Middleware for Communications

Contents

Preface **2**

1 QoS-enabled Middleware **1**
1.1 Introduction 2
1.1.1 Emerging Trends 2
1.1.2 Key Technical Challenges and Solution Approaches 3
1.1.3 Chapter Organization 5
1.2 The Evolution of Middleware 5
1.2.1 Overview of Middleware 6
1.2.2 Limitations of Conventional Middleware 7
1.3 Component Middleware: A Powerful Approach to Building DRE Applications 8
1.3.1 Overview of Component Middleware and the CORBA Component Model 9
1.3.2 Limitations with Conventional Component Middleware for Large-scale DRE Systems 12
1.4 QoS Provisioning and Enforcement with CIAO and QuO Qoskets . . . 14
1.4.1 Static Qos Provisioning via QoS-enabled Component Middleware and CIAO 16
1.4.2 Dynamic QoS Provisioning via QuO Adaptive Middleware and Qoskets 20
1.4.3 Integrated QoS provisioning via CIAO and Qoskets 24
1.5 Related Work 26
1.6 Concluding Remarks 28

Bibliography **30**

Preface

1

QoS-enabled Middleware

Nanbor Wang and Christopher D. Gill
Dept. of Computer Science and Engineering
Washington University
One Brookings Drive
St. Louis, MO 63130

Douglas C. Schmidt, Aniruddha Gokhale, Balachandran Natarajan,
Institute for Software Integrated Systems
Vanderbilt University
Box 1829, Station B
Nashville, TN 37203

Joseph P. Loyall, Richard E. Schantz, and Craig Rodrigues
BBN Technologies
10 Moulton Street
Cambridge, MA 02138

Middleware for Communications. Edited by Qusay H. Mahmoud

1.1 Introduction

1.1.1 Emerging Trends

Commercial-off-the-shelf (COTS) middleware technologies, such as The Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) (Obj 2002c), Sun's Java RMI (Wollrath et al. 1996), and Microsoft's COM+ (Morgenthal 1999), 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. While these middleware technologies have historically been applied to *enterprise applications* (Gokhale et al. 2002), such as online catalog and reservation systems, bank asset management systems, and management planning systems, over 99% of all microprocessors are now used for distributed real-time and embedded (DRE) systems (Alan Burns and Andy Wellings 2001) that control processes and devices in physical, chemical, biological, or defense industries.

These types of DRE applications have distinctly different characteristics 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 QoS requirements can lead to catastrophic consequences. Middleware that can satisfy stringent quality of service (QoS) requirements, such as predictability, latency, efficiency, scalability, dependability, and security, is therefore increasingly being applied to DRE application development. Regardless of the domain in which middleware is applied, however, its goal is to help expedite the software process by (1) making it easier to integrate parts together and (2) shielding developers from many inherent and accidental complexities, such as platform and language heterogeneity, resource management, and fault tolerance.

Component middleware (Szyperski 1998) 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.

Examples of COTS component middleware include the CORBA Component Model (CCM) (Obj 2002a), Java 2 Enterprise Edition (J2EE) (Sun Microsystems 2001), and the Component Object Model (COM) (Box 1998), each of which uses different APIs, protocols, and component models.

As the use of component middleware becomes more pervasive, DRE applications are increasingly 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 (Sharp 1998, 1999) that help aircraft pilots plan and navigate through their

routes. Often, these applications are combined further with other DRE applications to create "systems of systems," which we henceforth refer to as "large-scale DRE systems."

The *military command and control systems* shown in Figure 1.1 exemplifies such a large-scale DRE system, where information is gathered and assimilated from a diverse collection of devices (such as unmanned aerial vehicles and wearable computers), before being presented to higher level command and control functions that analyze the information and coordinate the deployment of available forces and weaponry. In the example, both the reconnaissance information such as types and locations

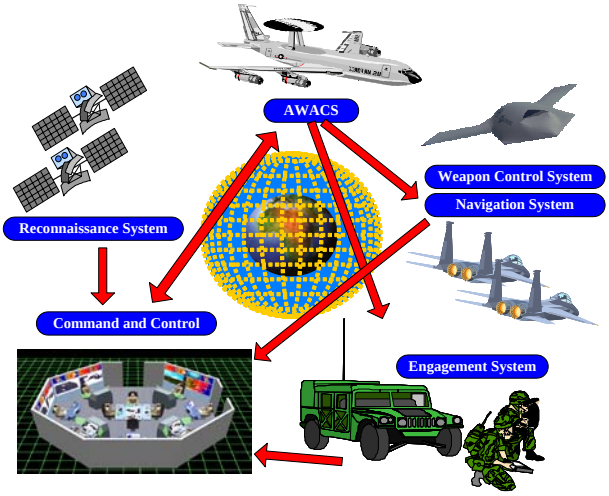

Figure 1.1 **An Example Large-scale DRE System**

of threats, and the tactical decisions such as retasking and redeployment, must be communicated within specified timing constraints across system boundaries to all involved entities.

1.1.2 Key Technical Challenges and Solution Approaches

The following key technical challenges arise when developing and deploying applications in the kinds of large-scale DRE systems environments outlined in Section 1.1.1:
1. Satisfying multiple QoS requirements in real-time. Most DRE applications have stringent QoS requirements that must be satisfied simultaneously in real-time. Example QoS requirements include processor allocation and network latency, jitter, and bandwidth. To ensure that DRE applications can achieve their QoS requirements, various types of *QoS provisioning* must be performed to allocate and manage system computing and communication resources end-to-end. QoS provisioning can be performed in the following ways:

- *Statically*, where ensuring adequate resources required to support a particular degree of QoS is pre-configured into an application. Examples of static QoS

provisioning include task prioritization and communication bandwidth reservation. Section 1.4.1 describes a range of QoS resources that can be provisioned statically.

- *Dynamically*, where the resources required are determined and adjusted based on the runtime system status. Examples of dynamic QoS provisioning include runtime reallocation re-prioritization to handle bursty CPU load, primary, secondary and remote storage access, and competing network traffic demands. Section 1.4.2 describes a range of QoS resources that can be provisioned dynamically.

QoS provisioning in large-scale DRE systems cross-cuts 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 enforcing the stringent, multiple and simultaneous QoS requirements of DRE applications end-to-end . 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 (Rajkumar et al. 1998) to meet those end-to-end real-time QoS requirements of DRE applications.

2. Alleviating the complexity of composing and managing large-scale DRE software. To reduce lifecycle costs and time-to-market, application developers today largely assemble and deploy DRE applications by manually 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 (Luis Iribarne and José M. Troya and Antonio Vallecillo 2002). This problem is further compounded by the existence of myriad strategies for configuring and deploying the underlying middleware to leverage special hardware and software features.

Moreover, DRE application demands for QoS provisioning (which in turn require end-to-end enforcement and often pervade an entire application) only exacerbate the complexity. Consequently, application developers 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.

This chapter provides the following two contributions to research and development efforts that address the challenges described above:

- We illustrate how enhancements to standard component middleware can simplify the development of DRE applications by composing QoS provisioning policies statically with applications. Our discussion focuses on a QoS-enabled

enhancement of standard CCM (Obj 2002a) called the *Component-Integrated ACE ORB* (CIAO), which is being developed at Washington University in St. Louis and the Institute for Software Integrated Systems (ISIS) at Vanderbilt University.

- We describe how dynamic QoS provisioning and adaptation can be addressed using middleware capabilities called *Qoskets*, which are enhancements to the Quality Objects (QuO) (Zinky et al. 1997) middleware developed by BBN Technologies. Our discussion focuses on how Qoskets can be combined with CIAO to compose adaptive QoS assurance into DRE applications dynamically. In particular, Qoskets manage modular QoS *aspects*, which can be combined with CIAO and woven to create an integrated QoS-enabled component model.

A companion chapter in this book (Gokhale et al. 2003) on Model Driven Middleware discusses how QoS-enabled component middleware can be combined with Model Driven Architecture (MDA) tools to rapidly develop, generate, assemble, and deploy flexible DRE applications that can be tailored readily to meet the needs of multiple simultaneous QoS requirements.

All the material presented in this chapter is based on the CIAO and QuO middleware, which can be downloaded in open-source format from `deuce.doc.wustl.edu/Download.html` and `quo.bbn.com/quorelease.html`, respectively. This middleware is being applied to many projects worldwide in a wide range of domains, including telecommunications, aerospace, financial services, process control, scientific computing, and distributed interactive simulations.

1.1.3 Chapter Organization

The remainder of this chapter is organized as follows: Section 1.2 gives a brief overview of middleware technologies and describes limitations of conventional object-based middleware technologies; Section 1.3 describes how component middleware addresses key limitations of object-oriented middleware and then explains how current component middleware does not yet support DRE application development effectively; Section 1.4 illustrates how the CIAO and QuO middleware expands the capabilities of conventional component middleware to facilitate static and dynamic QoS provisioning and enforcement for DRE applications; Section 1.5 compares our work on CIAO and Qoskets with related research; and Section 1.6 presents concluding remarks.

1.2 The Evolution of Middleware

In the early days of computing, 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, 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.

- **Technological advances** – With the advent of software development technologies, such as object-oriented programming languages and distributed object computing technologies, it has become easier to develop software with more capabilities and features.

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 Douglas C. Schmidt and Frank Buschmann (2003). Major modern software development paradigms all aim to achieve this common goal but differ in the type(s) and granularity of building blocks that form the core of each paradigm. The development and the evolution of middleware technologies also follow the similar goal to capture and reuse design information learned in the past, within various layers of software.

This section provides an overview of middleware and describes the limitations of conventional distributed object computing (DOC) middleware, which has been the dominant form of middleware during the 1990's. Section 1.3 then presents an overview of how component middleware overcomes the limitations of conventional DOC middleware.

1.2.1 Overview of Middleware

Middleware is reusable software that resides between applications and the underlying operating systems, network protocol stacks, and hardware (Schantz and Schmidt 2002). Middleware's primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure 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 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 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 them manually for each use.
- Provide a wide array of off-the-shelf developer-oriented services, such as transactional logging and security, 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 (Cemal Yilmaz and Adam Porter and Douglas C. Schmidt 2003).
- 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 (Postel 1980, 1981), interprocess communication and message passing architectures (Davies et al. 1981), micro-kernel architectures (Accetta et al. 1986), and Sun's Remote Procedure Call (RPC) model (Sun Microsystems 1988). The next generation of advances included OSF's Distributed Computing Environment (DCE) (Rosenberry et al. 1992), CORBA (Obj 2002c), and DCOM (Box 1998). More recently, middleware technologies have evolved to support DRE applications (*e.g.*, Real-time CORBA (Obj 2002b)), as well as to provide higher-level abstractions, such as component models (*e.g.*, CCM (Obj 2002a), J2EE (Sun Microsystems 2001)) and model driven middleware (Gokhale et al. 2003).

The success of middleware technologies has added the middleware 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.

1.2.2 Limitations of Conventional Middleware

Section 1.2.1 briefly described the evolution of middleware over the past 20+ years. One of the watershed events during this period was the emergence of distributed object computing (DOC) middleware in the late 1980's/early 1990s (Schantz et al. 1986). 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 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 (Buschmann et al. 1996; Gamma et al. 1995; Schmidt et al. 2000). 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) in the CORBA 2.x specification (Obj 2002d) defines an advanced DOC middleware standard for building portable dis-

tributed applications. The CORBA 2.x specification focuses on *interfaces*, which are essentially contracts between clients and servers that define how clients *view* and *access* object services provided by a server. Despite its advanced capabilities, however, the CORBA 2.x standard has the following limitations (Wang et al. 2000):

1. 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 is extra work that can yield brittle and non-reusable implementations.

2. Lack of generic server standards. CORBA 2.x does not specify a generic server framework to perform common server configuration work, including initializing a server and its QoS policies, providing common services (such as notification or naming services), and managing the runtime environment of each component. Although CORBA 2.x standardized 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) the ORB and object implementations interact. 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.

3. Lack of software configuration and deployment standards. There is no standard way to distribute and start up object implementations remotely in CORBA 2.x specifications. 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 are often 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 results in systems that are harder to maintain and software component implementations that are much harder to reuse.

1.3 Component Middleware: A Powerful Approach to Building DRE Applications

This section presents an overview of component middleware and the CORBA Component Model. It then discusses how conventional component middleware lacks support for the key QoS provisioning needs of DRE applications.

1.3.1 Overview of Component Middleware and the CORBA Component Model

Component middleware (Szyperski 1998) is a class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly. Recently, component middleware has evolved to address the limitations of DOC middleware described in Section 1.2.2 by

- Creating a virtual boundary around larger application component implementations that interact with each other only through well-defined interfaces,
- Defining standard container mechanisms needed to execute components in generic component servers, and
- Specifying the infrastructure to assemble, package, and deploy components throughout a distributed environment.

The CORBA Component Model (CCM) (Obj 2002a) is a current 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 component implementations. 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 application server runtime environments stipulated by the CCM specification. Components are larger building blocks than objects, with more of their interactions managed to simplify and automate key aspects of construction, composition, and configuration into applications.

A *component* is an implementation entity that exposes a set of *ports*, which are named interfaces and connection points that components use to collaborate with each other. Ports include the following interfaces and connection points shown in Figure 1.2:

- **Facets**, which define a named interface that services method invocations from other components synchronously.
- **Receptacles**, which provide named connection points to synchronous facets provided by other components.
- **Event sources/sinks**, which indicate a willingness to exchange event messages with other components asynchronously.

Components can also have *attributes* that specify named parameters that can be configured later via metadata specified in component property files.

Figure 1.3 shows the server-side view of the runtime architecture of the CCM model. A *container* provides the server runtime environment for component implementations called *executors*. It contains various pre-defined hooks and operations that give components access to strategies and services, such as persistence, event notification, transaction, replication, load balancing, and security. Each container defines a collection of runtime strategies and policies, such as an event delivery strategy and component usage categories, and is responsible for initializing and providing runtime

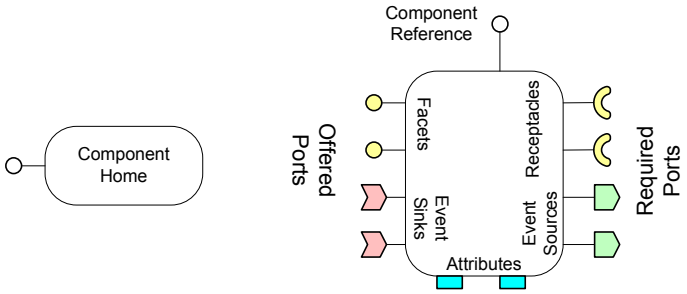

Figure 1.2 **Client View of CCM Components**

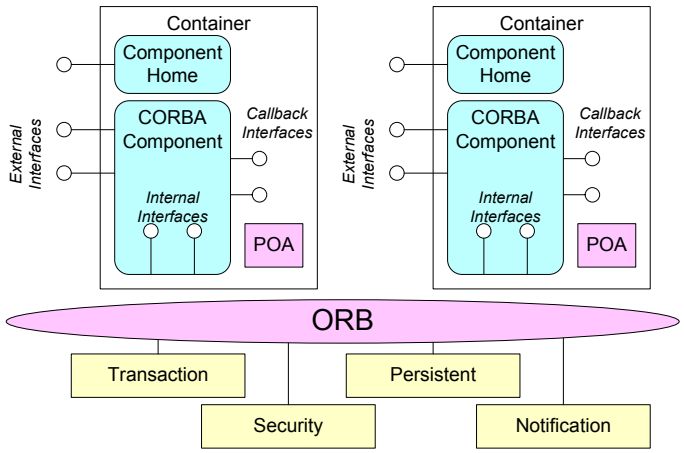

Figure 1.3 **Overview of the CCM Run-time Architecture**

contexts for the managed components. Component implementations have associated metadata, written in XML, that specify the required container strategies and policies.

In addition to the building blocks outlined above, the CCM specification also standardizes various aspects of stages in the application development lifecycle, notably component implementation, packaging, assembly, and deployment as shown in Figure 1.4, where each stage of the lifecycle adds information pertaining to these aspects. The numbers in the discussion below correspond to the labels in Figure 1.4. The CCM Component Implementation Framework (CIF) (1) automatically generates

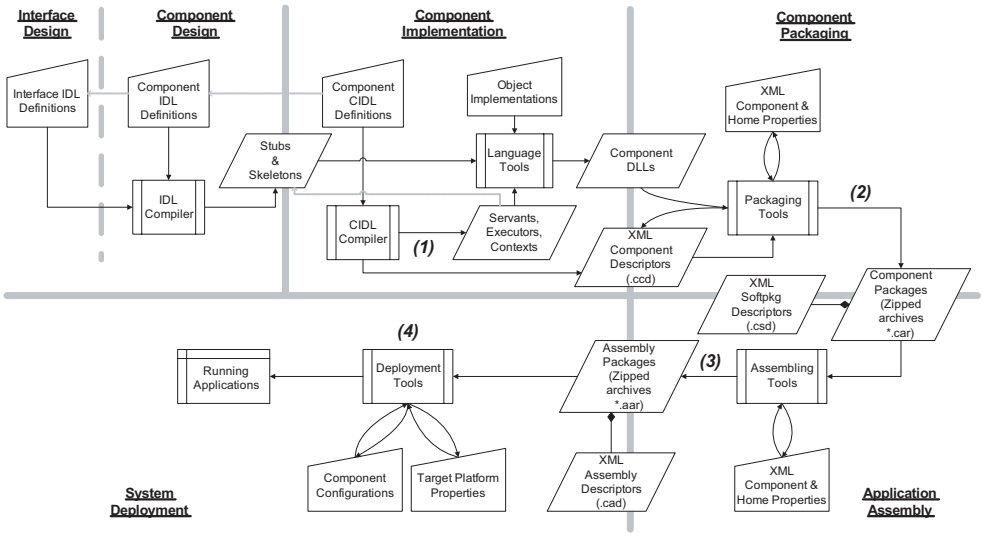

Figure 1.4 **Overview of the CCM Development Lifecycle**

component implementation skeletons and persistent state management mechanisms using the Component Implementation Definition Language (CIDL). CCM packaging tools (2) bundle implementations of a component with related XML-based component metadata. CCM assembly tools (3) use XML-based metadata to describe component compositions, including component locations and interconnections among components, needed to form an assembled application. Finally, CCM deployment tools (4) use the component assemblies and composition metadata to deploy and initialize applications.

The tools and mechanisms defined by CCM collaborate to address the limitations described in Section 1.2.2. The CCM programming paradigm separates the concerns of composing and provisioning reusable software components into the following development roles within the application lifecycle:

- **Component designers**, who define the component features by specifying what each component does and how components collaborate with each other and with their clients. Component designers determine the various types of ports that components offer and/or require.

- **Component implementors,** who develop component implementations and specify the runtime support a component requires via metadata called *component descriptors*.
- **Component packagers**, who bundle component implementations with metadata giving their default properties and their component descriptors into *component packages*.
- **Component assemblers**, who configure applications by selecting component implementations, specifying component instantiation constraints, and connecting ports of component instances via metadata called *assembly descriptors*.
- **System deployers**, who analyze the runtime resource requirements of assembly descriptors and prepare and deploy required resources where component assemblies can be realized.

The CCM specification has recently been finalized by the OMG and is in the process of being incorporated into the core CORBA specification.[1] CCM implementations are now available based on the recently adopted specification (Obj 2002a), including *OpenCCM* by the Universite des Sciences et Technologies de Lille, France, *K2 Containers* by iCMG, *MicoCCM* by FPX, *Qedo* by Fokus, and *CIAO* by the DOC groups at Washington University in St. Louis and the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. The architectural patterns used in CCM (Volter et al. 2002) are also used in other popular component middleware technologies, such as J2EE (Alur et al. 2001; Marinescu and Roman 2002) and .NET.

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 specification and enforcement capabilities are essential for supporting DRE systems. Our work therefore focuses on CCM as the basis for developing QoS-enabled component models that are able to support DRE systems.

1.3.2 Limitations with Conventional Component Middleware for Large-scale DRE Systems

Large-scale DRE applications require seamless integration of many hardware and software systems. Figure 1.5 shows a representative air traffic control system that collects and processes real-time flight status from multiple regional radars. Based on the real-time flight data, the system then reschedules flights, issues air traffic control commands to airplanes in flight, notifies airports, and updates the displays in an airport's flight bulletin boards.

The types of systems shown in Figure 1.5 require complicated provisioning, where developers must connect numerous distributed or collocated subsystems together and define the functionality of each subsystem. Component middleware can reduce the

[1]The CORBA 3.0 specification (Obj 2002c) released by the OMG only includes changes in IDL definition and Interface Repository changes from the Component specification.

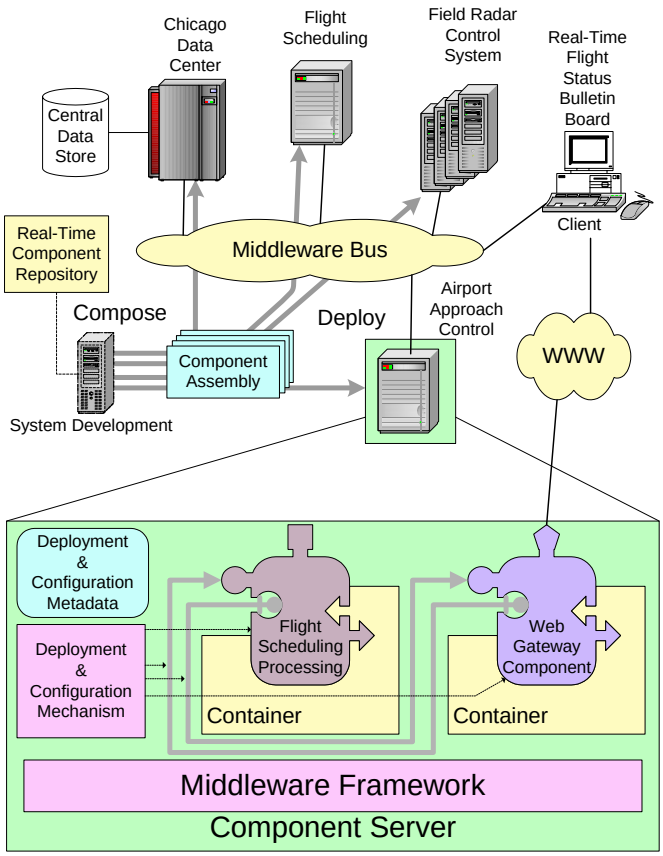

Figure 1.5 **Integrating DRE Applications with Component Middleware**

software development effort for these types of large-scale DRE systems by enabling application development through composition. Conventional component middleware, however, is designed for the needs of business applications, rather than the more complex QoS provisioning needs of DRE applications. Developers are therefore forced to configure and control these QoS mechanisms imperatively in their component implementations to meet the real-time demands of their applications.

It is possible for component developers to take advantage of certain middleware or OS features to implement QoS-enabled components by embedding certain QoS provisioning code within a component implementation. Many QoS capabilities, however, cannot be implemented *solely* within a component because:

- 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 its reusability.
- Certain resources, such as thread pools in Real-time CORBA, 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 with which other components a component implementation will collaborate, the component implementation is not the right level at which to provision QoS.

- Certain QoS assurance mechanisms, such as configuration of non-multiplexed connections between components, 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 component implementations to perform QoS provisioning in isolation.
- Many QoS provisioning policies and mechanisms require the installation of customized infrastructure modules to work correctly in meeting their requirements. However, some of the policies and mechanisms in support of controlled behaviors such as high throughput and low latency, may be inherently incompatible. It is hard for QoS provisioning mechanisms implemented within components to manage these incompatibilities without knowing the end-to-end QoS context *a priori*.

In general, isolating QoS provisioning functionality into each component prematurely commits every implementation to a specific QoS provisioning scenario in a DRE application's lifecycle. 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 hard to reuse, particularly for large-scale DRE systems whose components and applications possess stringent QoS requirements.

1.4 QoS Provisioning and Enforcement with CIAO and QuO Qoskets

This section describes middleware technologies that we have developed to

1. Statically provision QoS resources end-to-end to meet key requirements. Some DRE systems require strict preallocation of critical resources via static QoS provisioning for closed loop systems to ensure critical tasks always have sufficient resources to complete.
2. Monitor and manage the QoS of the end-to-end functional application interactions.
3. Enable the adaptive and reflective decision-making needed to dynamically provision QoS resources robustly and enforce the QoS requirements of applications for open loop environments in the face of rapidly changing mission requirements and environmental/failure conditions.

The middleware technologies discussed in this section apply various aspect-oriented development techniques (Kiczales et al. 1997) to support the separation of QoS systemic behavior and configuration concerns. Aspect-oriented techniques 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 1.3.2 discusses, as DRE systems grow in scope and criticality, a key challenge is to decouple the reusable, multi-purpose, off-the-shelf, resource management aspects of the middleware from aspects that need customization and tailoring to the specific preferences of each application.

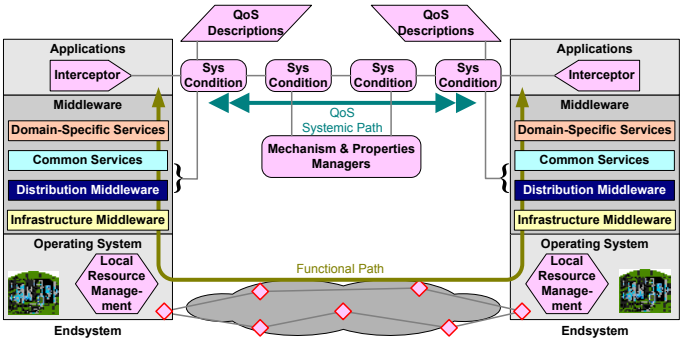

Figure 1.6 **Decoupling the Functional Path from the Systemic QoS Path**

Based on our experience developing scores of large-scale research and production DRE systems over the past two decades, we have found that it is most effective to separate the programming and provisioning of QoS concerns along the two dimensions shown in Figure 1.6 and discussed below:

Functional paths, which are flows of information between client and remote server applications. The middleware is responsible for ensuring that this information is exchanged efficiently, predictably, scalably, dependably, and securely between remote nodes. The information itself is largely application-specific and determined by the functionality being provided (hence the term "functional path").

QoS systemic paths, which are responsible for determining how well the functional interactions behave end-to-end with respect to key DRE QoS properties, such as (1) when, how, and what resources are committed to client/server interactions at multiple levels of distributed systems, (2) the proper application and system behavior if available resources are less than expected, and (3) the failure detection and recovery strategies necessary to meet end-to-end dependability requirements.

In next generation of large-scale DRE systems, the middleware – rather than operating systems or networks alone – will be responsible for separating QoS systemic properties from functional application properties and coordinating the QoS of various DRE system and application resources end-to-end. The architecture shown in Figure 1.6 enables these properties and resources to change independently, *e.g.*, over different distributed system configurations for the same application.

The architecture in Figure 1.6 assumes that QoS systemic paths will be provisioned by a different set of *specialists* – such as systems engineers, administrators, operators, and possibly automated computing agents – and *tools* – such as MDA

tools (Gokhale et al. 2003) – than those customarily responsible for programming functional paths in DRE systems. In conventional component middleware, such as CCM described in Section 1.3.1, there are multiple software development roles, such as component designers, assemblers, and packagers. QoS-enabled component middleware identifies yet another development role: the *Qosketeer* (Zinky et al. 1997). The Qosketeer 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.

1.4.1 Static Qos Provisioning via QoS-enabled Component Middleware and CIAO

Below we present an overview of static QoS provisioning and illustrate how QoS-enabled component middleware, such as CIAO, facilitates the composition of static QoS provisioning into DRE applications.

Overview of Static QoS Provisioning

Static QoS provisioning involves pre-determining the resources needed to satisfy certain QoS requirements and allocating the resources of a DRE system before or during start-up time. Certain DRE applications use static provisioning because they (1) have a fixed set of QoS demands and (2) require tightly-bounded predictability for system functionality. For example, key commands (such as collision warning alerts to a pilot avionic flight systems) should be assured access to resources, *e.g.*, through planned scheduling of those operations or assigning them the highest priority (Gill et al. 2003). In contrast, the handling of secondary functions, such as flight path calculation, can be delayed without significant impact on overall system functioning. In addition, static QoS provisioning is often the simplest solution available, *e.g.*, a video streaming application for an unmanned aerial vehicle (UAV) may simply choose to reserve a fixed network bandwidth for the audio and video streams, assuming it is available (R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali 2003).

To address the limitations with conventional component middleware outlined in Section 1.3.2, it is necessary to make QoS provisioning specifications an integral part of component middleware and apply the aspect-oriented techniques to decouple QoS provisioning specifications from component functionality. This separation of concerns relieves application component developers from tangling the code to manage QoS resources within the component implementation. It also simplifies QoS provisioning that cross-cut multiple interacting components to ensure proper end-to-end QoS behavior.

To perform QoS provisioning end-to-end throughout a component middleware system robustly, static QoS provisioning specifications should be decoupled from component implementations. Static QoS requirements should be specified instead using metadata associated with various application development lifecycles supported by the component middleware. This separation of concerns helps improve component reusability by preventing premature commitment to specific QoS provisioning param-

eters. QoS provisioning specifications can also affect different scopes of components and their behaviors, *e.g.*, thread-pools that are shared among multiple components, versus a priority level assigned to a single component instance.

Different stages of the component software development lifecycle involve commitments to certain design decisions that affect certain scopes of a DRE application. To avoid unnecessarily constraining the development stage, it is therefore important to ensure that we do not commit to specific QoS provisioning prematurely. We review these stages in the component software development lifecycle below and identify the QoS specifications appropriate for each of these stages:

- QoS provisioning specifications that are part of a component implementation need to be specified in the metadata associated with the component implementation, *i.e.*, component descriptors. Examples of QoS-supporting features include (1) a real-time ORB on which a component implementation depends, or (2) QoS provisioning parameters, such as the required priority model that a component implementation elects to make explicitly configurable.
- Configurable specifications of a component, such as its priority model and priority level, can be assigned default values in the component package by associating a component implementation with component property files as described in Section 1.3.1.
- Resources, such as thread-pools and prioritized communication channels, that need to be shared among multiple components in a component server should be allocated separately as logical resource policies. Component instances within the same component server can then share these logical resources by associating them with common logical resource policies in the application assembly metadata described in Section 1.3.1.
- Component assembly metadata must also be extended to provision QoS resources for component interconnections, such as defining private connections, performing network QoS reservations, or establishing pre-connections to minimize runtime latencies.
- Generic component servers provide the execution environment for components, as described in Section 1.3.1. To ensure a component server is configured with the mechanisms and resources needed to support the specified QoS requirements, deployment tools should be extended to include middleware modules that can configure the component servers. Examples include customized communication mechanisms and custom priority mappings.

Figure 1.7 illustrates several types of static QoS provisioning that are common in DRE applications, including:

1. **CPU resources**, which need to be allocated to various competing tasks in a DRE system to make sure these tasks finish on time,
2. **Communication resources**, which the middleware uses to pass messages around to "connect" various components in a distributed system together, and
3. **Distributed middleware configurations**, which are plug-ins that a middleware framework uses to realize QoS assurance.

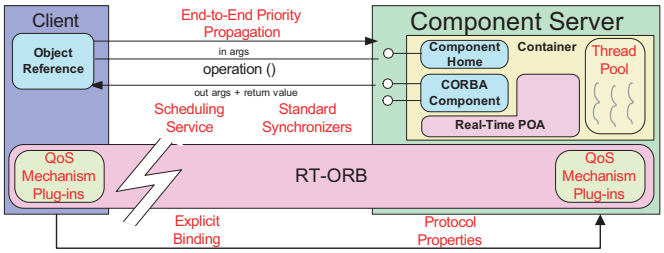

Figure 1.7 **Examples of Static QoS Provisioning**

Static QoS Provisioning with CIAO

Figure 1.8 shows the key elements of the Component-Integrated ACE ORB (CIAO), which is a QoS-enabled implementation of CCM being developed at Washington University, St. Louis and the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. CIAO extends the The ACE ORB (TAO) (Krishna et al. 2003) to sup-

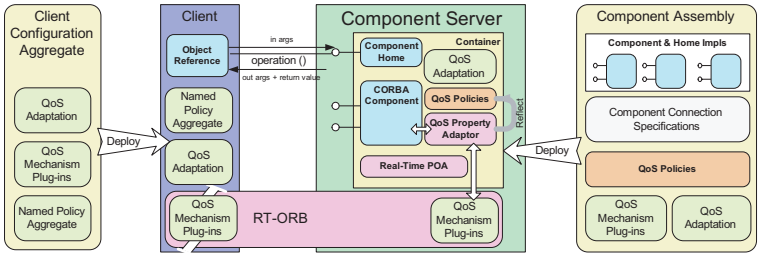

Figure 1.8 **Key Elements in CIAO**

port components and simplify the development of DRE applications by enabling developers to declaratively provision QoS policies end-to-end when assembling a system. TAO is an open-source, high-performance, highly configurable Real-time CORBA ORB that implements key patterns (Schmidt et al. 2000) to meet the demanding QoS requirements of DRE systems. TAO supports the standard OMG CORBA reference model (Obj 2002e) and Real-time CORBA specification (Obj 2002b), with enhancements designed to ensure efficient, predictable, and scalable QoS behavior for high-performance and real-time applications.

TAO is developed atop lower-level middleware called ACE (Schmidt and Huston 2002a,b), which implements core concurrency and distribution patterns (Schmidt et al. 2000) for communication software. ACE provides reusable C++ wrapper facades and framework components that support the QoS requirements of high-performance, real-time applications. ACE and TAO run on a wide range of OS platforms, including Windows, most versions of UNIX, and real-time operating systems such as Sun/Chorus ClassiX, LynxOS, and VxWorks.

To support the role of the Qosketeer, CIAO extends CCM to support static QoS provisioning as follows:

Application assembly descriptor. A CCM *assembly descriptor* specifies how components are composed to form an application. CIAO extends the notion of CCM assembly descriptor to enable Qosketeers to include QoS provisioning specifications and implementations for required QoS supporting mechanisms, such as container-specific policies and ORB configuration options to support these policies. We also extend the CCM assembly descriptor format to allow QoS provisioning at the component-connection level.

Client configuration aggregates. CIAO defines client configuration specifications that Qosketeers can use to configure a client ORB to support various QoS provisioning attributes, such as priority level policy and custom priority mapping. Clients can then be associated with named QoS provisioning policies (defined in an aggregate) that are used to interact with servers and provide end-to-end QoS assurance. CIAO enables the transparent installation of client configuration aggregates into a client ORB.

QoS-enabled containers. CIAO enhances CCM containers to support QoS capabilities, such as various server-specified Real-time CORBA policies. These QoS-enabled containers provide the central ORB interface that Qosketeers can use to provision component QoS policies and interact with ORB endsystem QoS assurance mechanisms, such as Real-time POA and ORB proprietary internal interfaces, required by the QoS policies.

QoS adaptation. CIAO also supports installation of meta-programming hooks, such as portable interceptors, servant managers, smart proxies and skeletons, that can be used to inject dynamic QoS provisioning transparently. Section 1.4.2 describes these QoS adaptation mechanisms in detail.

To support the capabilities described above, CIAO extends the CCM metadata framework so that developers can bind the QoS requirements and the supporting mechanisms during various stages of the development lifecycle, including component implementation, component packaging, application assembly, and application deployment. These capabilities enable CIAO to statically provision the types of QoS resources outlined in Section 1.4.1 as follows:

CPU resources. These policies specify how to allocate CPU resources when running certain tasks, *e.g.*, by configuring the priority models and priority levels of component instances.

Communication resources. These policies specify strategies for reserving and allocating communication resources for component connections, *e.g.*, an assembly can request a private connection between two critical components in the system and reserve bandwidth for the connection using the RSVP protocol (R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali 2003).

Distributed middleware configuration. These policies specify the required software modules that control the QoS mechanisms for:

- **ORB configurations.** To enable the configuration of higher level policies, CIAO needs to know how to support the required functionality, such as installing and configuring customized communication protocols.

- **Meta-programming mechanisms.** Software modules, such as those developed with the QuO Qosket middleware described in Section 1.4.2 that implement dynamic QoS provisioning and adaptation, can be installed statically at system composition time via meta-programming mechanisms, such as smart proxies and interceptors (Wang et al. 2001b).

In summary, DRE application developers can use CIAO at various points of the development cycle to (1) decouple QoS provisioning functionality from component implementations and (2) compose static QoS provisioning capabilities into the application via the component assembly and deployment phases. This approach enables maximum reusability of components and robust composition of DRE applications with diverse QoS requirements.

1.4.2 Dynamic QoS Provisioning via QuO Adaptive Middleware and Qoskets

Section 1.4.1 described the static QoS provisioning capabilities provided by CIAO. We now present an overview of dynamic QoS provisioning and describe how the QuO Qosket middleware framework Schantz et al. (2002) can be used to manage dynamic QoS provisioning for DRE applications.

Overview of Dynamic QoS Provisioning

Dynamic QoS provisioning involves the allocation and management of resources at runtime to satisfy application QoS requirements. Certain events, such as fluctuations in resource availability due to temporary overload, changes in QoS requirements, or reduced capacity due to failures or external attacks, can trigger reevaluation and reallocation of resources. Dynamic QoS provisioning requires the following middleware capabilities:

- To detect changes in available resources, middleware must *monitor* DRE system status to determine if reallocations are required. For example, network bandwidth is often shared by multiple applications on computers. Middleware for bandwidth-sensitive applications, such as video conferencing or image processing, is therefore responsible for determining changes in total available bandwidth and bandwidth usable by a specific application.
- To *adapt* to the change by adjusting the use of resources required by an application, when available resources change. For instance, middleware that supports a video conferencing application (R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali 2003) may choose to (1) lower the resolution temporarily when there is less available network bandwidth to support the original resolution and (2) switch back to the higher resolution when sufficient bandwidth becomes available. Other examples (such as changing the degree of fault tolerance, substituting a simple text interface in place of imagery intensive applications with short real-time constraints, or temporarily ceasing operation in deference to higher priority activities) illustrate the breadth of DRE application adaptive behavior.

Although applications can often implement dynamic QoS provisioning functionality themselves using conventional middleware, they must work around the current structures intended to simplify the development of DRE systems, which often becomes counter productive. Such *ad hoc* approaches lead to non-portable code that depends on specific OS features, tangled implementations that are tightly coupled with the application software, and other problems that make it hard to adapt the application to changing requirements, and even harder to reuse an implementation. It is therefore essential to separate the functionality of dynamic QoS provisioning from both the lower level distribution middleware *and* the application functionality.

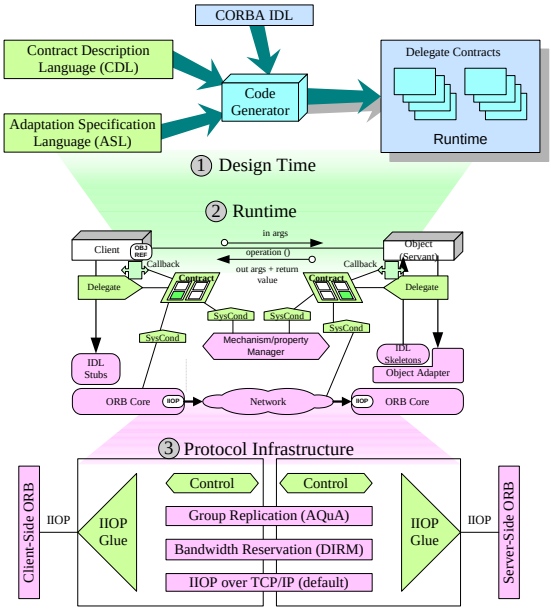

Figure 1.9 **Examples of Dynamic QoS Provisioning**

Figure 1.9 illustrates the types of dynamic QoS provisioning abstractions and mechanisms that are necessary in large-scale DRE systems:

1. **Design formalisms** to specify the level of service desired by a client, the level of service a component expects to provide, operating regions interpreting the ranges of possible measured QoS, and actions to take when the level of QoS changes.
2. **Runtime capabilities** to adapt application behavior based upon the current state of QoS in the system.
3. **Management mechanisms** that keep track of the resources in the protocol and middleware infrastructure that need to be measured and controlled dynamically to meet application requirements and mediate total demand on system resources across and between the platforms.

Overview of QuO

The Quality Objects (QuO) (Zinky et al. 1997) framework is adaptive middleware developed by BBN Technologies that allows DRE system developers to use aspect-oriented software development (Kiczales et al. 1997) techniques to separate the concerns of QoS programming from "business" logic in DRE applications. The QuO framework allows DRE developers to specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at runtime.

The runtime architecture shown in Figure 1.9 illustrates how the elements in QuO support the following dynamic QoS provisioning needs:

- **Contracts** specify the level of service desired by a client, the level of service a component expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes.
- **Delegates** act as local proxies for remote components. Each delegate provides an interface similar to that of the remote component stub, but adds locally adaptive behavior based upon the current state of QoS in the system, as measured by the contract.
- **System Condition objects** provide interfaces to resources, mechanisms, and ORBs in the DRE system that need to be measured and controlled by QuO contracts.

QuO applications can also use resource or property managers that manage given QoS resources such as CPU or bandwidth, or properties such as availability or security, for a set of QoS-enabled server components on behalf of the QuO clients using those server components. In some cases, managed properties require mechanisms at lower levels in the protocol stack, such as replication or access control. QuO provides a gateway mechanism (Schantz et al. 1999) that enables special-purpose transport protocols and adaptation below the ORB.

QuO contracts and delegates support the following two different models for triggering middleware- and application-level adaptation:

In-band adaptation applies within the context of remote object invocations, and is a direct side effect of the invocation (hence the name "in-band"). A QuO delegate applies the Interceptor Pattern (Schmidt et al. 2000) to provide adaptation and QoS control for the object access. The QuO delegate can perform in-band adaptation whenever a client makes a remote operation call and whenever an invoked operation returns. The delegates on the client and server check the state of the relevant contracts and choose behaviors based upon the state of the system. These behaviors can include shaping or filtering the method data, choosing alternate methods or server objects, performing local functionality, etc.

Out-of-band adaptation applies more generally to the state of an application, a system, or a subsystem outside the context of invocations. QuO supports out-of-band adaptation by monitoring system condition objects within a system. Whenever the monitored conditions change (or whenever they change beyond a specified threshold),

a system condition object triggers an asynchronous evaluation of the relevant contracts. If this results in a change in contract region, *i.e.*, a state change, it in turn triggers adaptive behavior asynchronous to any object interactions.

For more information about the QuO adaptive middleware, see (Loyall et al. 1998; Schantz et al. 1999; Vanegas et al. 1998; Zinky et al. 1997).

Qoskets: QuO Support for Reusing Systemic Behavior

One goal of QuO is to separate the role of a systemic QoS programmer from that of an application programmer. A complementary goal of this separation of programming roles is that systemic QoS behaviors can be encapsulated into reusable units that are not only developed separately from the applications that use them, but that can be reused by selecting, customizing, and binding them to new application programs. To support this goal, we have defined a *Qosket* (Schantz et al. 2002) as a unit of encapsulation and reuse of systemic behavior in QuO applications. A Qosket is each of the following, simultaneously:

- **A collection of cross-cutting implementations**, *i.e.*, a Qosket is a set of QoS specifications and implementations that are woven throughout a DRE application and its constituent components to monitor and control QoS and systemic adaptation.
- **A packaging of behavior and policy**, *i.e.*, a Qosket generally encapsulates elements of an adaptive QoS behavior and a policy for using that behavior, in the form of contracts, measurements and code to provide adaptive behavior.
- **A unit of behavior reuse**, largely focused on a single property, *i.e.*, a Qosket can be used in multiple DRE applications, or in multiple ways within a single application, but typically deals with a single property (*e.g.*, performance, dependability, or security).

Qoskets are an initial step towards individual behavior packaging and reuse, as well as a significant step toward the more desirable (and much more complex) ability to compose behaviors within an application context. They are also a means toward the larger goal of flexible design tradeoffs at runtime among properties (such as real-time performance, dependability, and security) that vary with the current operating conditions. Qoskets are used to bundle in one place all of the specifications for controlling systemic behavior, independent of the application in which the behavior might end up being used.

In practice, a Qosket is a collection of the interfaces, contracts, system condition objects, callback components, unspecialized adaptive behavior, and implementation code associated with a reusable piece of systemic behavior. A Qosket is fully specified by defining the following:

1. The contracts, system condition objects, and callback components it encapsulates
2. The Adaptation Specification Language (ASL) template code, defining partial specifications of adaptive behavior,

3. Implementation code for instantiating the Qosket's encapsulated components, for initializing the Qosket, and for implementing the Qosket's defined systemic measurement, control, and adaptation, and
4. The interfaces that the Qosket exposes.

The general structure of Qoskets, including modules they encapsulate and interfaces they expose, is illustrated in Figure 1.10. The two interfaces that Qoskets expose

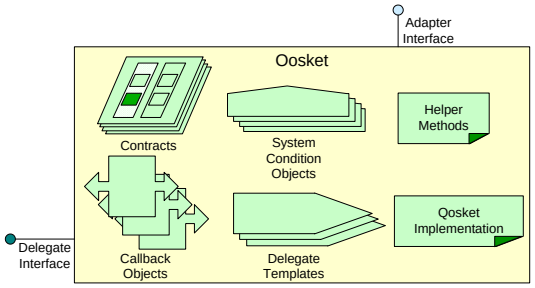

Figure 1.10: **Qoskets Encapsulate QuO Components into Reusable Behaviors**

correspond to these two use cases:

- **The adapter interface**, which is an application programming interface. This interface provides access to QoS measurement, control, and adaptation features in the Qosket (such as the system condition objects, contracts, and so forth) so that they can be used anywhere in an application.
- **The delegate interface**, which is an interface to the in-band method adaptation code. In-band adaptive behaviors of delegates are specified in the QuO ASL language. The adaptation strategies of the delegate are encapsulated and woven into the application using code generation techniques.

1.4.3 Integrated QoS provisioning via CIAO and Qoskets

As discussed in Section 1.4.2, Qoskets provide abstractions for dynamic QoS provisioning and adaptive behaviors. The current implementation of Qoskets in QuO, however, requires application developers to modify their application code manually to "plug in" adaptive behaviors into existing applications. Rather than retrofitting DRE applications to use Qosket specific interfaces, it would be more desirable to use existing and emerging COTS component technologies and standards to encapsulate QoS management, both static and dynamic.

Conversely, CIAO allows system developers to compose static QoS provisioning, adaptation behaviors, and middleware support for QoS resources allocating and managing mechanisms into DRE applications transparently, as depicted in Section 1.4.1. CIAO did not, however, initially provide an abstraction to model, define, and specify dynamic QoS provisioning. We are therefore leveraging CIAO's capability to configure Qoskets transparently into component servers to provide an integrated QoS provisioning solution, which enables the composition of both static and dynamic QoS provisioning into DRE applications.

The static QoS provisioning mechanisms in CIAO enable the composition of Qoskets into applications as part of component assemblies. As in the case of provisioning static QoS policies, developers can compose and weave together Qosket modules of different granularities into component-based applications at various stages of the development lifecycle. The most straightforward approach is to implement Qosket behaviors as CCM components, as shown in Figure 1.11. Each Qosket component offers

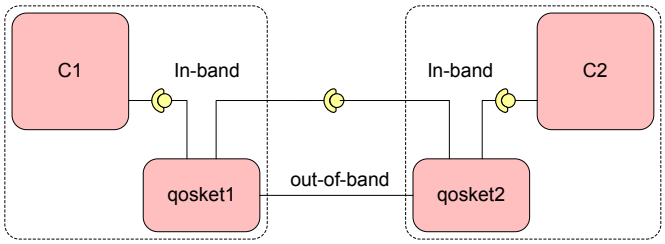

Figure 1.11 **Composing Qosket components in an application**

interception facets and receptacles that can be used to provide in-band adaptation along the functional path. Qosket components can also provide other ports to support out-of-band adaptation and interact with related QoS management mechanisms.

Implementing Qoskets as CCM components takes advantage of the CCM assembly framework and is thus suitable for prototyping a dynamic QoS provisioning capability. It requires modifying the application assembly directly, however, and must therefore be woven into an application at application composition time. Inserting Qoskets as conventional CCM components requires the use of regular CCM interfaces to define interception facets and receptacles. This exposure could in theory impose significant overhead in network communication because it permits Qosket delegate components to be deployed remotely from the targeting components, which defeats the purpose of using Qoskets to closely manage local behavior to achieve end to end QoS management.

To resolve these concerns, CIAO provides meta-mechanisms to weave in Qosket delegates and integrate Qosket modules into a CCM application. As shown in Figure 1.12, CIAO can install a Qosket using the following mechanisms:

- QuO delegates can be implemented as smart proxies (Wang et al. 2001b) and injected into components using interceptors by providing hooks in containers. The delegate configuration metadata can be injected into assembly descriptors or the client-side configuration aggregates described in Section 1.4.1.
- Developers can specify a Qosket-specific ORB configuration and assemble QoS mechanisms into the component server or client ORB.
- Out-of-band provisioning and adaptation modules, such as contracts, system conditions, and callback components, can continue to be implemented and assembled as separate CCM components into servers.
- Interactions between delegates and system condition objects can then be specified by additional assembly descriptors that, *e.g.*, connect the delegate meta-data with the packaging-time meta-data for the system condition components.

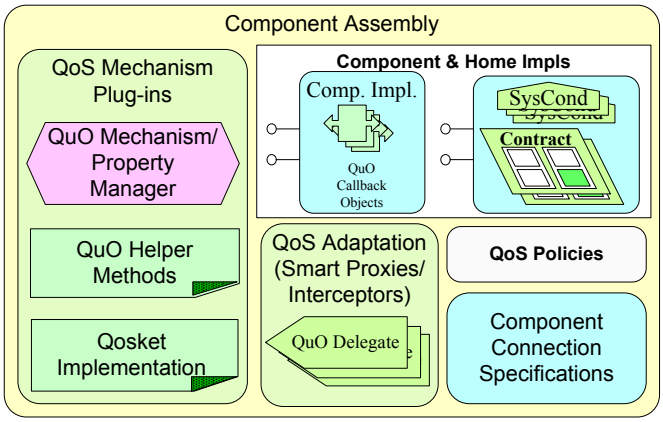

Figure 1.12 **Composing a Qosket using CIAO**

The approach described above requires support from the component middleware but provides better control over the integration of Qosket modules. Since this approach does not require the modification of application assembly at a *functional* level, it more thoroughly decouples QoS management from the application logic written by component developers.

Although using CIAO to compose Qoskets into component assemblies simplifies retrofitting, a significant problem remains: *component cross-cutting*. Qoskets are useful for separating concerns between systemic QoS properties and application logic, as well as implementing limited cross-cutting between a single client/component pair. Neither Qoskets nor CIAO yet provide the ability to cross-cut application components, however. Many QoS-related adaptations will need to modify the behavior of several components at once, likely in a distributed way. Some form of dynamic aspect-oriented programming might be used in this context, which is an area of ongoing research (Office n.d.).

1.5 Related Work

This section reviews work on QoS provisioning mechanisms using the taxonomy shown in Figure 1.13. One dimension depicted in Figure 1.13 is *when QoS provisioning is performed*, *i.e.*, static versus dynamic QoS provisioning, as described in Section 1.1. 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 1.13 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.

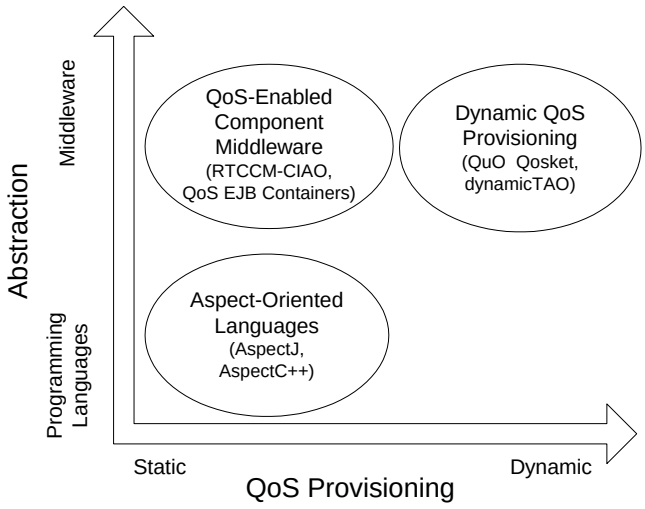

Figure 1.13 **Taxonomy of QoS Provisioning Enabling Mechanisms**

Dynamic QoS Provisioning. In their *dynamicTAO* project, Kon and Campbell (Kon et al. 2002) 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 falls into the same category as *Qoskets* (shown in Figure 1.13), 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 (Gokhale et al. 2003) 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.

QoS-enabled Component Middleware. Middleware can apply the Quality Connector pattern (Cross and Schmidt 2002) 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 the vehicle for applying meta-programming techniques for QoS assurance control in component middleware, as previously identified in (Wang et al. 2001a). Containers can also help apply aspect-oriented software development (Kiczales et al. 1997) techniques to plug in different systemic behaviors (Conan et al. 2001). These projects are similar to CIAO in that they provide a mechanism to inject "aspects"

into applications statically at the middleware level.

Miguel de Miguel further develops the work on 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 (de Miguel 2002). 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 1.3.2 examines the limitations of implementing QoS behavior logic in component implementations.

QoS Enabled Distributed Objects (Qedo). The Qedo project (FOKUS n.d.) 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 that defines 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 (Ritter et al. 2003).

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 the 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.

Aspect-Oriented Programming Languages. Aspect-oriented programming (AOP) (Kiczales et al. 1997) languages provide language-level abstractions for weaving different aspects that cross-cut multiple layers of a system. Examples of AOP languages include AspectJ (Kiczales et al. 2001) and AspectC++ (Olaf Spinczyk and Andreas Gal and Wolfgang Schröder-Preikschat 2002). Similar to AOP, CIAO supports injection of aspects into systems at the middleware level using meta-programming techniques. Both CIAO and AOP weave aspects statically, *i.e.*, before program execution, and neither defines an abstraction for dynamic QoS provisioning behaviors. In contrast, Qoskets use AOP techniques dynamically to organize and connect the various dimensions of the QoS middleware abstractions with each other and with the program to which the behavior is being attached.

1.6 Concluding Remarks

DRE applications constitute an increasingly important domain that requires stringent support for multiple simultaneous QoS properties, such as predictability, latency, and dependability. To meet these requirements, DRE applications have historically been custom-programmed to implement their QoS provisioning needs, making them

so expensive to build and maintain that they cannot adapt readily to meet new functional needs, different QoS provisioning strategies, hardware/software technology innovations, or market opportunities. This approach is increasingly infeasible, since the tight coupling between custom DRE software modules increases the time and effort required to develop and evolve DRE software. Moreover, QoS provisioning cross-cuts multiple layers in applications and requires end-to-end enforcement that makes DRE applications even harder to develop, maintain, and adapt.

One way to address these coupling issues is by refactoring common application logic into *object-oriented application frameworks* (Johnson 1997). This solution has limitations, however, since application objects can still interact directly with each other, which encourages tight coupling. Moreover, framework-specific bookkeeping code is also required within the applications to manage the framework, which can also tightly couple applications to the framework they are developed upon. It becomes non-trivial to reuse application objects and port them to different frameworks.

Component middleware (Szyperski 1998) has emerged as a promising solution to many of the known limitations with object-oriented middleware and application frameworks. Component middleware supports a higher level packaging of reusable software artifacts that can be distributed or collocated throughout a network. Existing component middleware, however, does not yet address the end-to-end QoS provisioning needs of DRE applications, which transcend component boundaries. QoS-enabled middleware is therefore necessary to separate end-to-end QoS provisioning concerns from application functional concerns.

This chapter describes how the Component Integrated ACE ORB (CIAO) middleware developed by Washington University in St. Louis and the Institute for Software Integrated Systems at Vanderbilt University is enhancing the standard CCM specification to support static QoS provisioning by pre-allocating resources for DRE applications. We also describe how BBN's QuO Qosket middleware framework provides powerful abstractions that help define and implement reusable dynamic QoS provisioning behaviors. By combining QuO Qoskets with CIAO, we are developing an integrated end-to-end QoS provisioning solution for DRE applications.

When augmented with Model Driven Middleware tools, such as CoSMIC (Gokhale et al. 2003), QoS-enabled component middleware and applications can be provisioned more effectively and efficiently at even higher levels of abstraction. CIAO, QuO Qoskets, and CoSMIC are open-source software that is currently available and can be obtained from `www.dre.vanderbilt.edu/CIAO/`, `quo.bbn.com/`, and `www.dre.vanderbilt.edu/cosmic`, respectively.

Bibliography

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

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

Alur D, Crupi J and Malks D 2001 *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall.

Box D 1998 *Essential COM*. Addison-Wesley, Reading, MA.

Buschmann F, Meunier R, Rohnert H, Sommerlad P and Stal M 1996 *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York.

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

Conan D, Putrycz E, Farcet N and DeMiguel M 2001 Integration of Non-Functional Properties in Containers. *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP)*.

Cross JK and Schmidt DC 2002 Applying the Quality Connector Pattern to Optimize Distributed Real-time and Embedded Middleware In *Patterns and Skeletons for Distributed and Parallel Computing* (ed. Rabhi F and Gorlatch S) Springer Verlag.

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

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

Douglas C. Schmidt and Frank Buschmann 2003 Patterns, Frameworks, and Middleware: Their Synergistic Relationships *International Conference on Software Engineering (ICSE)* IEEE/ACM, Portland, Oregon.

FOKUS n.d. Qedo Project Homepage `http://qedo.berlios.de/`.

Gamma E, Helm R, Johnson R and Vlissides J 1995 *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

Gill C, Schmidt DC and Cytron R 2003 Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*.

Gokhale A, Schmidt DC, Natarajan B and Wang N 2002 Applying Model-Integrated Computing to Component Middleware and Enterprise Applications. *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules*.

Gokhale A, Schmidt DC, Natarajan B, Gray J and Wang N 2003 Model Driven Middleware In *Middleware for Communications* (ed. Mahmoud Q) Wiley and Sons New York.

Johnson R 1997 Frameworks = Patterns + Components. *Communications of the ACM*.

Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J and Griswold WG 2001 An overview of AspectJ. *Lecture Notes in Computer Science* **2072**, 327–355.

Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier JM and Irwin J 1997 Aspect-Oriented Programming *Proceedings of the 11th European Conference on Object-Oriented Programming*.

Kon F, Costa F, Blair G and Campbell RH 2002 The Case for Reflective Middleware. *Communications of the ACM* **45**(6), 33–38.

Krishna AS, Schmidt DC, Klefstad R and Corsaro A 2003 Real-time Middleware In *Middleware for Communications* (ed. Mahmoud Q) Wiley and Sons New York.

Loyall JP, Bakken DE, Schantz RE, Zinky JA, Karr D, Vanegas R and Anderson KR 1998 QoS Aspect Languages and Their Runtime Integration. *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components (LCR98)* pp. 28–30.

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

Marinescu F and Roman E 2002 *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, New York.

Morgenthal JP 1999 Microsoft COM+ Will Challenge Application Server Market www.microsoft.com/com/wpaper/complus-appserv.asp.

Obj 2002a *CORBA Components* OMG Document formal/2002-06-65 edn.

Obj 2002b *Real-time CORBA Specification* OMG Document formal/02-08-02 edn.

Obj 2002c *The Common Object Request Broker: Architecture and Specification* 3.0.2 edn.

Obj 2002d *The Common Object Request Broker: Architecture and Specification* 2.6.1 edn.

Obj 2002e *The Common Object Request Broker: Architecture and Specification* 3.0.2 edn.

Office DIE n.d. Program Composition for Embedded Systems (PCES) `www.darpa.mil/ixo/`.

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

Postel J 1980 User Datagram Protocol. *Network Information Center RFC 768* pp. 1–3.

Postel J 1981 Transmission Control Protocol. *Network Information Center RFC 793* pp. 1–85.

R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali 2003 Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware *Proceedings of Middleware 2003, 4th International Conference on Distributed Systems Platforms* IFIP/ACM/USENIX, Rio de Janeiro, Brazil.

Rajkumar R, Lee C, Lehoczky JP and Siewiorek DP 1998 Practical Solutions for QoS-based Resource Allocation Problems *IEEE Real-Time Systems Symposium* IEEE, Madrid, Spain.

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

Rosenberry W, Kenney D and Fischer G 1992 *Understanding DCE*. O'Reilly and Associates, Inc.

Schantz R, Loyall J, Atighetchi M and Pal P 2002 Packaging Quality of Service Control Behaviors for Reuse *Proceedings of the* 5^{th} *IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)* IEEE/IFIP, Crystal City, VA.

Schantz RE and Schmidt DC 2002 Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications In *Encyclopedia of Software Engineering* (ed. Marciniak J and Telecki G) Wiley & Sons New York.

Schantz RE, Thomas RH and Bono G 1986 The Architecture of the Cronus Distributed Operating System *Proceedings of the 6th International Conference on Distributed Computing Systems*, pp. 250–259 IEEE, Cambridge, MA.

Schantz RE, Zinky JA, Karr DA, Bakken DE, Megquier J and Loyall JP 1999 An object-level gateway supporting integrated-property quality of service *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*.

Schmidt DC and Huston SD 2002a *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Boston.

Schmidt DC and Huston SD 2002b *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts.

Schmidt DC, Stal M, Rohnert H and Buschmann F 2000 *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York.

Sharp DC 1998 Reducing Avionics Software Cost Through Component Based Product Line Development *Proceedings of the 10th Annual Software Technology Conference*.

Sharp DC 1999 Avionics Product Line Software Architecture Flow Policies *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*.

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

Sun Microsystems 2001 Java^{TM} 2 Platform Enterprise Edition http://java.sun.com/j2ee/index.html.

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

Vanegas R, Zinky JA, Loyall JP, Karr D, Schantz RE and Bakken DE 1998 QuO's Runtime Support for Quality of Service in Distributed Objects. *Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware 98)*.

Volter M, Schmid A and Wolff E 2002 *Server Component Patterns: Component Infrastructures Illustrated with EJB*. Wiley Series in Software Design Patterns, West Sussex, England.

Wang N, Schmidt DC and O'Ryan C 2000 An Overview of the CORBA Component Model In *Component-Based Software Engineering* (ed. Heineman G and Councill B) Addison-Wesley Reading, Massachusetts.

Wang N, Schmidt DC, Kircher M and Parameswaran K 2001a Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications. *IEEE Distributed Systems Online*.

Wang N, Schmidt DC, Othman O and Parameswaran K 2001b Evaluating Meta-Programming Mechanisms for ORB Middleware. *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies* **39**(10), 102–113.

Wollrath A, Riggs R and Waldo J 1996 A Distributed Object Model for the Java System. *USENIX Computing Systems*.

Zinky JA, Bakken DE and Schantz R 1997 Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems* **3**(1), 1–20.