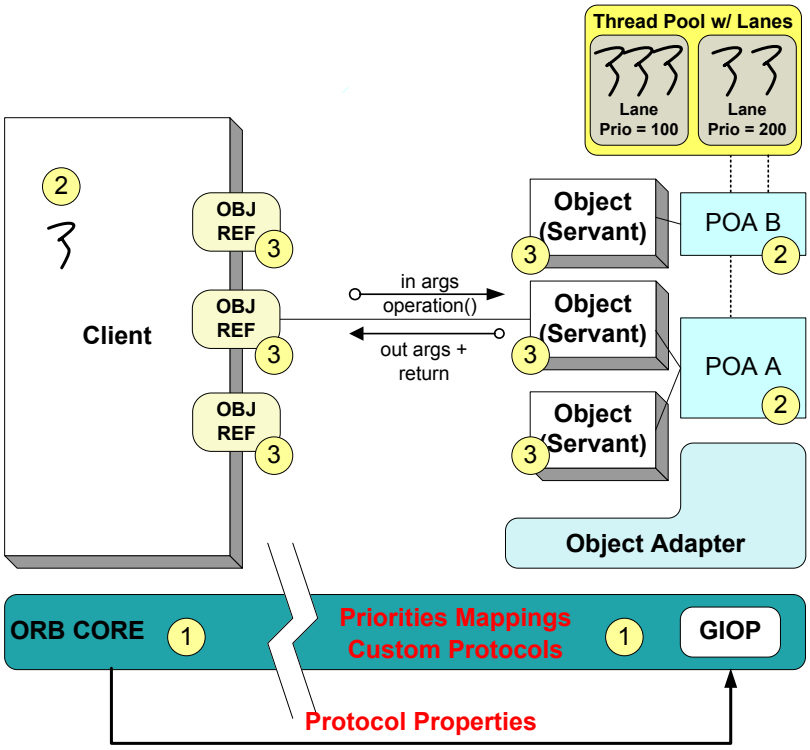

Figure 4.4: **Granularities for Applying Real-time CORBA Policies and Resources**

following granularities to affect how the client ORB invokes remote operations:

1. **ORB**: Policies applied to an ORB apply to all object references resolved by the ORB thereafter. These policies affect all operation invocations using these object references.

2. **Thread**: ORB-level policies can be overridden, within a thread of execution, by applying the policy to the `PolicyCurrent` pseudo-object associated with the thread. All subsequent operations invoked from that thread of execution will be affected by the applied policies.

3. **Object Reference**: An object can apply different policies from the ORB or the context of the thread of execution by overriding the policies in the scope of an object reference. Subsequent operations invoked on that object reference will be affected by the overriding policies.

Likewise, a policy can be applied with various granularities on the server-side ORB to affect how the ORB handle incoming operations:

1. **ORB**: Policies applied to an ORB affect all servants hosted by the ORB, *i.e.*, all incoming requests handled by the ORB will be affected by the applied policies.

2. **POA**: ORB-level policies on the server side can be overridden by applying the policy to the POA. All incoming requests to the servants managed by the POA which the overriding policy applied will be affected by the policy.

3. **Object reference**: Certain POA-level policies, such as priority level, can be overridden in an object reference by specifying the overriding policies when activating or creating a new object reference or servant. This overriding mechanism allows a servant to handle incoming requests using different sets of policies.

It is important to notice how policy overrides work in the CORBA policy management framework. When policies can be applied with different granularities, policies applied at finer levels of granularity override those applied with coarser granularity, as shown in Figure 4.5. This allows developers to configure an application with a set of default policies while being able to specialize parts of the application by overriding those default policies. It is therefore important for QoS-aware middleware to support such a policy override mechanism when integrating QoS provisioning support within the CCM.

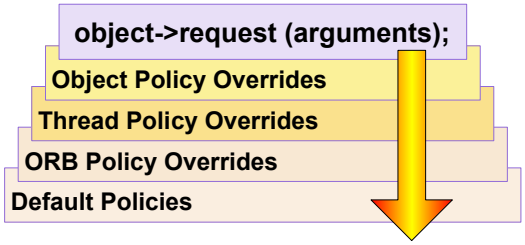

Figure 4.5: **Policy Override in CORBA's Policy Management**

To address this challenge, CCM metadata must be extended to specify real-time policies for different scopes that affect different entities in an application. Because this is related to the timing of inserting real-time policies, the table in Section 4.4.3 presents all the points for inserting real-time policies that CIAO supports.

4.4.2 Challenge 2 – Exploiting Composition Phases in CIAO

The CCM programming paradigm provides the foundation for composing systemic behaviors via specification of policies and mechanisms for a composed application. CIAO takes advantage of the multiple policy specification stages in the CCM development lifecycle to add hooks where systemic policies and mechanisms can also be specified, thus offering a significant advantage over the conventional RT-CORBA development process. To achieve this, several supporting constructs in CCM have been extended in to support composition of systemic policies and mechanisms. To support real-time applications in CIAO, it is first important to identify when and how real-time policies and supporting mechanisms can be composed into an application.

The following list provides an analysis of how different kinds of QoS provisioning policies can be composed at various stages of the CCM development lifecycle shown in Figure 4.6 and the consequences of using these composition strategies. The CCM development paradigm organizes the various concerns into the confines of different stages in the development lifecycle. Figure 4.6 gives a relatively complete portrayal of the interactions within and between each stage of the application deployment lifecycle. Historically, policies for managing systemic behaviors are done implicitly by developers in application programs. Identifying the kind of systemic policies that should be composed into each stage of the development lifecycle provides better organization to configure and manage these policies.

1. **Component implementation stage:** In the component implementation stage, component developers can specify the policies and mechanisms on which a component implementation depends to execute correctly, *i.e.,* to meet the QoS requirements of the component implementation. For example, a developer may decide to manage the priority level a component uses to invoke operations on a particular receptacle in the component implementation explicitly. In this scenario, there needs to be a way for the component implementation to specify its dependency on component servers and containers that support real-time behavior.

2. **Component packaging stage:** A component implementation package may also document its key systemic behaviors as constraints in this stage. For example, a component may document its implementation constraints on allowable rates it can achieve to process or to propagate an incoming operation invocation from a facet to operations to receptacles.

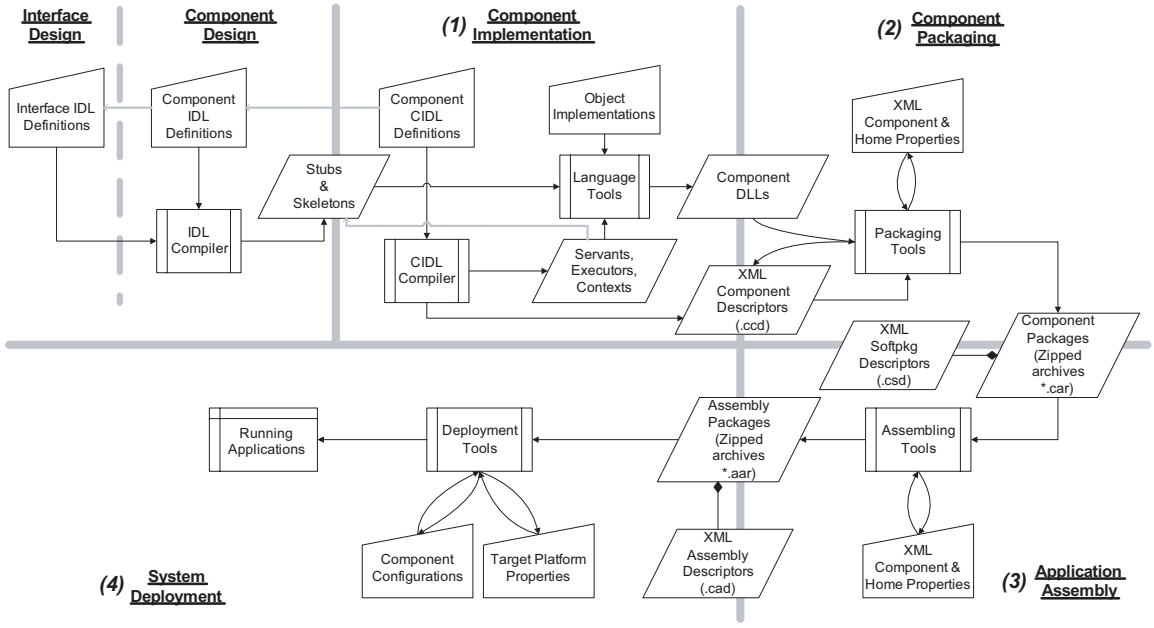

Figure 4.6: **CCM Development Lifecycle**

Binding systemic behaviors, such as RT-CORBA's priority model policy, into component implementations makes these behaviors part of component implementations. This approach allows application assemblers to use different component implementations for selecting different systemic behaviors. However, as we described earlier, this approach may hamper the reusability of component implementations as they assume certain kinds of support from the runtime environment and other components that coexist with them. Moreover, extra care must be taken when composing components with embedded systemic behaviors to ensure all the components used to assemble an application have compatible systemic behaviors [71, 79].

It is therefore important to extend component descriptors to allow developers to embed these implementation-specific dependencies and systemic behaviors. This extension will also provide hints to other tools to ensure that necessary supporting mechanisms are available in the composed application and that all the components have compatible behaviors.

3. **Application assembly stage:** During this stage, developers utilize various CASE design tools to create assembly specifications called *assembly descriptors* that describe how to build distributed applications using available component implementations. Information contained in assembly specifications includes the number of servers, what

component implementations to use, how and where to instantiate components, and how to connect component instances together in an application. Policies and mechanisms for allocating resources and controlling systemic behaviors can be applied at this stage to control and to allocate resources used by assembled applications. These policies and mechanisms can be applied and associated with different entities via assembly descriptors, to provide fine-grained control over systemic behaviors. These policies and resources can be specified in the following manner:

(a) Resources shared by multiple component instances should be specified in each server and process. Assembly descriptors can then specify the component instances or connections that use these resources by associating them with a resource. For example, thread-pools and priority-banded connections can be shared by multiple components and should be defined for the server.

(b) Policies and resources should be associated when specifying a component implementation. This approach allows specification of systemic behaviors of all component instances using the implementation. For example, a real-time priority model and priority level can be associated with the component implementation specifications.

(c) Likewise, policies and resources can be associated with specific components or home instances to weave in systemic behaviors. This approach is preferred over that of associating policies and resources with component implementations, since it provides more flexibility. Other than the priority model and priority level, component instances can be associated with shared resources, such as thread pools, allocated in the server.

(d) Policies can be applied with even finer granularity such as per provided interface, per receptacle, or per operation. For example, a component using the `CLIENT_PROPAGATED` priority model may want to invoke a certain operation of a receptacle with higher priority for prompt response from the object, while invoking other operations with lower priority for best-effort service.

(e) Resources and policies that affect the communication between two components should be associated with the connection specification in assembly descriptors. For example, a connection between two components may need to be pre-established using a private connection.

(f) Finally, certain QoS support mechanisms can be linked in dynamically at run time. The specification of this dependency should be associated with the server.

For example, protocol properties and custom priority mappings should be configured for each component server.

In addition, when component behaviors constraints are documented in component implementation packages, the application assembly stage allows application assembly tools to assimilate and reason about these constraints, make sure there are no conflicting constraints among component implementations, and deduce and synthesize a new set of constraints for the overall application. For example, if a series of component implementations is connected as a caller-callee chain via their facets and receptacles, with embedded allowable rate constraints, the assembly tools will be able to deduce a new set of allowable rates for the new assembled application and embed the constraints in the assembly descriptors.

4. **Application deployment stage:** At the final stage of transforming a component assembly into a fully specified and running application, component deployment tools are responsible to ensure the runtime environment, *e.g.*, the set of component servers, provides adequate support for the systemic behaviors the application demands. Support for systemic behaviors can either be provided by the deployment tools via a special component server implementation that offers the required mechanisms, or via dynamically linking the required mechanisms into component servers. By controlling the systemic aspect support mechanisms, the deployment stage provides the last chance in the CCM development lifecycle to control how resources are allocated prior to running the application.

   Similar to the component implementation, packaging and application assembly stages, tools can be used to model and assist the generation of deployment configurations to ensure the systemic requirements of applications can be met.

In summary, systemic aspects tend to cross-cut functional boundaries. Composing systemic behaviors often requires resources and mechanisms to be allocated and configured globally throughout an application. Therefore, component and assembly metadata must be expanded to parse, allocate and configure these resources and associate them with component instances or component connections. Moreover, to ensure a component server is equipped with the mechanisms needed to support the provisioned QoS requirements, component assembly metadata should include middleware modules that enable the control and configuration of these resources.

Table 4.2: **Stages for Specifying Real-time Policies and Resources**

Policy	Stage	Remarks
Requiring RT-ORB	1, 2	Requiring an RT-ORB is not a real CORBA policy but rather should be a requirement inferred from other real-time policies.
Priority Model	1	Embed the priority model a component implementation should always run in the implementation.
	2(b), 2(c)	Or more flexibly, specify the priority model when assembling an application.
	2(d)	Control partial priority model of a component for provided interfaces, receptacles, or operations.
Custom Priority Mapping	2(f)	Specify the custom mapping to use.
	3	Specify the custom priority mapping in this stage, allow reinterpretations of "priority".
Thread pools	2(a)	Allocate thread pools for later use.
	2(c)	Associate thread pools with instances of components.
Private connection	2(e)	Mark a connection as private.
Banded connection	2(a)	Define a banded connection policy that can be shared.
	2(c), 2(e)	Specify a component or a connection to use a banded connection.
Pre-connection	2(e)	Mark a connection for preconnection.

4.4.3 Solution – Identifying Points for Extending Metadata in CIAO

Based on our previous observations about stages and scopes for applying systemic policies, we summarize strategies that can be applied to compose real-time behaviors by assigning RT-CORBA policies into CCM-based DRE applications. Table 4.2 outlines the appropriate stages in the CCM development lifecycle where various real-time policies should be specified for building statically configured real-time applications.

Integrating these real-time policies into CCM requires extending various XML document formats flexibly to incorporate QoS information into component and assembly metadata. Chapter 5 gives a detailed explanation of how CIAO can utilize that knowledge to implement real-time composition support. These strategies are implemented both in CIAO's real-time metadata extensions and its deployment tools as is described in Section 5.5.

4.4.4 Composing Real-time Aspects with CORBA, CCM and CIAO

This section describes the key steps required to develop a simple client-server using CORBA, CCM, and CIAO mechanisms. Compared to using a traditional CORBA implementation to develop a simple client-server application, more steps are seemingly required to develop the same application using CCM, and CIAO adds even more steps for configuration of real-time aspects. However, this example will demonstrate that much of the complexity seen in CIAO is inherent to the other examples, and that in fact *accidental complexities* can arise in the other two examples that are implicitly addressed in CIAO.

Example Application

We now examine a simple but representative example application drawn from the motivating avionics mission computing domain [69]. Figure 4.7 illustrates a prototypical DRE application scenario involving three software entities:

1. A **Rate Generator**, which wraps a hardware timer that triggers pushing of events at specific periodic rates to event consumers that register for those events.

2. A **GPS**, which wraps one or more hardware devices for navigation. Because there is a delay in getting the location reading from the hardware directly, a cached location value is served via the exposed interface to provide immediate response. The cached location value is refreshed when the GPS software receives a triggering event and causes the controlling software to activate the GPS hardware for updated coordinates. A subsequent triggering event is then pushed to registered consumers to notify the availability of a refreshed location value.

3. a **Head-up Display**, which wraps the hardware for a display device in the cockpit to provide visual information to the pilot. This device displays a cached location value which gets updated by querying an interface when the controlling software receives a triggering event.

A DRE application like this can involve multiple controllers connected together via specialized networking devices, such as VME-bus backplanes.

This simple example represents a broader class of information flow systems to which our work on avionics mission computing systems belongs, as does the cockpit information system within each aircraft in the larger system-of-systems example shown in Figure 4.2 in Section 4.1. Furthermore, although details of the functional properties may

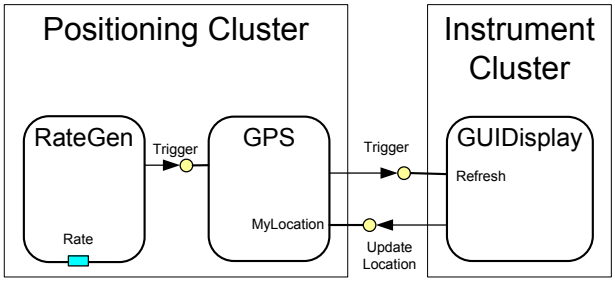

Figure 4.7: **A Prototypical CORBA DRE Application Scenario**

differ, many other DRE systems share the kinds of rate-activated computation and display/output QoS constraints illustrated here.

Development Using Conventional CORBA vs. CCM:

The first step in developing a CORBA-based system, using either conventional CORBA or CCM, always requires defining the interaction between software entities via interface definitions. For example, to implement the system shown in Figure 4.7, a developer must first define the interface for interactions such as sending the triggering message and querying the GPS for the current location reading. After the interaction interfaces are defined, implementing the example system using conventional CORBA middleware generally requires the following steps:

1. Develop servant implementations for previously defined interfaces. Often, these implementations are specific to system hardware.

2. Determine the location of each servant implementation in the network of controllers. Hardware layout often dictates the selection.

3. Based on the decision made in the previous steps, implement each server process. A server implementation needs to codify the following tasks:

   (a) Initialize and configure the ORB and hardware devices.

   (b) Initialize and configure POAs to suit the needs of different servant implementations.

   (c) Instantiate servants, register them with POAs, and activate them.

   (d) Initialize and configure an event delivery mechanism, if needed.

(e) Acquire necessary object references for the system. In effect, *connect* the referenced objects to this process.

(f) Facilitate synchronization with other services and server processes so they are initialized in the right order.

4. Deploy the assembled implementations to the target platforms.

As the list of steps indicates, in a conventional CORBA-based development paradigm, much of the overall system functionality is implemented in the server process, other than how to provide required control functions for the underlying hardware. This system functionality involves careful coordination among configuration and initialization code for specific hardware, ORB, POA, object connections, and initialization. This complexity is inherit to the CORBA development paradigm and requires careful programming of all processes involved.

The CCM development paradigm, however, uses a somewhat more elaborate development lifecycle which provides a better organization for managing all the different aspects in developing a system like this. Figure 4.8 presents a CCM-based design for the

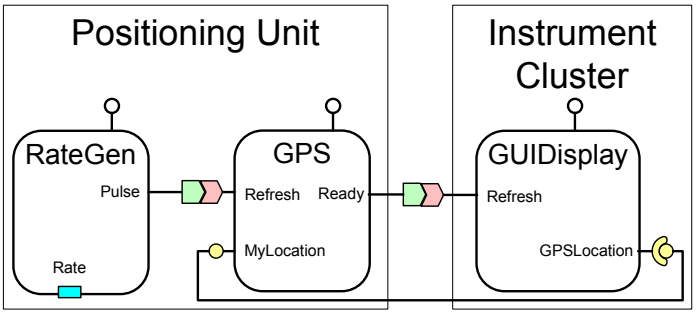

Figure 4.8: **A Prototypical CCM DRE Application Scenario**

same system shown previously. In this approach, each hardware device is wrapped under a *component* implementation. After the interfaces defining the interactions between hardware devices are defined, the CCM development lifecycle involves the following steps:

1. Identify a unit of installation as a component interface and design how the component interacts with the outside world by defining the ports and attributes of the component. It is straightforward to identify the component interfaces in this example as they map naturally to the hardware components.

2. For each type of component, developers will create one or more component implementations, *e.g.,* for different hardware or internal algorithms, and package them as component packages.

3. An implementation of the example system can then be composed by defining an application assembly file where developers

   (a) select the component implementations to use from a pool of available component implementations, which does not involve configuring any platform or runtime requirements, such as in the ORB or POA,

   (b) describe how to instantiate component instances using these component implementations, and

   (c) specify connections between component instances.

4. The application can then be deployed to the run-time platforms using a set of deployment tools.

Compared to the conventional CORBA development paradigm, the CCM development paradigm takes care of a lot of inherent complexity for the developers. The capability of the framework allows developers to concentrate on the problems at hand during each development stage without considering many details of the configuration platform, ORB, POA, and servant activation that are not directly related to the application. Moreover, CCM provides many flexible ways to configure an application. For example, the actual rate for the rate generator component can be specified as a default attribute value in the component package, or be overwritten in the application assembly, whereas conventional CORBA applications always require direct modifications to the application code.

Extending Applications Using Conventional CORBA vs. CCM:

When it comes time to modify or extend an existing application due to changes in requirements or hardware, it is even easier to notice the benefits of adopting CCM. Assume the previous example needs to be extended to include a collision warning system to notify the pilot of imminent danger. Figure 4.9 shows how a CCM-based implementation can be configured to support the new application. The added components are of the same component types as in the original example application but have different implementations to interact with different hardware devices, including a collision radar and the warning signal in the cockpit instrument cluster.

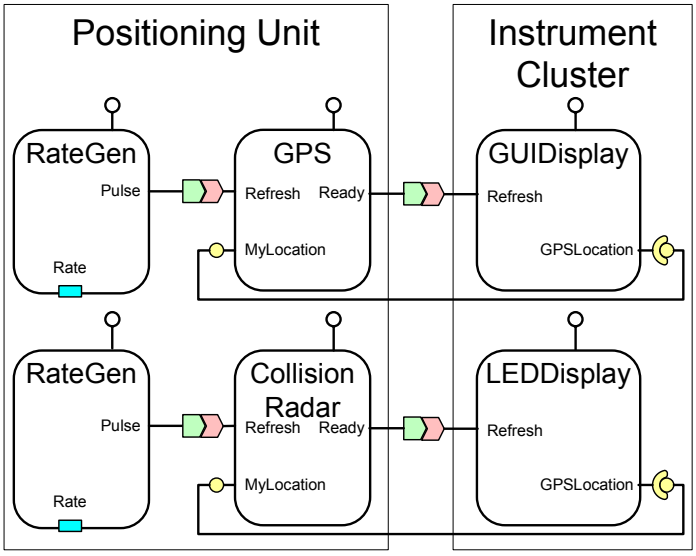

Figure 4.9: **Extended CCM DRE Application Scenario**

With the CCM development paradigm, extending the previous example application is as easy as providing the new component implementations for required component types, packaging them with component metadata, and using the new component implementation packages to compose the additional functionality into the new application via an assembly file. Extending a CORBA implementation on the other hand requires a lot more effort in modifying code throughout both subsystems in the extended application. The tasks required include creating the new servant implementations, configuring ORBs and POAs to accommodate these new interface implementations, activating these new interfaces, and modifying how the two subsystems interact and synchronize to prepare for new interconnections between the two programs.

Development Using Conventional RT-CORBA vs. CIAO**:**

DRE applications often require prioritization of various tasks to ensure critical tasks are handled within their time constraints. For example, as shown in Figure 4.10 in the extended example application, the collision warning system may run at a slow rate but it should always run at a higher priority than that of the GPS display since a pilot would perform an evasive maneuver to avoid a collision, instead of worrying about the exact location of the aircraft. ORBs conforming to the RT-CORBA specification allow developers to specify such real-time aspects by allocating computational resources and specifying real-time policies for different interfaces which instruct the endsystem to order the execution

of these requests according to different priorities. However, using conventional CORBA, adding the code for resource allocation and real-time policy specifications requires intrusive additions and modifications to the application code, including configuration of how to use the RT-ORB and RT-POA, and configuring the RT-POA policies for each POA, servant, or even servant activation.

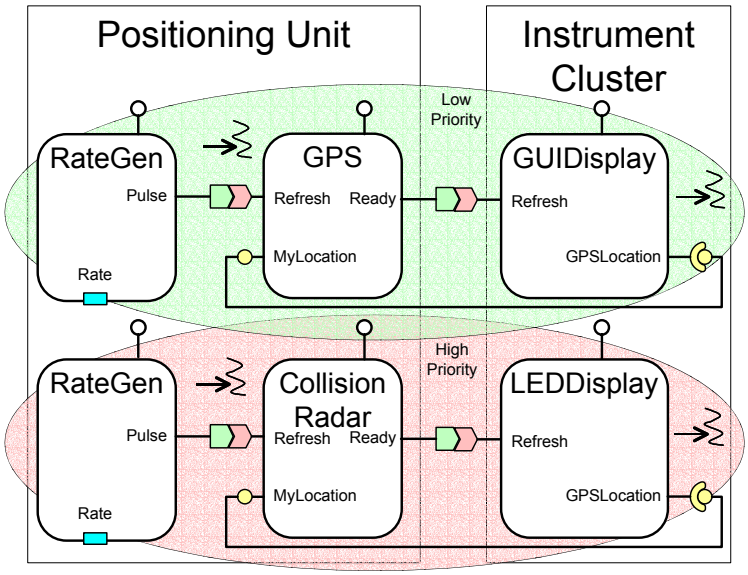

Figure 4.10: **Extended CIAO DRE Application Scenario**

In comparison, CIAO's real-time extensions provide a new real-time extension file format which we describe in more detailed in Section 5.5, to define all the real-time related resources and policies. These files can then be composed into an existing application assembly. The resources and policies defined in the file can then be specified to individual component instances. The resulting application assembly can then be deployed onto platforms that support the specified real-time requirements. The CIAO approach requires no changes whatsoever to the component implementations, nor any customized server modifications. Instead, this approach allows changing the real-time behavior of an application simply by composing real-time behaviors differently into another application assembly. Creating a system with different real-time behaviors is then as easy as deploying different application assemblies. Chapter ch:results presents empirical results that quantify the benefits and costs of prioritization of tasks using RT CORBA features directly, or through CIAO.

Chapter 5

CIAO Implementation

Previously, Chapter 4 highlighted the limitations of the existing CCM specification for supporting systemic aspects of DRE applications and the challenges in composing real-time policies into CCM applications. Moreover, it illustrated that current CCM specification offers insufficient support for configuring resource management mechanisms that are critical to ensure the correctness of DRE applications and presented solution strategies for how real-time policies can be specified and composed into CCM. This chapter presents the Component Integrated ACE ORB (CIAO), which is the prototype CCM implementation used in this research to apply the solution strategy for composing real-time policies into CCM that was outlined in Section 4.4. In addition to presenting the design and implementation of CIAO, this chapter also outlines challenges and solutions encountered while implementing CIAO.

This chapter is organized as follows. First, Section 5.1 provides an overview of the design of CIAO by reviewing its major building blocks. Sections 5.2 and Section 5.3 then describe the implementation of the core CIAO libraries and the tools that CIAO provides to help users implement and deploy component applications. Section 5.4 identifies the key design idioms that CIAO employs to address various implementation challenges. Finally, Section 5.5 provides a brief overview of Real-time CORBA specification and describes how CIAO extends CCM to support composition of real-time aspects into DRE applications and the challenges and solutions that arise in doing so.

5.1 Overview of CIAO

CIAO is a CCM implementation prototype with additional extensions to support QoS provisioning for DRE applications. As the major goal of this dissertation is to study the component middleware design and present an implementation to address the challenges and to demonstrate the effectiveness of the solutions for configuring real-time resources for DRE applications, CIAO's implementation has been carefully focused on a subset of CCM features that directly contribute to that goal. Features that are more relevant to enterprise applications than to DRE applications are not implemented in CIAO. CIAO consists of the following four major building blocks, as shown in Figure 5.1.

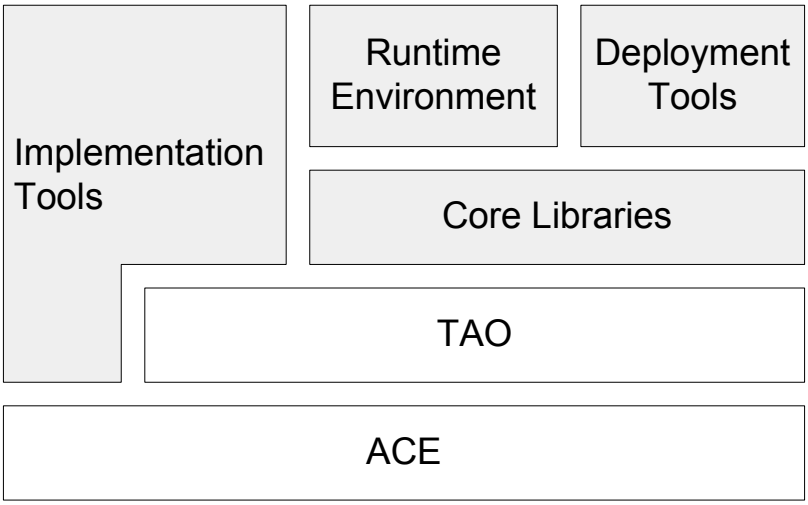

Figure 5.1: **Major CIAO Building Blocks**

1. **Core libraries:** The CIAO core libraries provide implementations for many CCM defined interfaces on which component users and implementations depend to build applications.

2. **Implementation framework:** CIAO provides prototypes and specifications of CIDL generated code for the component categories that are most relevant to DRE applications. It also identifies the canonical steps to implement a component.

3. **Runtime environment:** CIAO provides the runtime environment, *i.e.,* component servers and containers where component homes and components can be installed and executed. These tools form the basis of a real-time enabled run-time environment that supports CIAO's real-time extensions.

4. **Component deployment tools:** A set of tools in CIAO provides the capability to parse the component assembly files and software descriptors of component packages, and to deploy and configure applications which this research uses to conduct various experiments and provide example implementations.

These key building blocks together form the CIAO infrastructure that enables users to define component interfaces, generate helper implementation code for components, provide the runtime environment for component implementations and offer mechanisms to deploy and realize application assemblies. Based on these basic features, CIAO is then further extended to support real-time resource provisioning in component applications to address the challenges and to apply the solutions outlined in Section 4.4.

5.2 Core Libraries and Component Implementation in CIAO

The CCM specification defines a new set of interfaces to support the addition of new metatypes, such as `home` and `components`, component implementations, and deployment tools. Excluding the CCM defined IDL definitions, the core libraries consist of $\sim$4,000 lines of C++ and IDL code. Depending on their intended purposes, all the additional IDL interfaces defined by the CCM specification fall into one of three major categories:

- **Component interfaces:** These are stubs that component clients require to interact with the extended component interfaces. CORBA components and component homes inherit from a set of standard interfaces, such as `CCMObject` and `CCMHome`, which define the generic operations all components and component homes should support. There are also other utility interfaces and valuetypes, such as the `HomeFinder` interface and `ConfigValues` valuetypes, that help component clients locate, utilize, and configure components. The library containing stubs for component interfaces is called the *Client Library* in CIAO.

- **Component implementation interfaces:** These are the interfaces that component implementations and containers use to interact with each other. These interfaces include `CCMContext` and its derived interfaces that component executors use to interact with their containers, and `EnterpriseComponent` and its derived interfaces that all executors inherit from and are used by containers to call back to executors.

Moreover, the servant implementations of the aforementioned component interfaces also fall into this category as they are part of component implementations. Because these component implementation interfaces are used to interact with the container interfaces, this library is called the *Container Library*.

- **Deployment interfaces:** These are interfaces supported by various software entities, such as `ComponentServer` and `Container`, for deploying, configuring, and connecting components and applications. The deployment tools take advantage of these interfaces to deploy components across heterogeneous platforms. These interfaces are required to implement tools for deploying component implementations into component servers. The library containing these interfaces is therefore called the *Server Library*.

Separating these interface stubs and implementations into multiple libraries avoids the problem of CCM applications linking in unnecessary interface implementations, which not only prolongs the link time but also increases the footprint of component implementations and CCM applications. For example, for a pure client program that interacts with a CCM component, there's no need for the application to link with interfaces for component implementation and deployment. Likewise, when developing a component implementation library, there is no need to link in deployment interface implementations. Figure 5.2 shows the dependency relationships among the three core libraries upon which CIAO builds.

Figure 5.2: **Dependencies Among CIAO's Core Libraries**

Other than a set of core CIAO libraries that component clients, implementations, and other CIAO deployment and runtime services depend on, CIAO also offers a set of prototyping tools to assist in component development. A key tool CIAO provides for automating component implementations is the CIDL compiler. The CIAO CIDL compiler is being developed by the Distributed Object Computing (DOC) Group at the Vanderbilt University according to the prototypical code templates defined by the work presented in this dissertation. The code templates, which consist of approximately 2700 lines of C++ code, were designed in this research based on the Component Implementation Framework

(CIF) described in Section 3.2.4 which helps to automate the development of component implementations and includes two major parts:

- home and component servant glue code which provides the container hosting environment to home and component executors, and
- a component specific context which implements external connections and consumer subscriptions and manages interactions with the container framework.

As is shown in Figure 3.6 on page 29, the generated component servants contain and manage both component executors, which perform application functionality implemented by component developers, and component specific contexts, which manage component connections using code generated by the CIDL compiler. Figure 5.3 illustrates how a generated servant handles the server aspect of a component by forwarding operation invocations on component interfaces to component executors and connection management operations to the component-specific context. Figure 5.3 also illustrates how a component executor

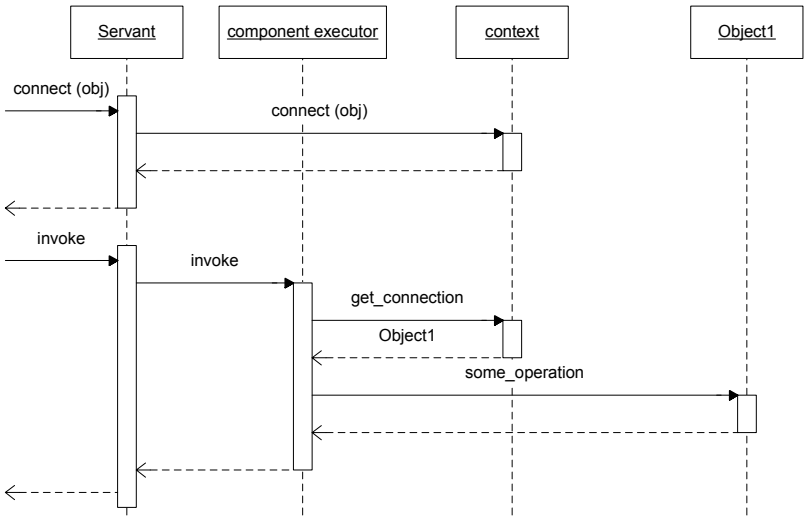

Figure 5.3: **Interactions Between Servant Glue Code, Executors, and Component Specific Context**

queries the component-specific context to acquire the object reference connected to a receptacle when the component needs to invoke operations and behave as a client. Together, the generated servants and component-specific contexts can provide hooks to intercept the operation flows in and out of components.

5.3 CIAO Run-time and Deployment Tools

Section 5.2 outlined CIAO's core libraries and its support for implementing CCM components. As Chapter 3 points out, the eventual goal of creating software components is to be able to assemble and deploy applications declaratively. CIAO focuses on providing prototypical mechanisms and runtime support to realize component application assembly descriptors. Tools for editing and generating the assembly descriptors, however, do not fall into the scope of this dissertation. This section illustrates key CIAO tools for deploying application assemblies as was outlined in Section 3.2.6. Moreover, they serve as the prototype implementations of the real-time run-time and deployment tools described later in Section 5.5.

CIAO provides the following tools for deploying and realizing application assemblies:

- **CIAO_Daemon**: As its name implies, the CIAO_Daemon is designed to be a constantly running process. Each machine that allows components to be installed must have a CIAO_Daemon process running to interact with other deployment tools. A CIAO_Daemon has two major responsibilities. First, it creates properly configured component server processes upon receiving requests from deployment management tools. Secondly, a CIAO_Daemon manages the component implementation libraries by providing look up service for the component servers it creates.

  Ideally, a CIAO_Daemon will manage the component implementation libraries on the host where it is running by interacting with a software distribution framework and downloading the necessary component libraries at appropriate times. CIAO, however, currently does not support automatic distribution of component implementations, so component implementation libraries must be distributed manually. A list of the available component implementations therefore must also be maintained manually. The CIAO_Daemon's component implementation library management role thus is currently limited to reading the list of installed component implementations and providing a look-up service for components specified by the assembly manager.

- **ComponentServer**: CIAO's component server process implements the `ComponentServer` interface that the CIAO_Daemon uses to spawn generic component server processes. A component server creates and configures containers and allocates and manages resources within the server process according to the needs of the components. Containers created by the component server support the

`Container` interface with which the deployment framework interacts to install component implementations within or remove them from the containers.

- **Assembly_Manager**: The Assembly_Manager tool in CIAO provides a deployment service that creates and manages instances of application assemblies. In CIAO, the Assembly_Manager is the central point of control over a deployment environment, and manages all the relevant information for building an application from an assembly descriptor. During the process of deploying an application, the Assembly_Manager parses the assembly descriptors that define the application. It then uses the collected information to interact with the CIAO_Daemon process to create an application as a hierarchy of servers, containers, component home installations, and component instances.

- **Assembly_Deployer**: The Assembly_Deployer is the command line front end tool used to interact with the Assembly_Manager. Application deployers use the Assembly_Deployer to instruct the Assembly_Manager to create an instance of an assembly descriptor, or to tear down an existing application instance.

Including the real-time extension support tools described later in Section 5.5 along with other development helper tools, the deployment and run-time prototype tools provided by CIAO consist of a total of 11,146 lines of C++ and CORBA IDL code.

Deploying an application requires all the deployment tools to operate in concert. The following list enumerates the interactions that occur when a system deployer uses the Assembly_Deployer to deploy an example two-component application. Figure 5.4 illustrates similar interactions among deployment tools.

0. Before a system deployer can use CIAO's Assembly_Deployer to realize an application assembly, the deployment environment must be set up and ready to accept application deployment. In CIAO, this means that

   (a) Because CIAO currently lacks the ability to distribute component implementations automatically over the network, proper component implementations must be compiled and copied to target deployment nodes, which usually consist of computers connected to a network.

   (b) For each deployment node, CIAO_Daemon must be running. Moreover, the location of all the component implementations must be passed to the

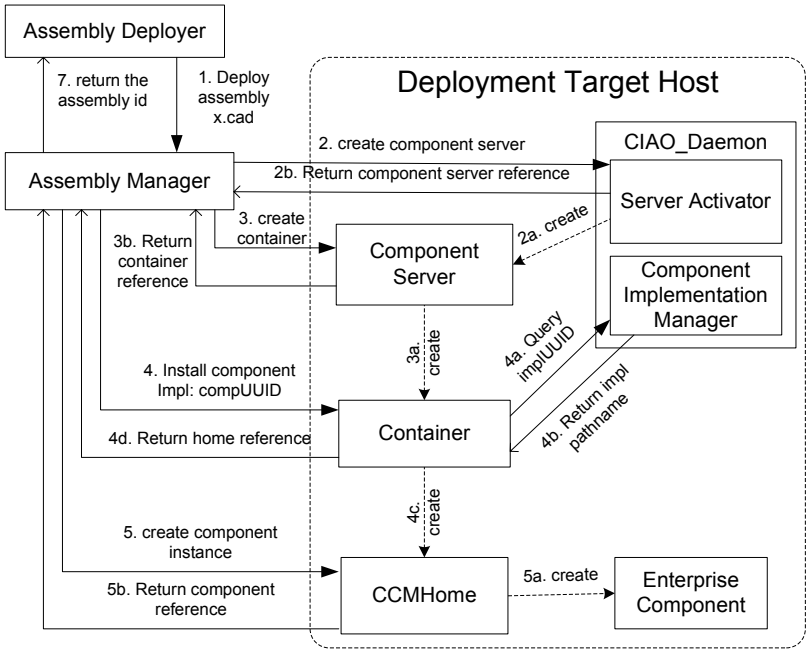

Figure 5.4: **Interactions Between** CIAO **Deployment Tools**

CIAO_Daemon which currently acquires the information from a component installation file. Each component implementation has a Universal Unique IDentifier (UUID) which is an universally unique 128 bit number. This file contains a table that maps a component's UUID to a local file name which contains the component implementation built for the deployment node's particular OS and languages.

(c) The Assembly_Manager server process should also be started with the information about where all the CIAO_Daemon processes belonging to the deployment environment can be contacted. Similar to the case of automated distribution of component implementations, CIAO does not currently provide mechanisms to automatically distribute assembly and component package descriptor files. These descriptor files should be made available for Assembly_Manager server by copying them manually to a known location.

1. A system deployer uses the Assembly_Deployer program to tell the running Assembly_Manager process which application assembly should be deployed. The Assembly_Manager then goes on to read the application assembly descriptor and, in turn, build the application according to the assembly descriptor by iterating through

the following steps. The status of the application assembly deployment is maintained as an object which is accessible through a CCM deployment interface called `Assembly`.

2. Whenever the Assembly_Manager decides that a new component server needs to be instantiated, it contacts the CIAO_Daemon process running on the target host. The Assembly_Deployer then requests the daemon process to create a new component server with a list of required features the new server should support, and the configuration options the new server should use. After the server has been successfully created, the daemon process eventually returns the object reference of the component server interface of the newly created component server back to the Assembly_Manager for later use.

3. After a component server is up and running, the Assembly_Manager then contacts the server and requests that the server create and instantiate the type of container the Assembly_Manager deems necessary to satisfy the needs of component implementations it will install within that container. Similar to the case of creating a component server, the Assembly_Manager can optionally pass the server a list of required properties and configuration options for the new container. The server returns an object reference to the newly created container back to the Assembly_Manager for later use. Notice that an Assembly_Manager may create several containers of different types and properties within a component server to satisfy component implementations with different needs.

4. Once an appropriate container is created, the Assembly_Manager can use the container reference and request it to install and activate a home for a component implementation. The Assembly_Manager specifies the component implementation and other runtime properties for the component installation. The component implementation is specified by a UUID that uniquely identifies that implementation. The container then queries the CIAO_Daemon for the location of the DLL/shared object that contains the component implementation. Once the component has been installed in the container and the object reference for the home interface is activated in the container, the home reference is returned to the Assembly_Deployer for later use. The Assembly_Manager can install more than one component home in a container.

5. If component instances need to be created for a home, as specified in the application assembly descriptor, the Assembly_Manager will then use the home interface to create the component instances and store the component references for later use.

6. The process of creating a component server and its containers, component homes and component instances may each iterate several times within the same level before proceeding to the next step. The Assembly_Manager may iterate through Step 1 through 5 before all statically configured home and component instances are created. After all statically created instances are created, the Assembly_Manager will establish the connections among these instances based on the specification in the application assembly descriptor.

7. Eventually, after all homes and components are installed and the application is running, the Assembly_Manager returns an assembly identifier back to the Assembly_Deployer. This identifier is used to identify the internal data structure that keeps track of this application deployment. A system deployer also needs to identify when the application assembly needs to be destroyed.

To clarify these nested instantiation and containment relationships when deploying an application, the previously mentioned procedures in the application deployment process can be modeled by the following regular expression. If Step 1 through Step 7 each generates a digit corresponding to the step number each time it creates an object or connection, a successfully deployment of an application assembly should produce a sentence that belongs to the following regular expression:

$$1 \cdot (2 \cdot (3 \cdot (4 \cdot 5*)+)+) + 6 * \cdot 7$$

Figure 5.5 demonstrates the containment relationships within an instantiated CCM application.

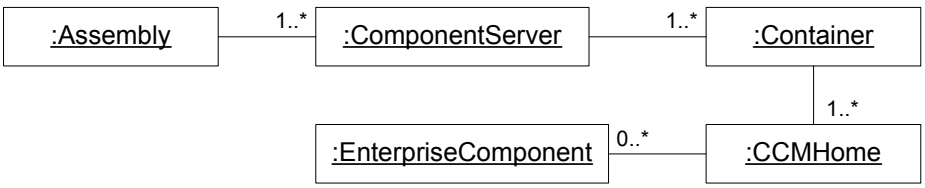

Figure 5.5: **Interactions Between Various** CIAO **Deployment Tools**

5.4 Significant CIAO Design Features

During the course of designing and implementing CIAO as the basic prototype framework for integrating the solutions for challenges in composing real-time policies in CCM, several new challenges were identified. New design features, which turned out to be essential to the component middleware framework, were devised to address these challenges. This section identifies these challenges and lists the design features CIAO employs and the benefits these features bring.

5.4.1 Separation of a Component Implementation Into Multiple Libraries

Context: As was described in Section 5.2, other than component executors implemented by component developers, a component implementation also depends on other software entities including interface stubs and skeletons generated by the IDL compiler, and the servant glue code and component-specific context generated by the CIDL compiler. Other than stating that component executors should be loaded dynamically into component servers, the CCM specification does not dictate how component libraries should be implemented. The CCM specification does not dictate how generated code and developer implemented component code should be linked into dynamic libraries. The packaging and deployment section of the specification assumes that each dynamic library contains the full implementation of a component, including all the stubs and skeletons of its interfaces, CIDL generated code and executors.

Problem: However, aggregating a component implementation in a single dynamic loadable library can raise the following concerns:

- CCM components often interact with other components. Because there's no way to foresee whether a component implementation will be collocated in the same process with another collaborating component, it is important to ensure the effectiveness of collocation optimization for CCM components regardless how they are assembled. When the code for both the stub and the skeleton is linked together dynamically, the ORB knows a reference to the interface could be collocated from the existence of the skeleton code. However, if both the stub and skeleton are linked into the component library serving the interface, the library of another component using the interface will contain a separate copy of the stub code for the interface. There will be no

way for this client component to figure out whether if it is collocated with the server component, and thus will defeat the collocation optimization of the underlying ORB.

- As was previously identified in Section 5.2, the servant glue code implements many systemic strategies, such as event delivery, for executors and provides interception points where calls in and out of components can be intercepted. Moreover, the CIDL compiler can be extended to generate servant glue code that supports different systemic support strategies for both servants and component-specific contexts. For example, a generated servant can intercept incoming operation invocations and execute necessary systemic aspect code, such as updating the urgency of the call, before relaying invocations to corresponding actual executor functions.

  Similarly, the CIDL compiler can be extended to generate component-specific contexts that, when queried by executors, will return references to receptacle and event consumer smart proxies which perform extra operations before relaying invocations to target receptacles or event consumer methods. Different combinations of generated servants and executors can create components with different implementation strategies or different systemic behaviors. Forcing generated servant and executor code into a single library not only diminishes the reusability of both the servant and executor but also causes increasing demands on memory, from code duplication when components have different implementations or different systemic behaviors.

Solution: To address the aforementioned problems, instead of linking all the entities into a single implementation library, the prototype component implementation tools provided by CIAO include scripts to generate `Makefile`s for creating 3 separate dynamic libraries for each component implementation. As shown in Figure 5.6, these libraries include:

- **Stub library** which contains collections of stub implementations a component client needs to invoke operations on the component.
- **Glue code library** that contains implementations for the servant glue code, the component-specific context, and the executor base interface.
- **Executor library** contains the executor implementations that perform the application functionality.

Specifically, separating a component implementation into multiple dynamically loadable libraries address the problems from a single component library as follows:

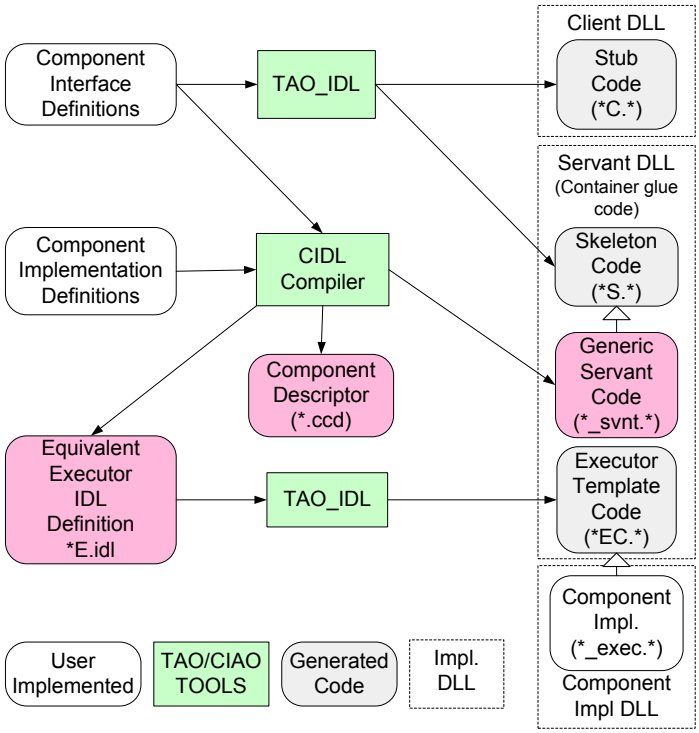

Figure 5.6: **Implementing** CIAO **Component Libraries**

- Separating the interface stub code into its own library is necessary for CIAO components to take advantage of the collocation optimization [85] offered by TAO. By separating the stub code into a separate library, the ORB endsystem can then determine whether an object reference is collocated or not during run-time. It is therefore important to separate stub code of various interfaces a component supports into individual libraries to make sure components can still take advantage of the collocation optimization.

- Separating the CIDL generated servant glue code into its own library provides mechanisms to flexibly compose executors with servants implementing different systemic strategies into component implementations. Replacing servant libraries independently allows CIAO to combine component implementations with servants of different strategies and thus compose the systemic behaviors into a component system. The CIDL compiler can also generate servants and component-specific contexts that provide hooks for composing systemic behaviors dynamically as shown in Figure 5.7. Separating servant libraries thus enhances flexibility to reuse the component implementation. Components can be composed with different systemic behaviors by

linking to different servant libraries. Likewise, multiple component implementations can share a common servant library if they all have similar requirements for systemic behaviors.

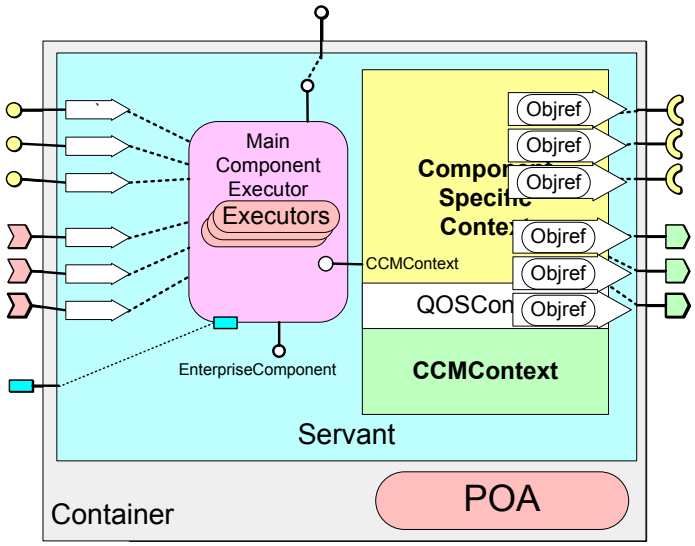

Figure 5.7: **CIDL Hooks for Integrating systemic Aspects**

The advantages of separating component libraries, however, come at a cost of additional complexity in extending and creating a component package and its descriptors and managing component libraries. However, all the procedures to generate component implementations and descriptors, and handle component dependencies and distribution are expected ultimately to be part of an automated process handled by higher-level tools. Component developers are therefore likely to be shielded from this added complexity.

5.4.2 Separation of Logical and Physical Configurations

Context: When defining an application assembly, there often is a need to assume a certain deployment configuration and specify how the application assembly can be deployed. For example, application assemblers can use the “destination” element in CCM’s assembly descriptors to specify the *location* where a component should be installed. Usually, a destination should refer to a service running on the destination machine and can activate component servers when being requested.

The CCM packaging and deployment specification, however, does not define many details on how to interpret these descriptor elements, including "destinations". A straightforward approach is to embedded the actual information pointing to the "destination" directly in the assembly. For example, a destination element can simply contain either the host name and port number pair or the stringified IOR of the server activation service, to identify the target endpoint on which a component can be installed.

A similar situation happens when specifying the run-time configurations for each deployment node. TAO provides a rich set of configuration options, via command line flags or a configuration file, for controlling various strategies and mechanisms of an ORB. Many mechanisms that can be configured into the ORB through these options are platform dependent, *e.g.*, the Unix-domain socket Inter-ORB Protocol (UIOP) is not available on Win32 platforms. Moreover, many ORB configuration options interact and must be configured compatibly end-to-end, throughout the application.

Problem: However, interpreting the destination elements as a reference directly to the destination machine or object combines the knowledge of a specific deployment environment directly into the application assembly. By committing to specific deployment targets prematurely during the application assembly stage, the application assembly descriptors are therefore bound to the specific deployment configuration and can not be reused in another deployment environment. Likewise, as each deployment node may require certain platform-dependent mechanisms to support certain configurations, it is not appropriate to embed the actual configuration information in the application assemblies. Embedding the platform-specific configuration options into assembly descriptors also prevents these descriptors from being deployed onto different target deployment platforms.

Solution: What is really needed is a way to defer the determination of the actual deployment location and its configuration until deployment time, when more complete information is available. To achieve this goal, CIAO employs a key design feature to separate logical and physical configurations. This separation of configuration decisions allows developers making design decisions at earlier stages to express only the design intentions as "logical names" and leave the decision on how to achieve these logical decisions to developers at a later stage.

Specifically, CIAO's deployment framework adopts this design feature to decouple the specification of the actual deployment location from the description of the deployment topology in an application assembly. Application assemblers can now specify deployment

destinations using logical names that represent some abstraction of these destinations that are meaningful to the assembly contexts, *e.g.*, names like "navigation computer" and "instrument panel display processor". The system deployers then map the application destinations into nodes within a deployment topology by translating the destination logical names into actual host names in the Assembly_Manager service. This separation of logical and physical deployment destinations alleviates the need to couple knowledge about the actual deployment configuration into application assemblies, and allows system deployers to redeploy application assemblies easily by reconfiguring the deployment destination maps in the Assembly_Manager.

To address the limitations of specifying ORB configuration options in application assemblies directly, CIAO applies the same design principle to separate the logical and physical specifications of ORB configurations. Instead of allowing application assemblers to specify a list of ORB configuration options directly for each installation destination, CIAO extends the semantics of the "destination" element so that a logical *configuration* name can be appended optionally to the logical destination name. Therefore, a logical destination can now be interpreted as a logical location plus a logical name for the required ORB configuration, *e.g.*, "RSVP-enabled" or "multi-endpoints". Unlike the installation destinations whose physical locations are interpreted by an Assembly_Manager that coordinates actual deployments, the logical configuration names are passed by the Assembly_Manager as-is directly to the CIAO daemon process where these logical configuration names can be translated into platform-specific ORB configuration options. Allowing CIAO daemon processes to interpret the meaning of logical configuration names frees the Assembly_Manager from the need to acquire detailed and platform-specific knowledge of all the possible deployment targets.

Figure 5.8 illustrates how and where CIAO's deployment framework determines the physical locations and configuration options it uses to start up a component server. As the figure demonstrates, a system deployer can easily redeploy an application assembly onto a set of different machines by reconfiguring the deployment framework without the need to modify the application assembly descriptor file. By deferring the selection of the deployment target location and the ORB configuration options, CIAO provides a flexible deployment framework for software modeling tools to model and synthesize not only application compositions but also other systemic aspects [19], such as deployment topologies and component server configurations. CIAO currently supports configuration of component servers via the ACE service configurator framework [25], but that same mechanism

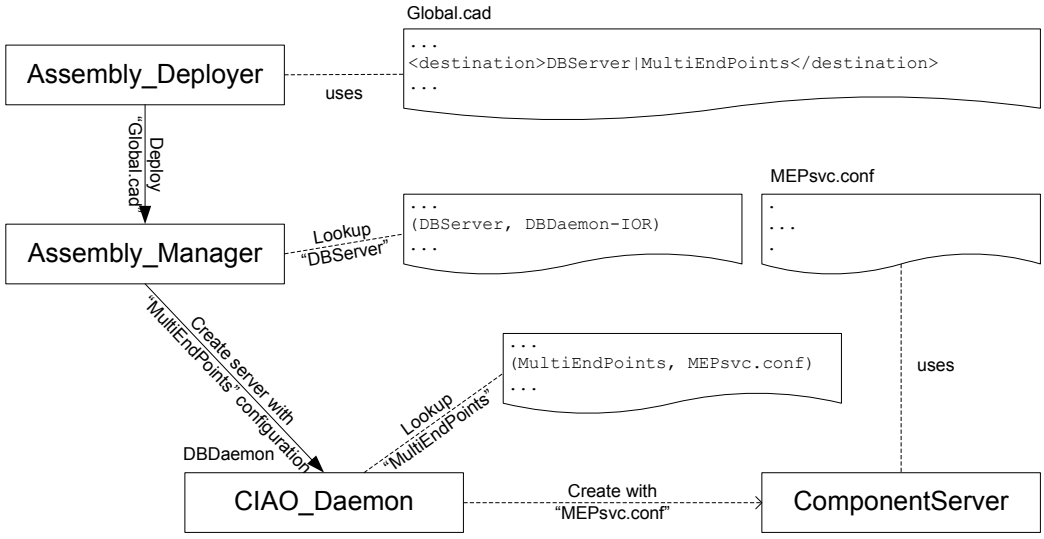

Figure 5.8: **Steps CIAO Takes to Determine the Location and Configuration of a Component Server**

can be extended to specify command line flags and other configuration property files for the component servers.

5.4.3 ACEXML and XML-based Service Configuration

Context: The eXtensible Markup Language (XML) provides an extensible standard syntax for describing information as a layered structure of mark-ups. XML documents are easy to parse because of their relatively concise grammar. At the same time, it is relatively easy for people to interpret XML documents because the information presented is well-structured and text-based. XML has gained great popularity in a wide variety of application domains where new document types are defined for describing the domain-specific knowledge. Many tools and extension features are also available for extracting, cross-referencing, and synthesizing information from XML documents.

The CCM specification also adopts XML as its language to describe many forms of component and application information. A CCM framework can generate and consume XML documents to pass component and application-related information between tools. For example, CCM tools can utilize XML documents in following fashion.

- The CIDL compiler generates XML component descriptors to document component capabilities, such as available interfaces and port mechanisms, and external dependencies of component implementations, such as the types of container services they require.

- The assembly tools import XML component feature descriptors to use these components in application assemblies. The knowledge from component descriptors allows assembly tools to ensure compatibility of components within assemblies. The assembly tools also generate XML assembly descriptors to document compositions of applications.

- The packaging tools generate XML packaging descriptors to describe the content of a software package that aggregates various software entities including binary files of component implementations, meta-data about these implementations, and other necessary libraries.

- The deployment tools parse the XML packaging descriptors, and other nested XML descriptors to deploy individual components or application assemblies.

Some of the information described in these XML descriptors is already available in the form of other languages. For example, descriptions about component capabilities are available in IDL files and descriptions about component implementations are available in CIDL files. Using XML descriptors is still a preferable choice as XML descriptors can be designed more concisely to contain only relevant information extracted from IDL and CIDL. XML also allows these descriptors to be extended to insert information not included in the original IDL or CIDL files.

Moreover, because XML's design allows the separation of parsing of XML documents and extraction of information from the reasoning and processing of the information, it is much easier for various tools to support multiple document formats. Therefore, instead of having to include multiple parser implementations for parsing and processing documents formatted in various grammars, a tool can use a common XML parser to process and extract information from all types of documents but implement only how to handle the extracted information. This approach not only reduces footprint but more importantly, makes it easy for these tools to add support for new document types.

Problem: Nevertheless, most of the XML generation and parsing in CCM is done offline, *i.e.*, processing XML documents is generally not required by the CCM specification

in runtime environments such as a daemon process or a component server, that are needed to run a CCM application. Existing XML parser implementations are designed to be fully compliant with the XML specification and often come with features for performing character code transformation along with additional capabilities such as for document transformation. These features and capabilities are essential for enterprise applications, such e-commerce and dynamic web page rendering, but increase the memory footprint significantly for features not used for our purpose of simply extracting information from XML files in CIAO's runtime environment.

Solution: An XML parser called ACEXML is therefore implemented to serve this purpose in a more appropriate manner for DRE applications. ACEXML is a small footprint, highly portable, non-validating basic XML parser implementation. It adopts the event-driven styled of the Simple API for XML (SAX) version 2 developed by David Megginson [58]. The Service Configurator framework [62] in ACE has been re-implemented to use XML-based configuration files to allow easy extension for providing more versatile configuration features in the future. The following section also illustrates how CIAO's real-time extensions utilize the ACEXML parser to support composition of real-time features into DRE applications.

5.5 CIAO's Real-time Extensions

Context: Although RT-CORBA standardizes the interfaces for controlling resources critical to support real-time QoS behaviors of DRE applications as was described in Section 4.2, RT-CORBA lacks sufficient higher level abstractions to separate real-time QoS policy configurations from application functionality. QoS policy configuration code is often intertwined with the application code which makes complex DRE applications hard to configure, validate, modify, and evolve consistently. The tight coupling of policy aspects with application logic is inherent to the CORBA 2.x object model, and has been carried forward into derivative specifications that focus on QoS properties, such as the Real-Time CORBA 1.0 and 2.0 specifications.

As was outlined in Chapter 4, it is necessary to make real-time properties an integral part of the CCM framework in order to apply the CCM programming paradigm in the DRE application domain. Section 4.4 further detailed the challenges and solutions to compose real-time policies into different scopes in a DRE application at various stages of the CCM development lifecycle.

Problem: However, unlike for more enterprise-oriented QoS support such as transactional behaviors, the existing CCM specification does not provide a notion of abstracting real-time behaviors as real-time policies in various XML descriptor formats. Moreover, scattering all the real-time policies across various XML descriptors makes it hard to analyze and reason about the composed real-time behaviors or to ensure their consistency. Because the work of this dissertation concentrates on composing the real-time behaviors into applications, the real-time extensions added by the work described in this dissertation focus on the final two stages of the development lifecycle, namely the application assembly stage and the application deployment stage.

Solution: To resolve the problem of applying the solution outlined in Section 4.4.3 to compose real-time behaviors during the application assembly stage, CIAO defines a new XML document type for describing the real-time behaviors and the resources to support those behaviors that a component server must support. CIAO calls this new XML document type Real-Time Component Assembly Descriptor (RTCAD). As Section 4.4 points out, during the application assembly stage policy specifications can be used to (1) allocate resources shared by multiple components, (2) associate shared resources, and (3) apply real-time policies within different scopes. A CIAO RTCAD file organizes definitions under two XML elements:

- `rtresources` which contains a list of named resource declarations that are to be allocated by a component server. Users can specify real-time CORBA related resources, including thread pools, thread pools with lanes, and connection bands. Resources defined in this section can be associated with a policy by referring to their names as is described in the next section.

- `rtpolicyset` which defines a named collection of real-time CORBA policies that should be applied together. The real-time CORBA policies that are supported in CIAO include priority model, thread pool, and banded connection policies. Both thread pool and banded connection policies need to refer to the corresponding shared resources defined previously in the same RTCAD file. A global resource defined in a previous `rtresources` element can be associated with more than one `rtpolicyset` to allow a resource to be shared by multiple components.

RTCAD files have the file extension name `rtd`. The following is a snippet from an example RTCAD extension file named `firstclass.rtd`.

```
<rtcad_ext>
  <rtresources>
    <threadpoolwithlanes
                id="shared_pool"
                stacksize="0"
                borrowing="no"
                buffering="no"
                max_buffer="0"
                buffer_size="0">
      <lane priority="1"
            static_threads="100"
            dynamic_threads="300"/>
      <lane priority="2"
            static_threads="2"
            dynamic_threads="2"/>
      <lane priority="3"
            static_threads="1"
            dynamic_threads="2"/>
    </threadpoolwithlanes>
  </rtresources>

  <rtpolicyset id="HIGH_PRIO_POLICY">
    <priority_model_policy
         type="server_declared"
         priority="3"/>
    <threadpool_policy
         idref="shared_pool"/>
  </rtpolicyset>

  <rtpolicyset id="LOW_PRIO_POLICY">
    <priority_model_policy
         type="client_propagated"
         priority="1"/>
    <threadpool_policy
         idref="shared_pool"/>
  </rtpolicyset>
```

In this example RTCAD file, a thread-pool-with-lanes is defined in the `rtsources` section. Two sets of policies are defined with different priority levels. However, both policy sets share the same common thread pool resource.

To compose the real-time behaviors defined in RTCAD extension descriptors, CIAO uses the `extension` element that the CCM specification alloted for assigning vendor-specific extensions in assembly descriptors. An application assembler can specify the RTCAD extension file a component server should use by defining the `extension` subelement as:

```
<extension
  class="RT-CAD-EXT"
  origin="CIAO">
    firstclass.rtd
</extension>
```

Here, the `class` and the `origin` attribute values identify the content of the extension to be a CIAO RTCAD extension file name named `firstclass.rtd`.

Once a component server uses the extension element to define the kind of RTCAD extension file it supports, a component hosted by the component server can then specify the real-time policy set *defined in the RTCAD extension file* this component or component home instance must support. The association is, again, defined via the `extension` flag under the component or home instance as:

```
<extension
   class="RT-POLICY-SET"
   origin="CIAO">
     A_POLICY_SET
</extension>
```

As in the case of specifying the RTCAD extension file a component server needs, the `class` and `origin` attribute values identify the name of the policy set that has been previously defined in the RTCAD extension file. The name of the specified policy set also refers to the real-time behaviors a component installation must support.

Besides extending the component assembly descriptors and the deployment tools to handle XML descriptors, both the component server and the container need to be extended to process and configure the real-time policies that are composed into an application assembly. CIAO provides implementations for a real-time component server and a real-time container. When deploying an application assembly file with real-time behaviors composed within it, the deployment tool tells the component server the RTCAD extension file to use when creating the component server. The real-time component server will open and parse

the specified RTCAD extension file and allocate the resources and maintain the lists of available resources and policy sets available in the component server.

CIAO's deployment tool also tells the the component server the name of the real-time policy set to use to create a new container instance. The component server then searches for the list of policy sets by their names and applies the policies defined in the set. The mechanism to apply the RTCAD extension follows the same design principle of separating logical and physical configurations. As during the application assembly stage, only the logical policy set names which represent logical concepts of real-time behaviors that should be applied to component installation, and logical RTCAD extension file names which identify collections of real-time policies and resources, are used to realize the policies. The actual policies applied to a component instance will not be determined in the application assembly but will be deferred until the component server parses the actual associating RTCAD extension file.

The following snippet of an example application assembly descriptor shows how the two policy sets can be composed into an application.

```
<processcollocation>
   <homeplacement id="a25_WorkerHome">
     <componentfileref idref="com-Worker"/>
     <componentinstantiation id="a_W25"/>
     <extension class="RT-POLICY-SET"
                origin="CIAO">LOW_PRIO_POLICY</extension>
   </homeplacement>

   <homeplacement id="a75_WorkerHome">
     <componentfileref idref="com-Worker"/>
     <componentinstantiation id="a_W75"/>
     <extension class="RT-POLICY-SET"
                origin="CIAO">HIGH_PRIO_POLICY</extension>
   </homeplacement>
   <extension
      class="RT-CAD-EXT"
      origin="CIAO">firstclass.rtd</extension>
</processcollocation>
```

As the example shows, two components are installed and instantiated in the same component server which needs to allocate the real-time resources, in this case the thread pool with lanes, defined in the RTCAD file "firstclass.rtd". The component server will then create two

real-time-enabled containers, each configured to support the `HIGH_PRIO_POLICY` policy set and the LOW_PRIO_POLICY policy set, respectively.

The service configuration framework provides the mechanism to determine the priority mappings. System deployers can, therefore, configure the actual real-time behaviors by providing customized RTCAD and service configurator files for a specific deployment environment.

Chapter 6

Empirical Studies

This chapter presents an empirical analysis of the performance cost and benefits of applications built using CIAO vs. TAO. The two main goals of these experiments are (1) to compare and document the overhead imposed by CIAO, *i.e.*, in terms of throughput, latency, jitter, and footprint of the same test application based on TAO vs. CIAO, and (2) to validate the efficacy of composing systemic aspects in CIAO, *i.e.*, prioritization and rate tuning, for real-time application performance. The results and analysis shown in this chapter will illustrate how CIAO effectively address the limitations of conventional CCM outlined in Table 4.1 on page 40.

To our knowledge, CIAO is the first component model implementation to integrate configurability of RT-CORBA features. Since TAO is a state-of-the-art real-time object request broker (ORB), the comparisons between CIAO and TAO's functional and real-time features in particular offer a representative profile of the potential overheads and benefits available to other ORB implementations through application of the techniques described in this dissertation.

Section 6.1 first describes the experimental platforms on which the results presented in this section were obtained. Section 6.2 then compares the time and space overhead of CIAO and TAO for both functional and real-time aspects. Using a simple rate-based client-server design commonly found in avionics systems such as the example shown in Figure 4.1, Section 6.3 presents a performance evaluation of configuring real-time aspects using CIAO's extension and an analysis of the benefits offered through configuration of those aspects, either through CIAO directly or via a higher-level modeling tool. Finally, Section 6.4 summarizes the results presented in this chapter and offers observations and recommendations based on those results, as guidance for system developers wishing to use CIAO's ability to configure both functional and systemic system aspects.

6.1 Experimental Platform

Two single-CPU 2.8 GHz Pentium-4 computers served as deployment targets, thus providing the execution environment for the experiments performed in this chapter. Both machines ran KURT-Linux [14] 2.4.18, which is distributed as a Linux kernel patch. KURT-Linux, developed at the University of Kansas, provides a highly predictable platform for the experiments described in this chapter. Both machines had 512 MB of memory, with 512 KB of on-chip cache memory. Another two single-CPU 2.53 GHz Pentium-4 computers with the same memory configuration as the 2.8 GHz machines were used to deploy the test programs. All four machines were connected via switched Fast (100 Mbps) Ethernet as is shown in Figure 6.1.

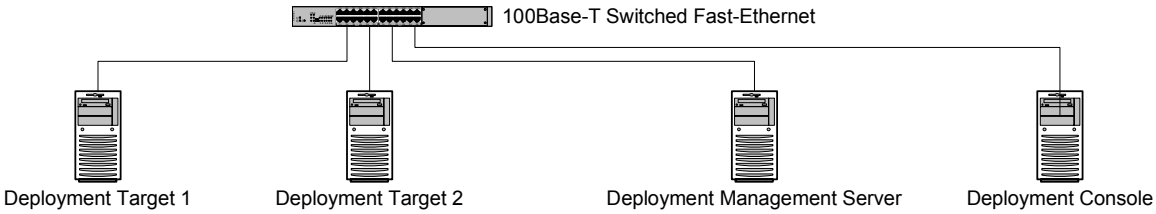

Figure 6.1: **Test-bed Configuration**

All the test programs, libraries, and tools were compiled using GCC version 3.2 with no embedded debug information, and with the highest level of optimization (-O3). All the tests were based on TAO version 1.3.5 and CIAO version 0.3.5. Because CIAO depends on dynamically loading component implementations at run-time, all the test programs are compiled to use dynamic libraries (as shared objects) in order to provide a realistic assessment of CIAO's performance and footprint. To remove spurious sources of variability from the tests, all the deployed applications were run as root in the real-time scheduling class, using the `SCHED_FIFO` policy.

6.2 Overhead Comparisons

Chapter 3 elucidates how a component-based application development framework makes developing complex large-scale applications easier and more flexible by separating concerns for designing, implementing, and deploying an application among developers with different areas of expertise. These benefits, however, do not come without cost, but instead

are achieved by adding extra layers and operations on top of conventional CORBA applications. The purpose of the experiments in Section 6.2.1 is to quantify the overhead of these added layers and operations by comparing performance of an application based on conventional and component middleware, *i.e.*, TAO and CIAO. These comparisons provide information developers need to assess the baseline feasibility of adopting component middleware approaches in general, and CIAO in particular.

As was described in Chapter 4.3 for DRE systems in particular, QoS provisioning must be made an integral part of the component framework to provide a flexible and effective solution to configuring *systemic* aspects of the system. The experiments in Section 6.2.2 compare the performance of an application using the real-time policies and mechanisms provided by TAO and CIAO.

Finally, Section 6.2.3 offers a detailed examination of the memory and disk sizes of the various libraries and executable software elements needed to support the test applications in this section, in both TAO and CIAO. Together with the performance comparisons in Sections 6.2.1 and 6.2.2, these experiments also help identify areas for future improvement. In particular the footprint results offer guidance on the areas where library refactoring in TAO could offer the greatest reductions in disk and memory requirements in CIAO, or where techniques like the `SOreduce` [33] effort by OCI could be applied as future work.

6.2.1 Performance Comparison for Functional Aspects

As Section 3.2 described, CCM components are hosted during execution by *containers*. CCM containers act as a bridge between the underlying ORB mechanisms and the component implementations by relaying operation invocations to and from components. This experiment aims to provide a quantitative performance comparison for the *functional* aspects of an application developed using TAO and CIAO, and identifies sources of overhead that may offset the development flexibility that CIAO achieves through the extra levels of indirection described in Section 3.2.

Experiment design: Figure 6.2 shows the interface definition used by both test programs. TAO's implementation of the test consists straightforwardly of a pair of client-server programs running on the two test machines. CIAO's test depends on two component implementations where one *provides* the target interface while another component performs the benchmarking operation by *using* the same interface. The CIAO test then can be built by using the deployment tools to deploy the two component to two separate machines.

```
module Benchmark {
  interface LatencyTest {
    long makeCall (in long send_time);
  };
};
```

Figure 6.2: **Interface Definition for Performance Tests**

This experiment measures and compares performance by invoking the simple operation repeatedly in each test using either TAO or CIAO. The performance metrics in this experiment were:

- **throughput**, the average number of operation invocations completed per second;
- **latency,** the time to complete each invocation of the operation; and
- **jitter,** which measures the standard deviation in latency of all invocations sampled, as well as the maximum measured latency and the latency within which 99% of the samples fell.

As is shown in Figure 5.3 on page 65, an operation invocation on a component will incur an additional virtual function call for a generated servant to forward the invocation to the executor. Likewise, when a component invokes an operation on a receptacle interface, it needs to retrieve the reference before invoking the operation on it. The cost for both the virtual function call and the retrieval of the object reference should be constant and relatively small.

This experiment therefore selects an operation signature (its argument and return type) resulting in a small message payload, to make the overhead of CIAO more significant in comparison and offer an approximation of the worst case CIAO performance for non-trivial method invocations. Not including the length of other protocol headers, the message payload going over the wire is kept to only 8 bytes. The small payload size reduces the time the operation spends in marshaling the data and sending message over the wire, which both TAO and CIAO tests require, and thus makes CIAO's overhead stand out in comparison to TAO.

Experimental results: Figure 6.3 shows the throughput results of TAO and CIAO to be 9219 and 9144 calls/sec respectively over a sequence of approximately 10,000 repeated

calls. The results show that using CIAO only reduces throughput by a modest 0.81% compared to TAO's throughput performance. The relative performance difference would be even less significant for operations that carried larger payloads. The latency and jitter

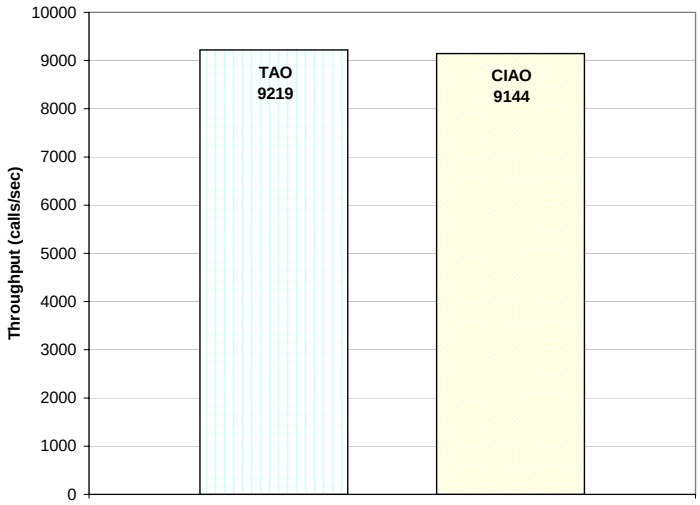

Figure 6.3: **Throughput Comparison between TAO and** CIAO

results gathered from the same experiment are shown in Figure 6.4. The mean latency numbers of TAO and CIAO are 107.3 μsec and 109.1 μsec respectively, which indicate a 1.8 μsec, *i.e.*, about 1.68% increase in mean latency. This is consistent with the throughput results and shows that the difference in relative overhead imposed by CIAO compared to TAO is small.

The other graphs in Figure 6.4 show the jitter in the latency measurements in this experiment in terms of standard deviation, the time within which 99% of all samples fell, and the maximum measured latency. The standard deviation for both TAO and CIAO were both vary small – around 1% of the measured mean latency. 99% of TAO and CIAO latency samples are under 109 μsec and 111 μsec respectively, which are relative close to the respective mean latency numbers. Finally, the maximum measured latencies for TAO and CIAO were comparable, at 141 μsec and 156 μsec, respectively. Complete distributions of all 10,000 latency samples for both TAO and CIAO tests, shown in Figure 6.5, also reveal that both tests had similar jitter.

Analysis of results: The experiment results show that the extra level of indirection in CIAO adds only a small overhead to the conventional TAO ORB as documented above. Moreover, CIAO does not adversely affect the the jitter and worst case performance, which

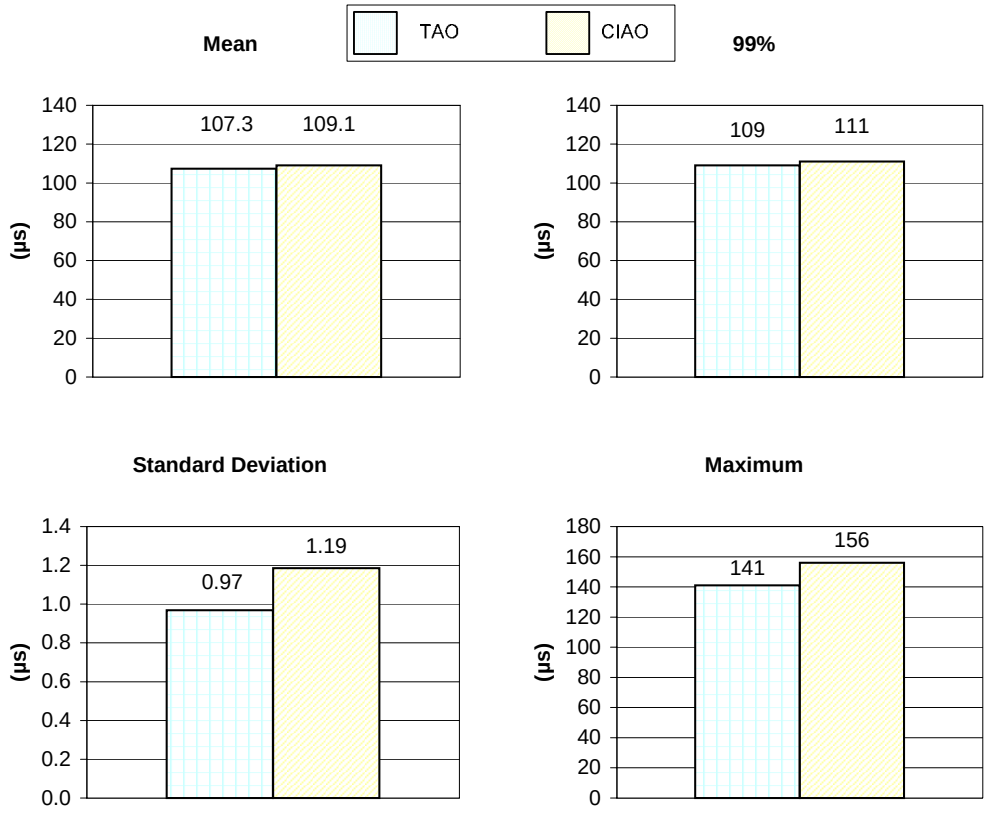

Figure 6.4: **Latency Comparison between TAO and** CIAO

are of greater importance to many DRE systems. As the results from this experiment show, both CIAO and TAO performed similarly.

6.2.2 Performance Comparison for Real-time Aspects

CCM containers can also apply meta-programming techniques in order to configure different systemic aspects of CCM components, such as priorities or rates of invocation. A fundamental extension to CIAO's capability to configure *functional* aspects is the addition of a real-time ComponentServer run-time environment.

Experiment design: This part of the experiment evaluates the performance overhead to componentize real-time applications, by repeating the experiment in Section 6.2.1 but with RT-CORBA features enabled in the TAO ORB and using CIAO's real-time ComponentServer environment. Other than that difference, this experiment was conducted in an

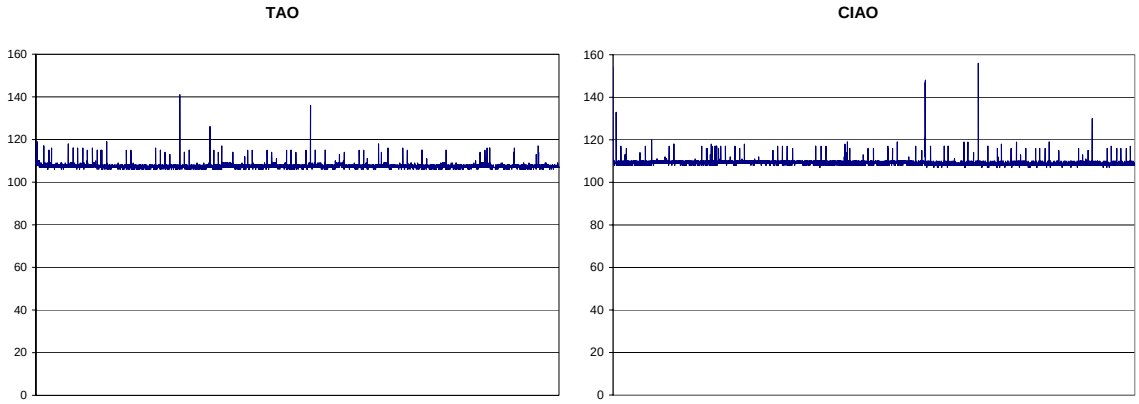

Figure 6.5: **Complete Latency Distributions for TAO and CIAO in μsec**

identical manner to the experiment described in Section 6.2.1, and the same metrics, *i.e.*, throughput, latency, and jitter were evaluated.

Experimental results: Figure 6.6 shows the throughput results of these real-time benchmarking tests using TAO and CIAO to be 8420 and 8107 calls/sec respectively. The results show that when real-time ORB features are enabled, CIAO suffers a 3.7% reduction in mean throughput compared to TAO.

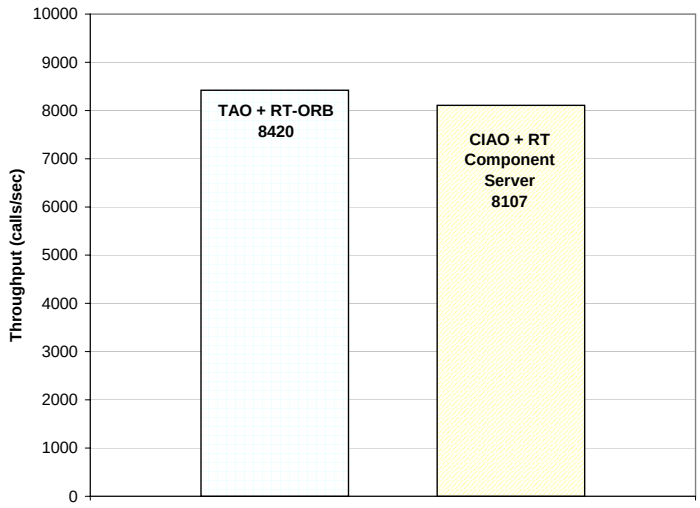

Figure 6.6: **Throughput Comparison between Real-Time Enabled TAO and CIAO**

The latency results from this experiment are shown in Figure 6.7. The mean latencies of TAO and CIAO calls were 118.9μsec and 122.9 μsec respectively, indicating an increase of 4 μsec or $\sim$3.4% in mean latency. This is consistent with the TAO real-time

ORB and CIAO real-time ComponentServer throughput results, and shows that the overhead imposed by CIAO's implementation when using real-time CORBA is still very small, albeit exacerbated by the addition of real-time features. As in the case of the functional aspect performance test, CIAO's relative performance cost in terms of throughput and latency is reasonably expected to diminish with any increase in payload size.

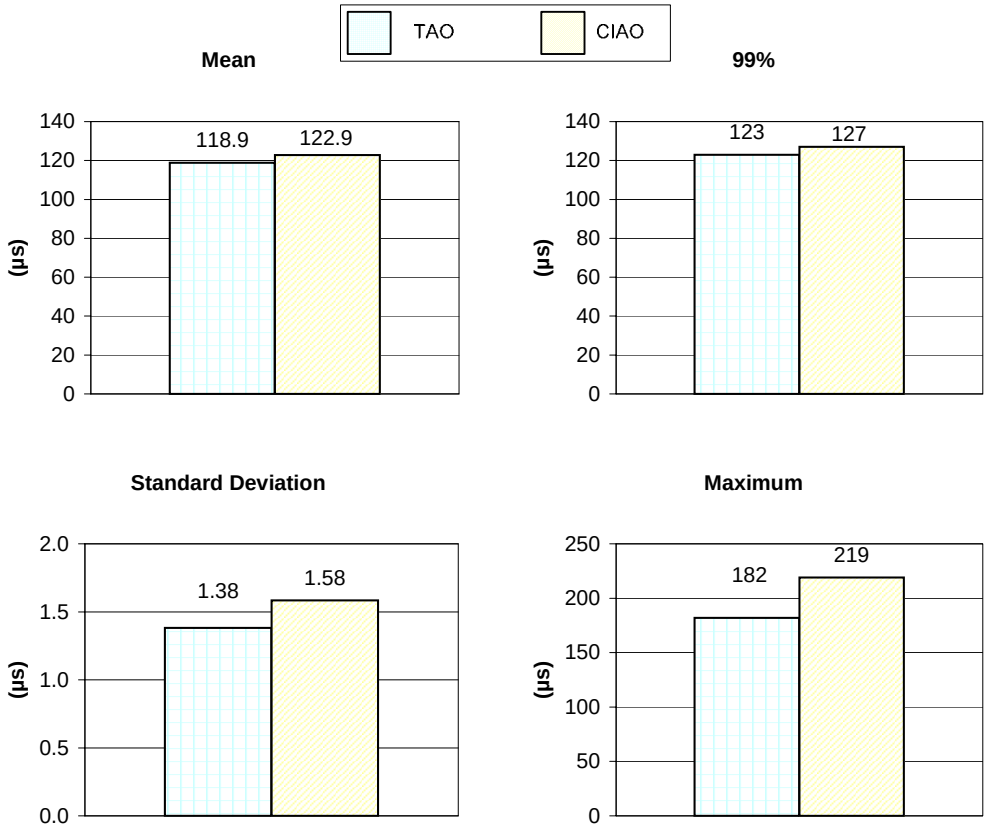

Figure 6.7: **Latency Comparison between Real-Time Enabled TAO and** CIAO

The other graphs in Figure 6.7 show the standard deviations, 99% latency bounds, and maximum measured latencies for TAO and CIAO with real-time ORB features enabled. The standard deviations were again both small: less than 2 μsec for both TAO and CIAO real-time tests. TAO's measured real-time latency had 99% of all samples under 123 μsec, and 99% of the measurements for CIAO fell within 127 μsec. In both cases 99% of all samples fell within $\sim$4 μsec above their mean latencies. The maximum latency results for TAO and CIAO tests were also comparable, at 182 μsec for TAO and 219 μsec for CIAO. Figure 6.8 also shows that both TAO and CIAO real-time tests have the similar jitter based on the plots of all latency samples.

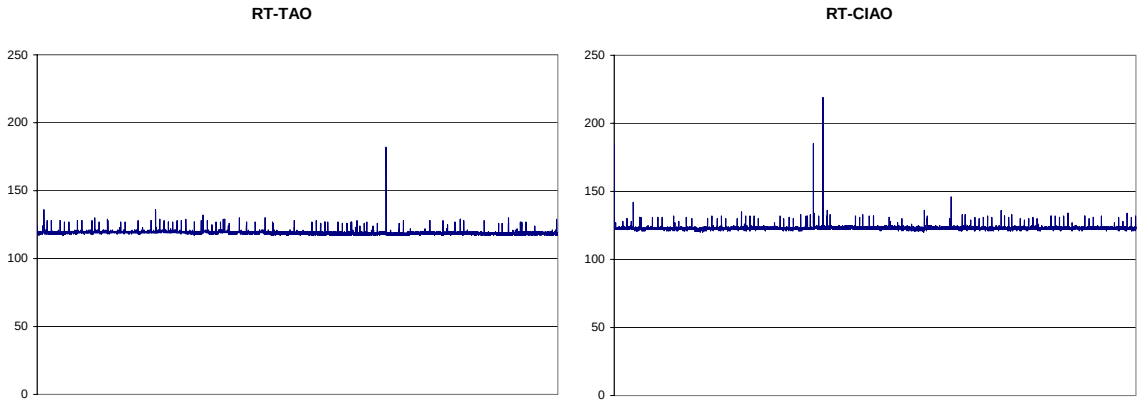

Figure 6.8: **Complete Latency Distributions for Real-time TAO and CIAO in μsec**

Analysis of results: Together these results show that here a similar conclusion can be drawn to that in Section 6.2.1, that with real-time features enabled the mean and worst case performance for CIAO was slightly worse than for TAO, but was reasonably close overall.

6.2.3 Footprint Comparisons

Experiment design: Application footprint is seldom an issue for most enterprise applications, as both memory and secondary storage have become more and more affordable and available. Many computer systems on which DRE applications run do however have much stricter memory constraints due to physical or budgetary limitations. As a consequence, increases in software footprint can have direct and significant effects on the overall weight, power consumption, and heat dissipation in end products, which are key issues for certain types of DRE applications. This section compares the memory and disk storage requirements for running the benchmarking test applications described in Sections 6.2.1 and 6.2.2, using TAO and CIAO respectively.

The footprint metrics considered here include both the file size and the memory size, counting both code and data sections reported by the GNU *size* utility, of the required middleware libraries and executables. This comparison does not include system libraries, such as glibc or standard C++ libraries. This experiment aims to compare and document the footprint increase for running a comparable application in CIAO and to provide hints on how to cut down the memory requirement in CIAO.

Experiment results: Table 6.1 lists the sizes for server-side and client-side common libraries for TAO programs. Table 6.2 lists the total sizes of server-side and client-side

Table 6.1: **Storage Required for Server-side and Client-side Common Libraries**

Library	Server Libraries		Client Libraries	
	File	*size*	File	*size*
ACE	1,905,727	1,504,346	1,905,727	1,504,346
TAO	2,739,052	2,168,541	2,739,052	2,168,541
Valuetype	61,213	42,518	61,213	42,518
Strategies	669,326	499,404	669,326	499,404
PortableServer	2,344,311	1,770,156		
ObjRefTemplate	61,501	41,187		
IORInterceptor	48,415	29,493		
Total	7,829,545	6,055,645	5,375,318	4,214,809

Table 6.2: **Storage Required for Server and Client Executables**

Library	Server		Client	
	File	*size*	File	*size*
Common Libraries	7,829,545	6,055,645	5,375,318	4,214,809
Executables	162,713	107,213	78,116	46,578
Total	7,992,258	6,162,858	5,453,434	4,261,387

executables plus the total common library sizes listed in Table 6.1 that are required by the client and server to run the basic TAO benchmark tests.

These numbers represents the amount of secondary and primary storage needed respectively to store and run the basic *functional* TAO benchmark programs on the server and client side machines.

Table 6.3 lists the sizes for the common run-time support, including all the CIAO-specific shared libraries and executables necessary for a machine to become a deployment target. This includes the object libraries and executables needed to run the deployment daemon and all of the CIAO run-time supporting libraries. Note that the CIAO run-time environment always requires the set of common libraries in TAO, as CIAO's run-time support programs can always act as servers. As seen in this table, it requires almost 18 MB of secondary storage space to hold the libraries and executables for deploying a CIAO server. Each target machine also needs to have a deployment daemon running to start up component server processes as was described in Section 5.3. Many libraries on which the CIAO_Daemon executable depends overlap with those required by the component server. The total secondary storage space for the deployment and run-time support programs is

Table 6.3: **Storage Required for CIAO Common Run-time Support Libraries**

CIAO tools Common Libraries		
Libraries & Executables	File	*size*
Common Server Libraries	7,829,545	6,055,645
Security	1,104,741	832,149
IFR_Client	4,264,757	3,431,512
IORTable	76,966	54,473
CIAO_Client	1,473,327	1,114,966
CIAO_Container	1,720,269	1,320,232
CIAO_Server	1,350,701	1,016,489
XML_Common	198,638	144,575
XML_Parser	92,046	74,713
CIAO_XML_Helpers	266,486	206,284
CIAO_Daemon	211,443	147,047
Total	18,588,919	14,343,611

Table 6.4: **Storage Required for Benchmark Client and Server Components in CIAO**

Library	Server Component		Client Component	
	File	*size*	File	*size*
Stub	234,429	164,307	265,155	184,383
Servant Glue Code	486,926	355,428	563,041	409,720
Executor	98,847	67,498	103,651	70,400
Total	820,202	587,233	931,847	664,503

then the sum of the storage needed for CIAO_Daemon, and all the libraries they depend on, which comes to 18,588,919 bytes.

In addition to the common runtime support and deployment infrastructure required to deploy and run the application, the storage required by the component libraries themselves must also be taken into account. Table 6.4 lists the libraries for the client and server components of the benchmark tests. Less than 1 MB of storage space on the deployment target machine is required for the component implementations. We can then calculate the total disk space required by a target linux machine to become a deployment target for deploying either server or client components to be $\sim$ 19 GB as shown in Table 6.5.

The memory space (*i.e.*, the primary storage) required to execute each application is equally important. Table 6.6 then compares both the secondary storage required to hold the benchmark client and server and the run-time memory footprints when the bench client

Table 6.5: **Storage Required for Deploying Benchmark Client and Server Components in** CIAO

Library	Server		Client	
	File	*size*	File	*size*
Component	820,202	587,233	931,847	664,503
Common Run-time	18,588,919	14,343,611	18,588,919	14,343,611
ComponentServer	32,674	15,687	32,674	15,687
Grand Total	19,441,795	14,946,531	19,553,440	15,023,801

Table 6.6: **Total Secondary Storage and Memory Required for Benchmark Client and Server Using Either TAO and** CIAO

	Server		Client	
	Disk Space	Memory Size	Disk Space	Memory Size
TAO	7,805 K	5,568 K	5,326 K	4,220 K
CIAO	18,968 K	11,036 K	19,095 K	11,240 K

and server first start up with TAO or CIAO. Figure 6.9 shows head-to-head comparison of disk space requirements for storing TAO and CIAO client and server programs on target machines.

We now review and compare the secondary storage and the memory footprints required to support real-time behaviors in both TAO and CIAO. Table 6.7 lists the additional secondary storage space required for the TAO server or client to support real-time behavior. The additional storage required by both server and client are only moderate, 820 KB and 640 KB each. To deploy a real-time CIAO application, a target platform must also support the additional server-side common RT libraries. CIAO also provides a special real-time component server that is capable of configuring additional real-time policies for applications. Table 6.8 lists the secondary storage requirements for a deployment target to install

Table 6.7: **Storage Required for Additional Server-side and Client-side Real-time Libraries**

Library	Server Libraries		Client Libraries	
	File	*size*	File	*size*
RTCORBA	664,830	512,486	664,830	512,486
RT-PortableServer	170,450	120,753		
Total	835,280	633,239	664,830	512,486

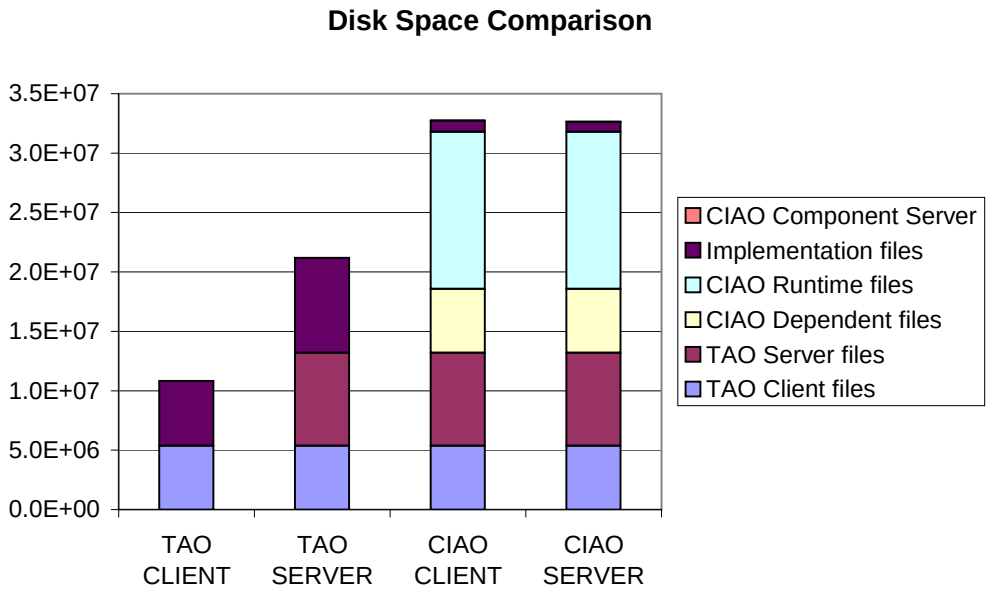

Figure 6.9: **Comparison of Disk Space Required for TAO and CIAO in Bytes**

Table 6.8: **Storage Required for Deploying Real-time Benchmark Client and Server Components in CIAO**

Library	Server		Client	
	File	*size*	File	*size*
Component	820,202	587,233	931,847	664,503
Common Run-time	18,588,919	14,343,611	18,588,919	14,343,611
Common RT Server Libs	835,280	633,239	835,280	633,239
RTComponentServer	124,125	81,749	124,125	81,749
Grand Total	20,368,526	15,645,832	20,480,171	15,703,102

a server or client component in the real-time benchmark test. Finally, Table 6.9 compares both the secondary storage requirements and the memory footprint for running a real-time server or client using TAO and CIAO. As can be seen comparing Table 6.6 and Table 6.9, adding real-time capabilities only adds a modest additional storage requirement. Figure 6.10 illustrates the comparison of disk space requirements for storing real-time enabled TAO and CIAO client and server programs on target machines.

Analysis of results: The tables showed in this section demonstrate that the current implementation of CIAO entails a notable increase in overall system storage and memory requirements. However, many services and interfaces in the list of CIAO common libraries

Table 6.9: **Total Secondary Storage and Memory Required for Benchmark Real-time Client and Server Using TAO and CIAO**

	Server		Client	
	Disk Space	Memory Size	Disk Space	Memory Size
TAO	8,625 K	5,856 K	5,974 K	4,568 K
CIAO	19,891 K	11,984 K	20,000 K	12,192 K

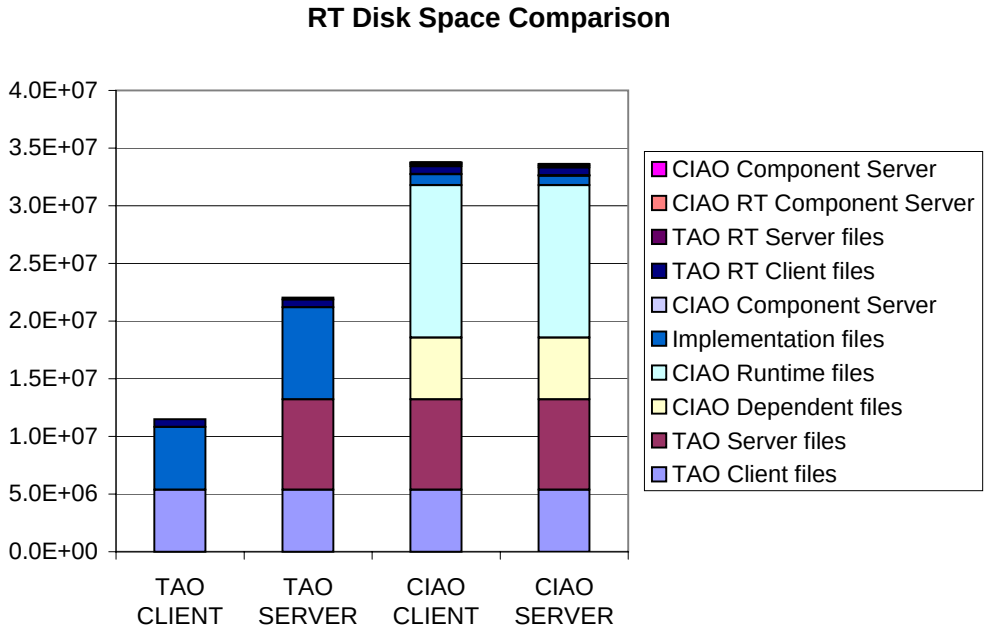

Figure 6.10: **Comparison of Disk Space Required for Real-time Enabled TAO and CIAO in Bytes**

are not actually used by CIAO or its components. Furthermore, some services may not in fact be useful for all application domains. For example, CIAO currently does not support the Security Service or utilize the Interface Repository. Many DRE applications will not need either of these services as they often avoid the more dynamic features of CORBA that require the Interface Repository, and exist within isolated deployment environments where security is already provided.

However, these libraries are still currently required by many CIAO libraries because their interfaces are used by standard CCM interfaces. Additional future work is motivated by these results, to decouple those dependencies. This will allow CIAO infrastructure to be more dynamically configured and requires only libraries based on the application requirement.

In addition, CIAO applications are hosted inside a component server that in principle could be linked either statically or dynamically to other libraries. While CIAO uses *dynamically* linked shared object libraries to achieve its run-time configuration and deployment capabilities, the author has also contributed to the design of *statically* linked compile-time assembly and run-time configuration capabilities in CIAO for use on VxWorks and other conventional real-time operating systems, to reduce code size, avoid dependence on dynamic libraries which are not supported on those platforms, and improve predictability of system initialization times. These capabilities have been prototyped as part of an ongoing collaboration between the DOC groups at Washington University and Vanderbilt University, and are currently undergoing experimentation within the DARPA PCES program.

6.3 Validating Effectiveness in Configuring Real-Time Aspects

The experiments in this section evaluate the effectiveness of CIAO's support for composing systemic aspects to achieve real-time behavior. They also illustrate how different real-time behaviors can be composed and configured into existing applications through the use of CIAO's real-time extensions.

6.3.1 Validating CIAO's Real-time Extensions

The performance experiments outlined in Section 6.2 show that CIAO introduces only a small amount of overhead to the run-time environment for real-time applications. Experiments in this section, on the other hand, aim to assess the *effectiveness* of CIAO's ability to compose real-time behaviors into existing application components. To achieve this goal, all experiments used simple components whose functional implementation was amenable to configuration of but decoupled from any real-time aspects. Different real-time aspects were then composed with them in these experiments, to model several of the original real-time tests described in earlier work on RT-CORBA features in TAO [53]. Those original experiments will be referred to as TAO RT tests hereafter.

Experiment Implementation:

Unlike the TAO RT tests, where procedures for different tests were hard-coded into many client and server execution paths, which in turn depend on complicated logic and scripts to

determine the exact tests to perform, the CIAO tests conducted in this dissertation consisted of only a handful of simple component implementations. With CIAO, different tests were composed, instead of programmed, by selecting and connecting different combinations of components and systemic policies. Therefore, only a few component implementations are needed to perform the tests using CIAO.

The basic interactions in the TAO RT tests occurred between a test object provided by the server and a client invoking an operation on the object, thus requesting the server to perform a given amount of CPU-intensive work. Different tests were derived from different configurations of the server and client. For example, the number of objects served by the server and their real-time constraints were varied. Also, clients were given different numbers of threads, each invoking server objects with different workloads in different ways, *e.g.*, at a fixed rate vs. continuously.

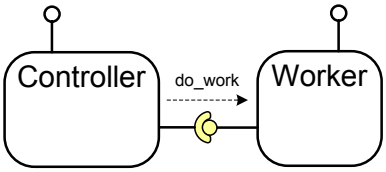

Figure 6.11: **Basic CIAO RT Experiment Design**

As Figure 6.11 shows, the CIAO RT tests needed only two basic component types, called Controller and Worker, to emulate the TAO RT tests. A Worker component *provides* a common interface which contains the following operation that a client can invoke with an `in` parameter named `work` to specify the number of repetitions of a fixed increment of work to perform within that invocation:

```
long do_work (in long work);
```

The CIAO RT tests only required one component implementation for the Worker component that performs the specified amount of CPU computation when requested.

A Controller component *uses* a common interface to request that a connected Worker component perform a given amount of work. Several Controller implementations are provided for the experiments. Each Controller implementation realizes a particular invocation strategy, such as *continuous* or *rate-based* at 25, 50, or 75 Hz. A Controller component also *supports* an interface for starting and stopping the test operation and performing output of the statistic results observed in the controller. Multiple controllers then act as sources of

execution threads invoking operations on the server component at different rates. These experiments follow the design commonly found in avionics systems of the kind shown in Figure 4.1 and described in Section 4.4.4.

Validating Experiments:

We now describe the experiments performed, present the results of those experiments, and explain how these results validate CIAO's support for composing real-time aspects. All experiments performed in this section use the real-time component server to ensure that all experiments are executed in the same environment and that the composed real-time behaviors are realized using state-of-the-art ORB middleware features, from TAO.

Workload vs. invocation rate: This experiment measures the maximum frequency a server can sustain to complete tasks at different workloads.

Experiment design: Figure 6.12 illustrates the design of the experiment. In the test application, a continuous controller component starts requesting the connected worker to do a certain fixed amount of work continuously. Upon completion of pre-defined iterations of requests, the controller calculates the frequency at which the worker can perform under the given workload. The workload is defined by the number of repetitions of a simple computation that the worker performs for each request from the controller. That simple computation consists of the CPU intensive operation of checking whether a big prime number is or is not prime.

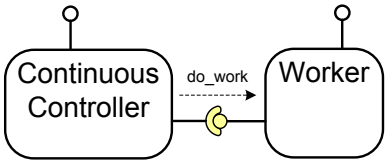

Figure 6.12: **Experiment Design for Workload vs. Rate**

Experiment results: Figure 6.13 shows the maximum number of requests the server can handle under different workloads.

Analysis of results: As the computational capacity of a server is limited, the rate ($\mathcal{R}$) at which a server can handle requests decreases as the workload ($\mathcal{W}$) increases as

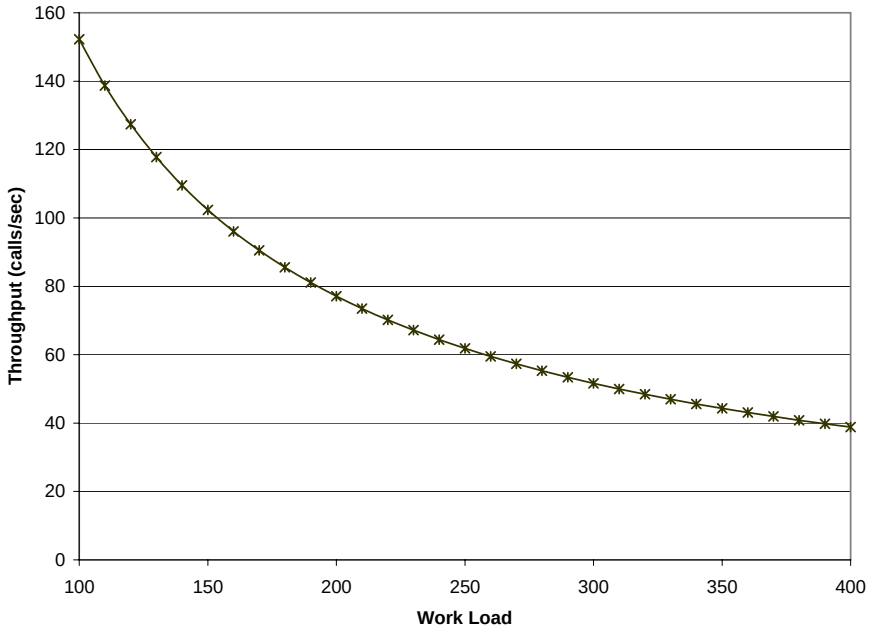

Figure 6.13: **Relationship between Workload and Invocation Rate**

shown in the following equation:

$$\frac{k}{\mathcal{W}+c} = \mathcal{R}$$

where k and c are both constants specific to the computational power of the platform.

Multiple fixed-rate controller-worker pairs: This experiment demonstrates the behavior of a server handling multiple worker threads over a range of workloads without any scheduling.

Experiment design: This experiment consisted of 3 Controller-Worker pairs running concurrently, as shown in Figure 6.14. Each of the three controllers made requests to the worker at its respective fixed rate of 25, 50, or 75 hertz. The worker handles requests from each individual controller by doing the specified amount of work in a separate thread. When invoking an operation on the worker, a controller will block until the invocation returns from the Worker. Therefore, if the Worker can not handle the request at the given rate, a Controller will not be able to invoke operations at its designed rate.

This experiment measures the rate at which each of the three controllers can make requests to the worker component under different workloads. The total work performed by all 3 worker threads eventually exceeds the amount of work the server can handle. As

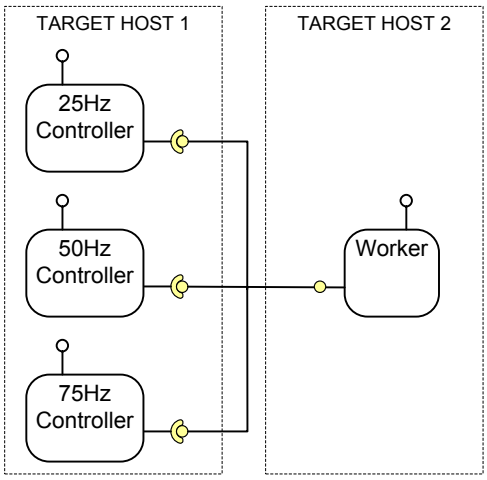

Figure 6.14: **Experiment Design for Workload vs. Rate**

shown in Figure 6.13, the server has only limited capability to perform work. Because all three worker threads perform the same amount of work, we expect one or more worker threads will not be able to maintain the designated rate when the workload increases to the point where the server is not able to perform at a rate higher than 150 Hz.

Experiment results: Figure 6.15 shows the measured results from the experiment: when the workload increases initially from 20, all 3 controllers can achieve their designed rates. However, once the workload increases to above 110 repetitions per invocation, the 75 Hz controller starts to fall short of its target rate. Similarly, above a workload of 130 repetitions the 50 Hz controller also falls short of its target rate, and above 210 repetitions all three controllers fail to perform at their designed rates of invocation.

Analysis of results: The result observed is in consistent with the results observed in the previous "Rate vs. Workload" experiment, *i.e.*, as workload increases the rate at which Workers can handle requests eventually decreases. However, failing to meet the invocation rate of controllers indiscriminately is often not acceptable for most DRE applications. Instead, certain tasks must be able to meet their execution rates even if there are not sufficient computational resources to enable all the tasks to run at their specified rates. The following experiment presents two different ways to compose real-time aspects into an application flexibly to meet this application requirement.

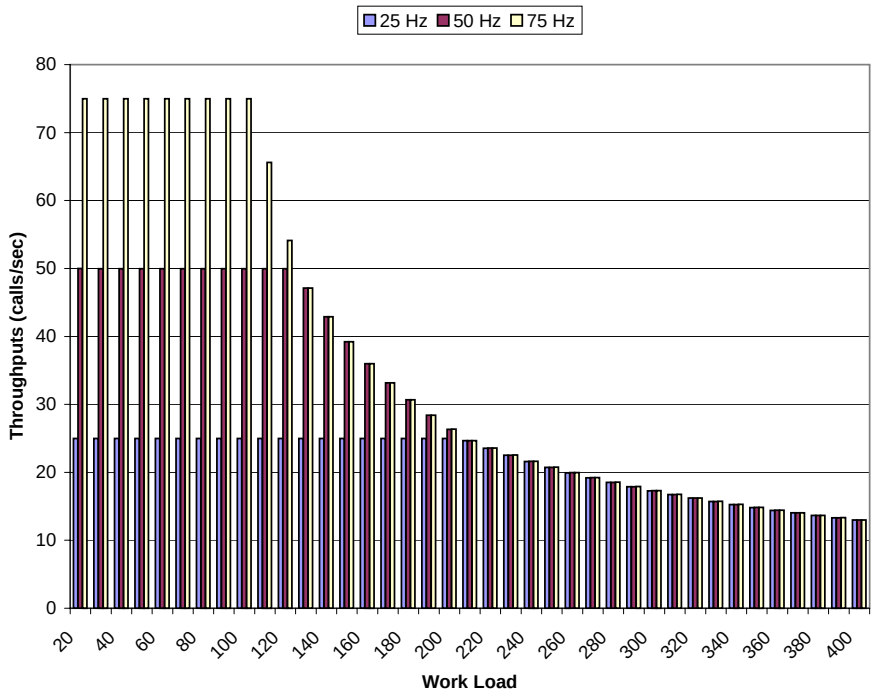

Figure 6.15: **Achievable Rates vs. Workload**

Prioritized workers using separate threadpools: This experiment intents to show how real-time behaviors can be composed into an application during the CIAO assembly phase, to satisfy systemic requirements that are specific to the application.

Experiment design: This experiment uses an RTCAD extension file described in Section 5.5 to specify resources and policy sets a component server must provide and defines policy sets that can be associated with various entities in an application assembly. Real-time systemic behaviors can then be composed into the experimental application assembly by adding the RT aspect configuration parameters each component instance needs, to the assembly files. This experiment extends the previous experiment by composing either “increase rate, increase priority” or “increase rate, decrease priority” prioritization aspects into the assembly descriptor.

The real-time aspects defined in this experiment include several server-declared priority policies with different priority levels. Each priority-level policy is then associated with a single-priority thread pool. Because each Worker component instance uses a policy set that is different from the other Worker instances, the deployment tools install each of the three Worker instances in different containers.

Figure 6.16 (a) shows the application configuration after applying this approach to support "increase rate, increase priority" real-time behavior, *i.e.*, the rate monotonic scheduling (RMS) strategy. RMS is a canonical priority-based scheduling strategy [17]. For certain systems, however, RMS may not provide the kind of QoS required. Using non-

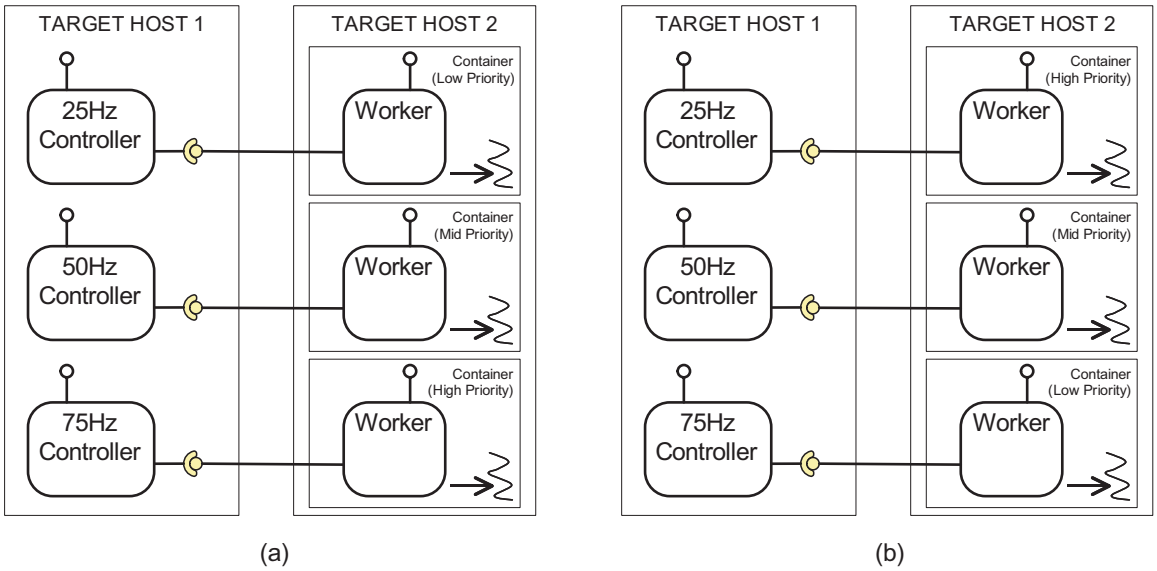

Figure 6.16: **Experiment Design for Multi-rate Test With (a) "Increase Rate, Increase Priority" Behavior, and (b) "Increase-rate, Decrease Priority" Behavior**

RMS strategies such as maximum urgency first (MUF) for these systems may out-perform the case when RMS is used. The next experiment therefore composes an anti-RMS real-time scheduling strategy into the test application as shown in Figure 6.16 (b).

Experiment results: Figure 6.17 shows the resulting achievable rates of Controllers at the 3 specified rates after composing the "increase rate, increase priority" real-time behavior into the experiment assembly. As the figure reveals, the 75 Hz Controller is now able to maintain its rate at the expense of Controllers at lower rates, as dictated by the composed real-time behavior. The 75 Hz Controller eventually fails to maintain the rate as workload increases, but that is because the computer hosting the Workers simply can not sustained the rate under that controller's own heavy workload, as shown in Figure 6.13.

Figure 6.18 shows the resulting performance after composing the anti-RMS real-time behavior into the test, *i.e.*, "increase rate, decrease priority". As can be observed from the figure, the Worker connected to the 25 Hz Controller is now deemed more important

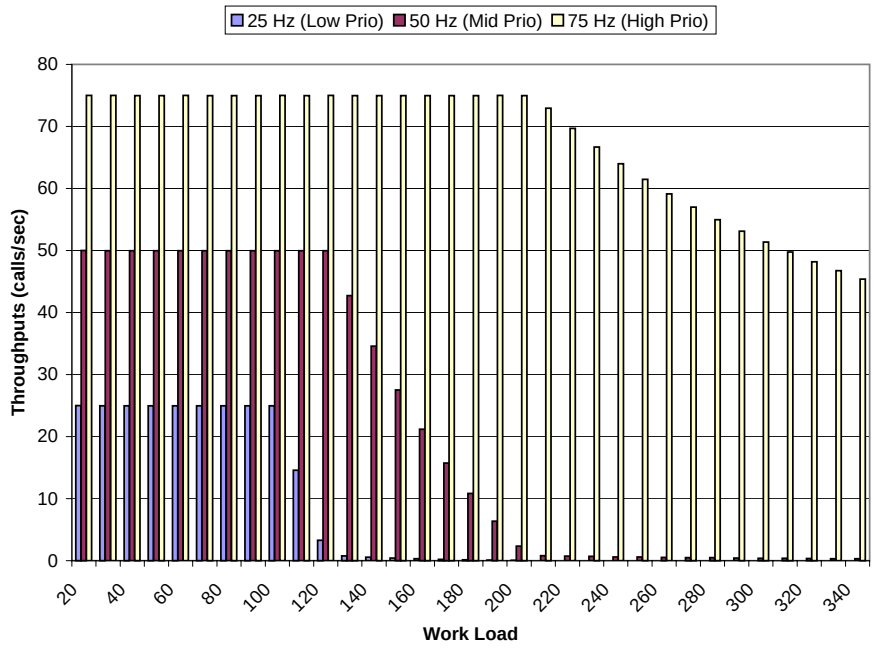

Figure 6.17: **Achievable Rates vs. Workload When Using "Increase Rate, Increase Priority" Real-time Behavior**

in the composed behavior. Therefore, as workload increases, other less important Workers, starting from the lowest priority 75 Hz one, have to yield computational resources to the 25 Hz Worker.

Analysis of results: The experiment results show that the composed real-time behaviors did successfully add the desired systemic aspects, *i.e.*, prioritizing task handling. In the experiments, real-time aspects were composed at different stages, *i.e.*, real-time CORBA policies and resources at the component assembly stage and certain real-time ORB configurations at the deployment stage. Furthermore, the applied RTCAD file also utilizes CIAO's support to compose real-time behaviors to different granularities in an application, *i.e.*, threadpool configurations at the per-ORB level and sets of real-time policies at the container level. The way the experiments composed the real-time policies demonstrates how CIAO is using the solution outlined in Section 4.4.3.

The two tests performed in the experiment also show how the RTCAD extension files help in managing the consistency of real-time systemic behaviors flexibly. An application can adopt a different real-time strategy by either using different RTCAD files which provide different definitions of real-time policy sets, or by changing the application assembly file to use real-time policy sets differently according to their logical names.

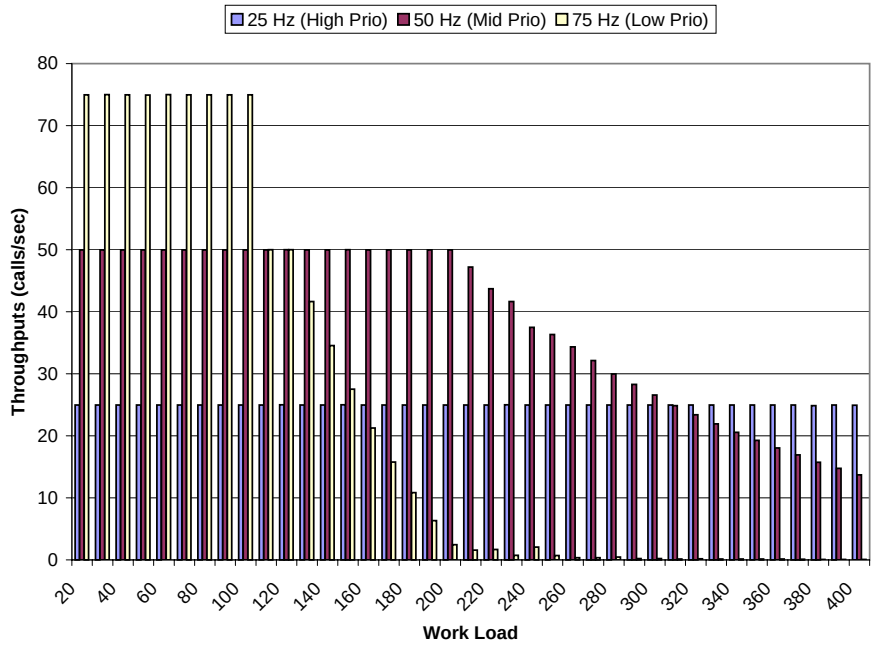

Figure 6.18: **Achievable Rates vs. Workload When Using "Increase Rate, Decrease Priority" Real-time Behavior**

Prioritized workers using threadpools with lanes: This experiment also shows the effectiveness of composing real-time behaviors into an application, but using a different strategy, *i.e.*, threadpools with lanes, for managing the reservation of processing resources.

Experiment design: Figure 6.19 shows an alternative approach to compose the RMS, *i.e.*, "increase rate, increase priority" real-time aspect, into the test assembly. In this approach, instead of creating multiple thread pools for every priority level, the same effect is achieved in this experiment by allocating a thread pool with different lanes for different priorities. Although the deployment tools still create multiple containers for host components at different priorities, they all share the same thread pool using this approach. The same experiment is also performed with the "increase rate, decrease priority" real-time behaviors.

Experiment results: The result of the alternative approach of using a threadpool with lanes is shown in Figure 6.20 which yields the same result as that of using RMS with individual threadpools. Similarly, the result graph of composing the anti-RMS real-time behaviors with threadpool with lanes in Figure 6.21 shows the same effect is achieved using the alternative approach.

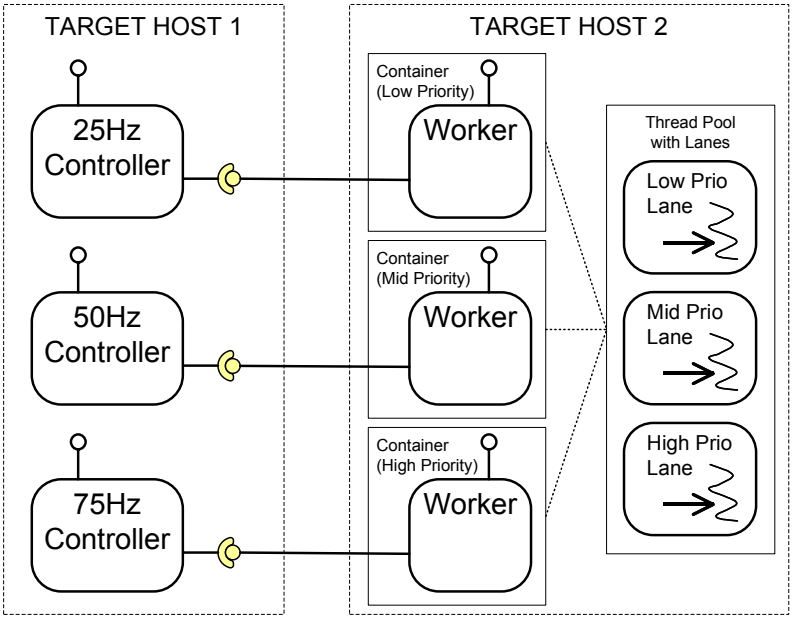

Figure 6.19: **Experiment Design for Multi-rate Test With "Increase Rate, Decrease Priority" Behavior Using Shared Thread Pool With Lanes**

Analysis of results: As expected, the alternate real-time strategies applied in this experiment demonstrate the same effectiveness. More importantly, the series of real-time experiments have illustrated how integrating real-time behaviors declaratively can help address the limitations of conventional CCM and make it suitable for DRE applications. The validating experiments are modeled after the experiments conducted in TAO's Real-time CORBA work [53]. The benefit of bringing the CCM development paradigm to bear is evident when comparing the equivalent test programs used for TAO and those used for CIAO. TAO's RT experiments require complex logic in both the client and server test programs and the collaboration of complicated scripts to cover configurations for all real-time behaviors performed. In comparison, CIAO based tests are *composed* by using simple component implementations and XML definitions to specify the test applications and real-time systemic behaviors, which are much easier to reason about, maintain, and manage.

6.3.2 Exploiting Configuration Phases

As was pointed out in Section 4.4.2, systemic behaviors can be both documented and configured at the component implementation, application assembly, or deployment stages of the CIAO development lifecycle. The extra information captured at each stage can enable

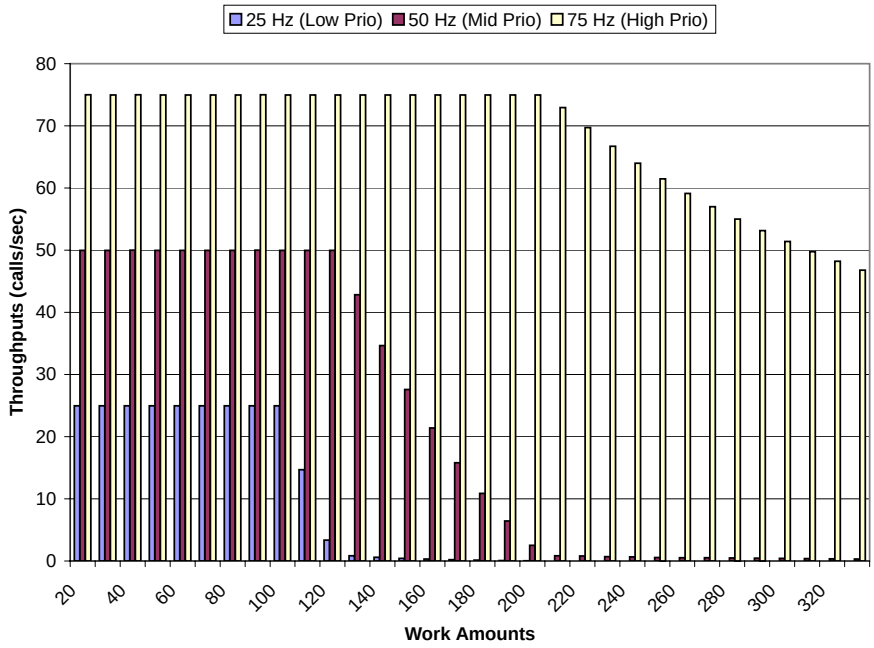

Figure 6.20: **Achievable Rates vs. Workload When Using "Increase Rate, Increase Priority" Real-time Behavior with Threadpool with Lanes**

a developer using CIAO either directly, or in an automated manner via third-party modeling tools, to reason about and configure the resource allocations, priorities, rates and other real-time aspects of an application in the later stages of development lifecycle, *i.e.*, in the application assembly and deployment stages. Such a capability in turn allows key properties of the resulting real-time performance, *e.g.*, feasible allocation of resources for deadline assurance, to be *checked*. It also allows, within the checked constraints, other properties like rates of invocation to be *optimized* with respect to various objective functions.

The experiment in this section will demonstrate how the additional information can be used by both developers and modeling tools, to ensure correctness and enable optimization of CIAO application assemblies and execution environments, which can not be achieved either easily or systematically with TAO-based applications.

Experiment design: Suppose we were developing a simple application similar to the Basic RT test shown in Figure 6.11. However, instead of a continuous controller which requests that the worker do work continuously, the target application needs to be able to invoke the worker component to do work at some fixed rate without failing to meet that specified rate. Suppose further that higher rates yield higher utility for the application, so

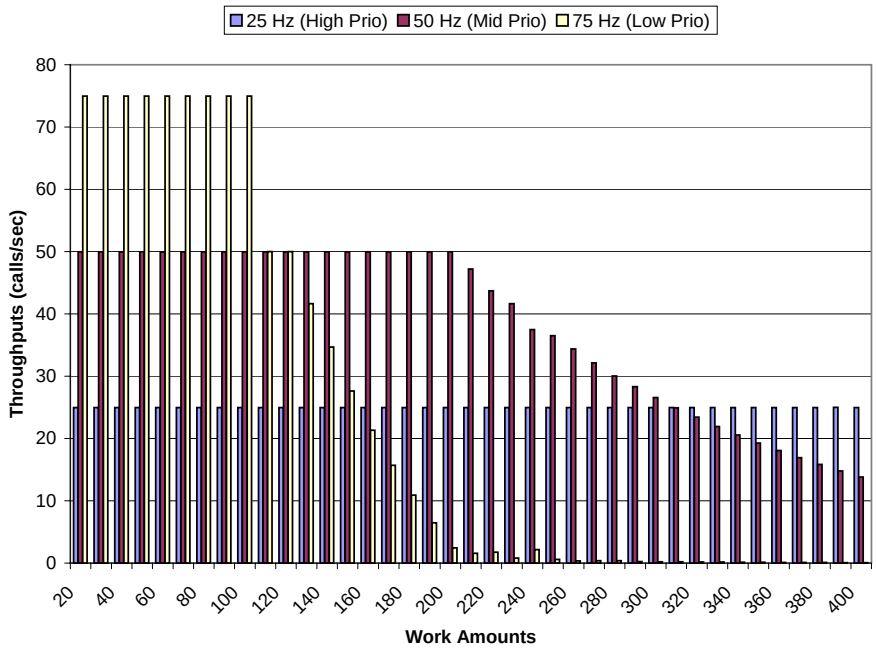

Figure 6.21: **Achievable Rates vs. Workload When Using "Increase Rate, Decrease Priority" Real-time Behavior with Threadpool with Lanes**

that the ideal configuration is the highest *sustainable* rate for each worker running on a server.

Experiment results: Figure 6.22 shows the maximum achievable rates the worker implementation can achieve for different workloads on machines with different capabilities to which workers can potentially be deployed. As may be reasonably expected, Figure 6.22 shows that the worker component implementation can do work at higher rates when running on a 2.8 GHz Pentium-4 machine than it does on a 2.53 GHz Pentium-4 machine, and higher there than it does on a 930 MHz Pentium-III machine. For example, when performing a workload of 150 repetitions, the worker component implementation can perform work at $\sim$ 103 Hz on a 2.8 GHz machine but only $\sim$ 95 Hz on a 2.53 GHz machine and $\sim$ 34 Hz on a 930 MHz machine. In fact, as described in Section 6.3, on each platform the relationship between the amount of work ($\mathcal{W}$) and the achievable rates ($\mathcal{R}$) can be described in the following formula:

$$\frac{k}{\mathcal{W}+c} = \mathcal{R}$$

Table 6.10: **Constants for Workload and Rate Relationship**

CPU Types	k	c
2.8 GHz Pentium-4	15626	1.73
2.53 GHz Pentium-4	14332	1.66
930 MHz Pentium-3	5105	0.593

where k and c are machine-specific constants that define the capability of the machine. Table 6.3.2 shows the constant values of the machines on which this experiment was performed, determined from the empirical results obtained on each machine.

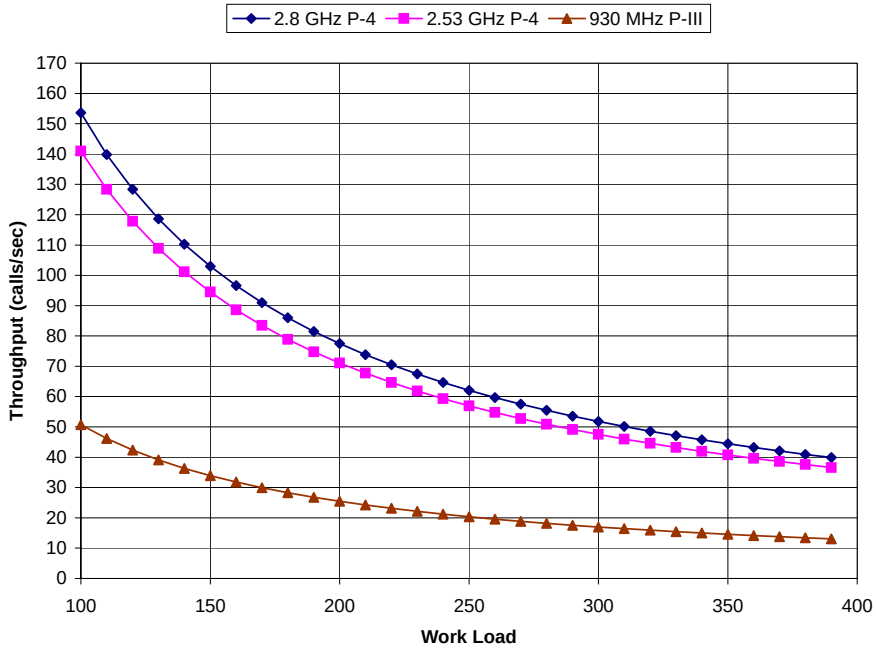

Figure 6.22: **Achievable Rates of the Worker Component vs. Workload for Different Hardware Configurations**

Analysis of results: We now consider developing and deploying an application as shown in Figure 6.23 with an illustrative workload of 150 repetitions. The controller should trigger the worker to perform as much work as possible by requesting as high a rate as possible without failing to sustain the configured rate. Assume the controller can be configured to run at any of the following discrete rates: 25 Hz, 75 Hz, 100 Hz, and 125 Hz. The following list discusses how and what information can be added and utilized in this example through CIAO's configuration capabilities at different lifecycle stages.

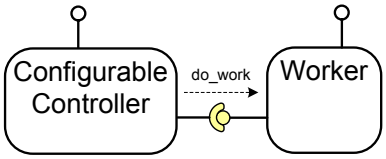

Figure 6.23: **Application with Possibly Variable Rates**

- **Component implementation packaging stage:** During this stage, the packaging tool should document the component behaviors targeted for later analysis. For example, the descriptors for the controller component should document the allowable rates at which it can run. The descriptors for the worker component should document the maximum achievable rates under both known hardware configurations, *i.e.*, 103 Hz using a 2.8 GHz machine and 95 Hz using a 2.53 GHz machine. Alternatively, the descriptor can record the formulas describing the maximum achievable rate as behavioral constraints of the component implementation.

- **Application assembly stage:** After composing the two components together into an application assembly, the assembly tool should review the behavioral constraints and create a new set of constraints for the assembly. In this example, it is obvious that in neither configuration can the controller run at 125 Hz. Since the goal of the application is to perform work at highest allowable rate, a developer (or modeling tool) can then set the rate information in the assembly descriptor to configure the controller at 100 Hz when using the 2.8 GHz machine, 75 Hz when using the 2.53 GHz machine, or 25 Hz when using the 930 MHz machine.

- **Application deployment stage:** At this stage, the developer (or modeling tool) can compare the actual characteristics of the deployment target (obtained, *e.g.*, through *a priori* analysis or run-time benchmarking) to the rate configurations established at the assembly stage, to select the highest feasible rate that can be sustained on each given platform subject to all other constraints, *e.g.*, invocation rate dependencies between components deployed across possibly heterogeneous processors.

As can be seen in this analysis, the addition of systemic information enables developers and modeling tools to verify, optimize, and flexibly reconfigure application assembly and deployment platform information at various stages of the development lifecycle. Although CIAO does not itself perform the analysis to do such verification, optimization, or reconfiguration, its rich capabilities to configure both functional and systemic aspects at

different phases of the system lifecycle are key enablers for these additional features. Compared to TAO-based applications, this added robustness and adaptability can not only ease the process of attaining correctness in a complex system, but also to improve the performance of the overall application when the narrower development cycle needed to configure (rather than to program) systemic aspects allows on-the-spot optimizations using CIAO.

6.4 Summary, Observations and Recommendations

The experiments performed in the dissertation show that CIAO adds only a small amount of overhead by comparing the performance of an equivalent CIAO application to one based on TAO. The proportion of overhead will diminish even more with the increase of the size of the payload of an operation. CIAO also does not degrade the predictability of applications by adversely affecting the jitter. The current implementation of CIAO, however, does demand more secondary storage and primary memory for running CIAO applications. The impact on extra footprint and storage space is expected to lessen as the implementation of CIAO evolves, *e.g.*, when the dependencies on unused libraries are removed.

This chapter also shows that CIAO's run-time support for real-time applications impose only a small amount of overhead to the overall performance and does not adversely affect the predictability. Moreover, CIAO's real-time extensions have been shown to enable the composition of real-time behaviors into an application flexibly and *effectively*. Because developers can now integrate real-time behaviors over the whole application end-to-end, these extensions can make developing, maintaining and validating large scale DRE applications easier.

Based on the results obtained in this section, we now offer the following observations and recommendations for developers of complex real-time systems:

Observation: The experiments performed in this chapter are modeled after existing TAO performance tests and the real-time validation tests for TAO's RT-CORBA implementation [52]. Comparing CIAO's test program implementations to their TAO-based counterparts, one striking difference is how easy it is to develop and modify CIAO-based tests. Developing TAO test programs requires writing new tests. That is, several specific programs are required to provide different tests of different configurations, as in the case of basic performance tests for TAO with and without RT-ORB. Conversely, CIAO requires only one application assembly but different configurations were achieved by using different standard

tools provided in CIAO, *i.e.*, by changing the deployment environment configuration to use a regular component server or a real-time component server.

The real-time validation tests provide an even more obvious contrast. TAO-based test programs requires elaborate design of test procedures, options, and configuration into the programs themselves and can only be run easily using a set of corresponding Perl scripts. In contrast, CIAO tests use only a limited number of simple component implementations running on a common run-time environment. Different test configurations can be selected by simply deploying an assembly with different combinations of application compositions and real-time behaviors. However, CIAO does require significantly more disk and memory storage space.

Recommendation: This observation shows that although CIAO does impose modest performance overhead, applications developed using CIAO actually can achieve superior performance much more easily than using TAO, as CIAO-based applications are much easier to reason about, maintain, modify and enhance. CIAO applications can therefore be optimized to achieve better performance with lower cost. Further work on CIAO is needed, however, to reduce CIAO's footprint so that CIAO can be applied to application domains where there are stringent memory limitations. It is also important to be able to make the CIAO implementation more robust and be able to compose CIAO internal modules based on application requirements. Reflective middleware techniques, such as dynamicTAO [28], are worth investigating.

Observation: The development paradigm provided by CCM allows developers to attach relevant metadata in each lifecycle phase. It is important to "bind" the information at the proper lifecycle phase to maintain as much flexibility as possible. Based on the example scenario in our experiment, we draw the following recommendations:

Recommendation:

- **Bind decisions early if possible.** Some information can be ignored after it is checked at a particular lifecycle phase. For example, after the component packaging phase *ensures* that each event sink has a corresponding executor, the application assembly and system deployment phases need not be concerned with that issue. This has the overall benefit of simplifying decisions made later in the system lifecycle.
- **(Re)bind decisions flexibly.** Other information, and the results of previous decisions that have relied on it, cross-cut several phases of the system lifecycle. For

example, after the sets of available event source and sink rates are ensured to match along component dependencies in the assembly phase, the resulting sets of rates and the component dependencies are still needed in the system deployment phase. This allows configuration of concerns that cross-cut the architectural boundaries of component, application, and system, as well as the configuration phase for each of these architectural levels.

- **QoS aspects tend to cross-cut functional boundaries.** Notice especially that QoS information is often refined in subsequent lifecycle phases after it is introduced. Functional information, on the other hand, tends to be more fixed once it is specified in a given phase. This reflects a natural point of difference between CIAO and conventional CCM in which functional information tends to compose in a more object-oriented manner, while the "locality of reference" of QoS decisions tends to be organized around aspect modularity that cross-cuts object and even component boundaries. CIAO is designed with the necessary refinement of QoS aspects in mind, and the understanding that decisions can improve with additional information as long as prior decisions can be kept flexible and revisited as needed. Conventional CCM on the other hand is designed more for functional properties that once specified remain stable for all subsequent composition stages.

Chapter 7

Conclusions and Future Research

This dissertation concentrated on examining and presenting techniques for composing systemic behaviors into component-based middleware, using CCM as an example component middleware and real-time behaviors as example systemic behaviors. Conventional component middleware has been shown to improve the software development process and reduce the time and cost to develop and maintain software system considerably in a wide range of application domains. Historically, component middleware is designed for enterprise applications and does not sufficiently support the kind of systemic behaviors pertinent to DRE applications to satisfy their stringent QoS requirements. To bring the benefit of component-based middleware frameworks into DRE application development, it is necessary to extend the component middleware flexibly to support the composition of systemic behaviors and supporting mechanisms into applications based on component middleware.

The focal theme of the work in this dissertation is enabling the composition of systemic behaviors and supporting mechanisms into component middleware so it is suitable for new domain of applications, such as DRE applications. An implementation of the CORBA Component Model (CCM) called the Component Integrated ACE ORB (CIAO) was developed by the work presented in this dissertation to provide the context of this research. RT-CORBA was used as the concrete example of systemic behaviors and supporting mechanisms that are of crucial importance to DRE applications. Although it can be made to work with other ORBs that support the RT-CORBA 1.0 specification [37], CIAO is currently built on top of TAO [24] which is a high-quality, freely available, open-source CORBA-compliant middleware platform with a complete implementation of the RT-CORBA 1.x specification. As of this writing, CIAO is being used in several research projects, as is described later in Section 7.2.

Composing real-time behaviors into component-based middleware applications requires the flexible extension of both the metadata used in the middleware to describe the kind of behaviors to be composed, and the middleware runtime mechanisms to support the composed behaviors. In extending the component metadata, this work investigated various aspects of composing systemic behaviors, specifically in granularity, scope, and timing, and analyzed their consequences for the overall development processes and application quality. In extending the middleware runtime platform, this dissertation focused on both the middleware configuration and composition of supporting mechanisms. Moreover, this dissertation investigated not only how to compose systemic behaviors into a system but also how to utilize mechanisms to extend metadata and to embed behavioral information for modeling tools to enhance the robustness of the application.

Finally, the implementations of the extensions presented in this dissertation along with the experimental results show that real-time behaviors can be effectively and flexibly composed into applications. This approach can greatly enhance the degree of reuse of component implementations and applications. It also enables modeling tools to ensure optimized configuration of applications can be achieved.

In summary, this dissertation has made the following contribution to research on composing systemic behaviors into component middleware:

1. It analyses and documents the proper binding points in various stages of the component development lifecycle for key real-time policies critical to DRE applications.

2. It reviews and documents granularities for binding QoS provisioning policies, and examines the impact of binding with different granularities.

3. It presents extensions to manage application assembly and deployment configurations to minimize inconsistency and to make analyzing deployment information easier.

4. It develops and documents solutions for managing differences in QoS management mechanisms when composing QoS policies with platform-specific QoS management mechanisms.

5. It conducts experiments to evaluate the cost and to demonstrate the effectiveness and benefits of adopting component-based middleware and making systemic behaviors into composable aspects.

7.1 Lessons Learned

Based on the experiences and observations from conducting the research presented in this dissertation, the following lessons are important to remember. First, the ability not only to separate systemic aspects but also to manage and maintain these aspects is very important for developing any non-trivial application. In CIAO, the CCM defined framework takes care of various aspects in developing, composing, and deploying applications. The real-time extensions defined in CIAO in turn augment that process by providing a framework for defining, composing and managing real-time aspects of DRE applications. Even with just a few examples and test components during the development of CIAO framework, we were able to come up with many different applications with an interesting degree of complexity, and deploy them easily among multiple machines.

This work has also shown that because the lifecycle of a CCM application has clearly defined development stages, it is important to identify the right stages where certain information should be added and then used toward improving the final deployed application. As shown in CIAO's support for deployment, a logical location defined in an application assembly file provides the right amount of information needed to document the intention of the application designer. This logical information later provides "hints" in the deployment phase for system deployer to realize the intention as an actual application deployment.

Similarly, it is important for component middleware to compose not only the application from components but also the runtime environment from modules supporting capabilities to control and manage different systemic aspects. Moreover, CIAO is a good research vehicle within which to explore meta-programming and aspect-oriented programming techniques, providing a robust mechanism for injecting active software functionality into a system to provide dynamic systemic behaviors transparently. The XML format adopted by the CCM specification makes it easy to extend the metadata to support composition of new systemic behaviors. The same can also be said for extending the metadata to document key component features as constraints. Other higher level tools can then reason about and synthesize new constraints when components are composed into applications.

7.2 Future Research

As CORBA has matured over the years, more and more standard mechanisms have been added to address QoS concerns for a variety of application domains. CORBA has gained

acceptance in a wide range of domains and is being applied to many projects worldwide, including telecommunications, aerospace, financial services, process control, scientific computing, and distributed interactive simulations. CIAO, the resulting CCM implementation from the work described in this dissertation looks to bring the many advantages of component-based middleware into application domains that were not targeted in the original CCM specification and have different QoS requirements than the original target domain, *i.e.,* enterprise applications.

CIAO provides the first real-time-enabled CCM framework targeting DRE applications. At this writing, CIAO has been released as open-source software and is also freely available on the web. CIAO is currently being used in several research projects:

- **Static CCM Applications:** This project conducted by Washington University seeks to enable the composition of CCM components into statically linked applications. Statically linked programs are often required in many embedded applications due to inherent constraints such as the need to bound system start up time, or the lack of secondary storage system. However, this requirement prevents the adoption of CCM implementations which require dynamic loading of component implementations into primary memory.

  Enabling the generation of statically linked program images from the CCM assembly descriptors and component implementations brings the benefits of CIAO to many more embedded system developers. Applications can easily be composed, prototyped, and verified in traditional deployment environments but can still be converted to statically linked program images for embedded environment.

- **Integration of Event Channels:** CIAO's existing implementation of event source/sink connections only supports a trivial implementation using synchronous method invocations. Researchers at Vanderbilt University are looking into integrating an Event Channel implementation to support different event delivery strategies.

- **Dynamic QoS Adaptation:** Currently, CIAO only supports statically composing QoS behaviors that can be predetermined before an application starts running [81]. Researchers at BBN Technologies are exploring composing CCM components called Qoskets, which implement dynamic QoS provisioning behaviors, into an application. A modeling tool can assist to create new application assemblies that have Qoskets woven in.

Dynamic Scheduling Service: CIAO's real-time extensions to the CCM framework concentrate on fixed-priority scheduling as defined in the RT-CORBA 1.0 specification. However, certain applications require dynamic end-to-end management of priorities to ensure better resource utilization while satisfying the system QoS requirements. It is therefore important to integrate dynamic scheduling frameworks, such as Kokyu [18], into CCM middleware to allow applications to compose dynamic scheduling strategies into applications.

Robust Support for Adding New Systemic Behaviors: The real-time behaviors covered by CIAO's extensions are sufficient for many DRE applications. However, other DRE applications require more sophisticated systemic behaviors. Supporting a new systemic behavior usually involves 1) describing the required behavior in various CCM descriptors, and 2) integrating mechanisms supporting the behavior into the run-time environment. It is important to be able to compose both the XML handlers and the supporting mechanisms into the CCM framework to maintain flexibility. This requires an XML handler composition mechanism that can plug in XML handlers for different XML extensions.

Providing Fine-grained Control for Systemic Behaviors: Existing CIAO implementation supports the composition of systemic behaviors for containers. This affects the behaviors of a component instance. However, some applications require the configuration of systemic behaviors only within an interface or even a single operation of a component. To use CIAO to develop such applications, it is important to specify and control systemic behaviors for these finer-grained entities.

Integration of Dynamic QoS Management: As was described earlier in this section, BBN technologies is currently looking into composing dynamic QoS management behaviors, *e.g.*, qoskets, by implementing these behaviors as interceptor CCM components and weaving these components into application assemblies. This approach not only requires the modification of application assemblies, which may make them hard to maintain, but it also may impose too much performance and footprint overhead.

Another alternative would be to apply the meta-programming techniques [84] directly in the component middleware. Dynamic QoS modules could be generated by a smart CIDL compiler into the servant glue code. The CIDL compiler could alternatively generate servants with integrated meta-programming hooks as smart proxies and skeletons. Systemic modules could then be composed into the application assemblies as "add-ons"

without modifying the application assemblies directly, and therefore would allow them to be maintained easily by the original designers.

Alleviating the Complexity of Managing Large-scale DRE Software: A component development paradigm can reduce lifecycle costs and time-to-market by enabling application developers to assemble and deploy DRE applications by selecting a set of compatible common-off-the-shelf (COTS) and custom-developed components. To compose an application successfully requires that these components have compatible interfaces, semantics, and protocols, which makes selecting and developing a compatible set of application components a daunting task [31]. This problem is further exacerbated by the existence of myriad strategies for configuring and deploying the underlying middleware to leverage special hardware and software features.

Moreover, composing systemic behaviors end-to-end often pervades an entire application which only exacerbates the overall complexity. Consequently, application developers have to spend non-trivial amounts of time debugging problems associated with the selection of incompatible strategies and components. What is therefore needed is an integrated set of software development processes, platforms, and tools that can (1) select a suitable set of middleware and application components, (2) analyze, synthesize, and validate the component configurations, (3) assemble and deploy groups of related components to their appropriate execution platforms, and (4) dynamically adjust and reconfigure the system as operational conditions change, to maintain the required QoS properties of DRE applications. It is fair to say that extended component middleware provides the mechanisms to make systemic behaviors into modular aspects that can be realized and synthesized.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages, 3rd Edition*. Addison Wesley Longmain, March 2001.

[3] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

[4] Bertrand Meyer. The Significance of Components. `http://www.sdmagazine.com/documents/s=752/sdm9911k/9911k.htm`, November 1999. in Beyond Objects.

[5] Don Box. *Essential COM*. Addison-Wesley, Reading, MA, 1998.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York, 1996.

[7] Cemal Yilmaz and Adam Porter and Douglas C. Schmidt. Distributed Continuous Quality Assurance: The Skoll Project. In *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Portland, Oregon, May 2003. IEEE/ACM.

[8] Denis Conan, Erik Putrycz, Nicolas Farcet, and Miguel DeMiguel. Integration of Non-Functional Properties in Containers. *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*, 2001.

[9] Joseph K. Cross and Douglas C. Schmidt. Applying the Quality Connector Pattern to Optimize Distributed Real-time and Embedded Middleware. In Fethi Rabhi and

Sergei Gorlatch, editors, *Patterns and Skeletons for Distributed and Parallel Computing*. Springer Verlag, 2002.

[10] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[11] D.W. Davies, E. Holler, E.D. Jensen, S.R. Kimbleton, B.W. Lampson, G. LeLann, K.J. Thurber, and R.W. Watson. *Distributed Systems- Architecture and Implementation – An Advanced Course*. Springer-Verlag, 1981.

[12] Miguel A. de Miguel. QoS-Aware Component Frameworks. In *The* 10^{th} *International Workshop on Quality of Service (IWQoS 2002)*, Miami Beach, Florida, May 2002.

[13] Douglas C. Schmidt and Frank Buschmann. Patterns, Frameworks, and Middleware: Their Synergistic Relationships. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, Portland, Oregon, May 2003. IEEE/ACM.

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

[15] FOKUS. Qedo Project Homepage. `http://qedo.berlios.de/`.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[17] Chris Gill, Douglas C. Schmidt, and Ron Cytron. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), January 2003.

[18] Christopher D. Gill, Ron Cytron, and Douglas C. Schmidt. Middleware Scheduling Optimization Techniques for Distributed Real-Time and Embedded Systems. In *Proceedings of the* 7^{th} *Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002. IEEE.

[19] Aniruddha Gokhale, Balachandran Natarajan, Douglas C. Schmidt, Andrey Nechypurenko, Jeff Gray, Nanbor Wang, Sandeep Neema, Ted Bapty, and Jeff Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop*

on Generative Techniques in the Context of Model Driven Architecture, Seattle, WA, November 2002. ACM.

[20] Aniruddha Gokhale, Douglas C. Schmidt, Balachandra Natarajan, and Nanbor 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), October 2002.

[21] Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang. Model Driven Middleware. In Qusay Mahmoud, editor, *Middleware for Communications*. Wiley and Sons, New York, 2003.

[22] Pradeep Gore, Ron K. Cytron, Douglas C. Schmidt, and Carlos O'Ryan. Designing and Optimizing a Scalable CORBA Notification Service. In *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, pages 196–204, Snowbird, Utah, June 2001. ACM SIGPLAN.

[23] John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, and Venkatesh Prasad. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[24] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

[25] Prashant Jain and Douglas C. Schmidt. Dynamically Configuring Communication Services with the Service Configurator Pattern. *C++ Report*, 9(5), June 1997.

[26] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[27] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[28] Fabio Kon and Roy H. Campbell. Supporting Automatic Configuration of Component-Based Distributed Systems. In *Proceedings of the* 5^{th} *Conference on*

Object-Oriented Technologies and Systems, pages 175–178, San Diego, CA, May 1999. USENIX.

[29] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The Case for Reflective Middleware. *Communications of the ACM*, 45(6):33–38, June 2002.

[30] Tao Lu, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt. CoSMIC: An MDA Tool suite for Application Deployment and Configuration. In *Proceedings of the OOPSLA 2003 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA, October 2003. ACM.

[31] Luis Iribarne and José M. Troya and Antonio Vallecillo. Selecting Software Components with Multiple Interfaces. In *Proceedings of the* 28^{th} *Euromicro Conference (EUROMICRO'02)*, pages 26–32, Dortmund, Germany, September 2002. IEEE.

[32] Floyd Marinescu and Ed Roman. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, New York, 2002.

[33] Phil Mesnier. A Shared Library Footprint Reduction Tool. In *Proceedings of the Second Annual TAO Workshop*, Arlington, VA, July 2002.

[34] Microsoft. .NET Web Services Platform. `www.microsoft.com/net`.

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

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

[37] Object Management Group. *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 edition, February 1999.

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

[39] Object Management Group. *Real-time CORBA Specification*, OMG Document formal/02-08-02 edition, August 2002.

[40] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, December 2002.

[41] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.6.1 edition, May 2002.

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

[43] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.

[44] Object Management Group. *Qualify of Service for CORBA Component RFP*, OMG Document mars/03-06-12 edition, June 2003.

[45] Object Management Group. *Streams for CORBA Component RFP*, OMG Document mars/03-06-11 edition, June 2003.

[46] Object Management Group. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Joint Revised Submission*, OMG Document realtime/03-05-02 edition, May 2003.

[47] Olaf Spinczyk and Andreas Gal and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, February 2002.

[48] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell Noseworthy. Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations. *International Journal of Computer Systems Science and Engineering*, 17(2), March 2002.

[49] J. Postel. User Datagram Protocol. *Network Information Center RFC 768*, pages 1–3, August 1980.

[50] J. Postel. Transmission Control Protocol. *Network Information Center RFC 793*, pages 1–85, September 1981.

[51] Irfan Pyarali and Douglas C. Schmidt. An Overview of the CORBA Portable Object Adapter. *ACM StandardView*, 6(1), March 1998.

[52] Irfan Pyarali, Douglas C. Schmidt, and Ron Cytron. Achieving End-to-End Predictability of the TAO Real-time CORBA ORB. In 8^{th} *IEEE Real-Time Technology and Applications Symposium*, San Jose, September 2002. IEEE.

[53] Irfan Pyarali, Douglas C. Schmidt, and Ron Cytron. Techniques for Enhancing Real-time CORBA Quality of Service. *IEEE Proceedings Special Issue on Real-time Systems*, 91(7), July 2003.

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

[55] Ragunathan Rajkumar, Chen Lee, John P. Lehoczky, and Daniel P. Siewiorek. Practical Solutions for QoS-based Resource Allocation Problems. In *IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998. IEEE.

[56] Tom Ritter, Marc Born, Thomas Unterschütz, and Torben Weis. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, Honolulu, HW, January 2003. HICSS.

[57] Ward Rosenberry, David Kenney, and Gerry Fischer. *Understanding DCE*. O'Reilly and Associates, Inc., 1992.

[58] SAX Project. Simple API for XML. www.saxproject.org, 2002.

[59] Richard Schantz, Joseph Loyall, Michael Atighetchi, and Partha Pal. Packaging Quality of Service Control Behaviors for Reuse. In *Proceedings of the* 5^{th} *IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, Crystal City, VA, April/May 2002. IEEE/IFIP.

[60] Richard E. Schantz and Douglas C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering*. Wiley & Sons, New York, 2002.

[61] Richard E. Schantz, Robert H. Thomas, and Girome Bono. The Architecture of the Cronus Distributed Operating System. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 250–259, Cambridge, MA, May 1986. IEEE.

[62] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts, 2002.

[63] Douglas C. Schmidt, Rick Schantz, Mike Masters, Joseph Cross, David Sharp, and Lou DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. *CrossTalk*, November 2001.

[64] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[65] Douglas C. Schmidt and Steve Vinoski. An Overview of the CORBA Messaging Quality of Service Framework. *C++ Report*, 12(3), March 2000.

[66] Douglas C. Schmidt and Steve Vinoski. Real-time CORBA, Part 1: Motivation and Overview. *C/C++ Users Journal*, October 2001.

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

[68] David C. Sharp. Avionics Product Line Software Architecture Flow Policies. In *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, October 1999.

[69] David C. Sharp and Wendy C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

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

[71] Venkita Subramonian and Christopher Gill. A Generative Programming Framework for Adaptive Middleware. In *Hawaii International Conference on System Sciences, Software Technology Track, Adaptive and Evolvable Software Systems Minitrack, HICSS 2003*, Honolulu, HW, January 2003. HICSS.

[72] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.

[73] Sun Microsystems. Java^{TM} 2 Platform Enterprise Edition. java.sun.com/j2ee/index.html, 2001.

[74] Sun Microsystems, Inc. *Java Remote Method Invocation Specification (RMI)*, October 1998.

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

[76] Bruce Trask. A Case Study on the Application of CORBA Products and Concepts to an Actual Real-Time Embedded System. In *OMG's First Workshop On Real-Time & Embedded Distributed Object Computing*, Washington, D.C., July 2000. Object Management Group.

[77] France Universite des Sciences et Technologies de Lille. The OpenCCM Platform. `corbaweb.lifl.fr/OpenCCM/`, 2003.

[78] Markus Volter, Alexander Schmid, and Eberhard Wolff. *Server Component Patterns: Component Infrastructures Illustrated with EJB*. Wiley Series in Software Design Patterns, West Sussex, England, 2002.

[79] Nanbor Wang and Christopher Gill. Improving Real-Time System Configuration via a QoS-aware CORBA Component Model. In *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2003*, Honolulu, HW, January 2003. HICSS.

[80] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz. Total Quality of Service Provisioning in Middleware and Applications. *The Journal of Microprocessors and Microsystems*, 27(2):45–54, mar 2003.

[81] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill. QoS-enabled Middleware. In Qusay Mahmoud, editor, *Middleware for Communications*. Wiley and Sons, New York, 2003.

[82] Nanbor Wang, Douglas C. Schmidt, Michael Kircher, and Kirthika Parameswaran. Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications. *IEEE Distributed Systems Online*, 2(5), July 2001.

[83] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. An Overview of the CORBA Component Model. In George Heineman and Bill Councill, editors, *Component-Based Software Engineering*. Addison-Wesley, Reading, Massachusetts, 2000.

[84] Nanbor Wang, Douglas C. Schmidt, Ossama Othman, and Kirthika Parameswaran. Evaluating Meta-Programming Mechanisms for ORB Middleware. *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, 39(10):102–113, October 2001.

[85] Nanbor Wang, Douglas C. Schmidt, and Steve Vinoski. Collocation Optimizations for CORBA. *C++ Report*, 11(10):47–52, November/December 1999.

[86] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. *USENIX Computing Systems*, 9(4), November/December 1996.

[87] John A. Zinky, David E. Bakken, and Richard Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.

Vita

Nanbor Wang

Date of Birth June 13, 1964

Place of Birth Taipei, Taiwan

Degrees B.S. Mineral and Petroleum Engineering, May 1986,
M.S. Mineral and Petroleum Engineering, May 1988,
from National Cheng-Kung University, Tainan, Taiwan
M.S. Computer Science, May 1997,
from Washington University in Saint Louis.

Book Chapters

Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill, "QoS-enabled Middleware", in *Middleware for Communications* (Qusay Mahmoud, ed.), New York, Wiley and Sons, 2003.

Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, Jeff Gray, and Nanbor Wang, "Model Driven Middleware", in *Middleware for Communications* (Qusay Mahmoud, ed.), New York, Wiley and Sons, 2003.

Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan, "An Overview of the CORBA Component Model", in *Component-Based Software Engineering* (G. Heineman and B. Councill, eds.), Reading, Massachusetts: Addison-Wesley, 2000.

Journal Publications

Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall and Richard E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications", *Microprocessors and Microsystems, special issue on Middleware Solutions for*

QoS-enabled Multimedia Provisioning over the Internet (Paolo Bellavista ed.), vol. 27, no. 2, pp. 45-54, March 2003.

Aniruddha Gokhale, Douglas C. Schmidt, Balachandra Natarajan and Nanbor 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* (Ali Arsanjani ed.), vol. 45, no. 10, October 2002.

Nanbor Wang, Douglas C. Schmidt, Michael Kircher, and Kirthika Parameswaran, "Adaptive and Reflective Middleware for QoS-Enabled CCM Applications", *IEEE Distributed Systems Online*, vol. 2, July 2001.

Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo and Aniruddha Gokale, "Using Principle Patterns to Optimize Real-Time ORBs", *IEEE Concurrency Magazine*, vol. 8, no. 1, 2000.

Nanbor Wang, Douglas C. Schmidt, Ossama Othman, and Kirthika Parameswaran, "Evaluating Meta-Programming Mechanisms for ORB Middleware", *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies* (Bill Opdyke and Algirdas Pakstas, eds.), October 2000.

Conference Publications

Arvind S. Krishna, Nanbor Wang, Balachandran Natarajan, Anniruddha Gokhale, Douglas C. Schmidt and Gautam Thaker, "CCMPerf: A Benchmarking Tool for CORBA Component Model Implementations", *Proceedings of the 10th Real-time Technology and Application Symposium (RTAS '04)*, Toronto, Canada, May 2004.

Nanbor Wang and Christopher D. Gill, "Improving Real-Time System Configuration via a QoS-aware CORBA Component Model", Hawaii International Conference on System Sciences (HICSS '04), Software Technology Track, Distributed Object

and Component-based Software Systems Minitrack, HICSS 2004, Honolulu, Hawaii, January, 2004.

Anirudda Gokhale, Balachandran Natarjan, Douglas C. Schmidt, Nanbor Wang, Sandeep Neema, Ted Bapty, Jeff Parsons, Jeff Gray and Andrey Nechypurenko, "CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications", *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, November, 2002, ACM.

Nanbor Wang, Kirthika Parameswaran, and Douglas C. Schmidt, "The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware", in *Proceedings of the* 6^th *Conference on Object-Oriented Technologies and Systems*, (San Antonio, TX), pp. 103-118, USENIX, Jan/Feb 2000.

Nanbor Wang, Douglas C. Schmidt, Kirthika Parameswaran, and Michael Kircher, "Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation", in *the* 24^th *Computer Software and Applications Conference*, (Taipei, Taiwan), IEEE, October 2000.

Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo and Aniruddha Gokale, "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.

Workshop Publications

Svetlana G. Shasharina, Nanbor Wang and John R. Cary, "Grid Service for Visualization and Analysis of Remote Fusion Data" , in Proceedings of Challenges of Large Applications in Distributed Environments (CLADE), Honolulu, Hawaii, June, 2004.

Nanbor Wang, Craig Rodrigues, and Chris Gill, "A Qos-aware CORBA Component Model for Distributed, Real-time, and Embedded System Development" in *OMG Workshop On Embedded and Real-Time Distributed Object Systems*, (Washington D.C.), OMG, July, 2003.

Krishnakumar Balasubramanian, Nanbor Wang, Christopher D. Gill, and Douglas C. Schmidt, "Towards Composable Distributed Real-time and Embedded Software", *Proceedings of the 8^th Workshop on Object-oriented Real-time Dependable Systems*, Guadalajara, Mexico, January, 2003, IEEE.

Nanbor Wang, Krishnakumar Balasubramanian, and Chris Gill, "Towards a Real-time CORBA Component Model" in *OMG Workshop On Embedded and Real-Time Distributed Object Systems*, (Washington D.C.), OMG, July, 2002.

Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, and Nanbor Wang, "Applying the Model Driven Architecture to Distributed Real-time and Embedded Applications" in *OMG Workshop On Embedded and Real-Time Distributed Object Systems*, (Washington D.C.), OMG, July, 2002.

Nanbor Wang, Douglas C. Schmidt, Kirthika Parameswaran, and Michael Kircher, "Towards a Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Applications", in *Reflective Middleware Workshop*, ACM/IFIP, April 2000.

Nanbor Wang, Douglas C. Schmidt, and David Levine, "Optimizing the CORBA Component Model for High-Performance and Real-Time Applications", in *'Work-in-Progress' session at the Middleware 2000 Conference*, ACM/IFIP, April 2000.

Trade-Journal Publications

Nanbor Wang, Douglas C. Schmidt, and Steve Vinoski, "Collocation Optimizations for CORBA", *C++ Report*, vol. 11, pp. 47-52, November/December 1999.