Total Quality of Service Provisioning in Middleware and Applications

Nanbor Wang
Christopher D. Gill
{nanbor,cdgill}@cs.wustl.edu
Dept. of Computer Science

Washington University
One Brookings Drive
St. Louis, MO 63130, USA

Douglas C. Schmidt
schmidt@uci.edu
Dept. of Electrical
and Computer Engineering
University of California
616E Engineering Tower
Irvine, CA 92697, USA

Aniruddha Gokhale
Balachandran Natarajan
{a.gokhale,b.natarajan}@vanderbilt.edu
Institute for Software
Integrated Systems
Vanderbilt University
P.O. Box 36, Peabody
Nashville, TN 37203, USA

Craig Rodrigues, Joseph P. Loyall and Richard E. Schantz
{crodrigu,jloyall,schantz}@bbn.com
BBN Technologies
10 Moulton Street
Cambridge, MA 02138, USA

Abstract

Commercial of-the-shelf (COTS) distribution middleware is gaining acceptance in the distributed real-time and embedded (DRE) community as (1) the cost and time required to develop and verify DRE applications precludes developers from implementing DRE applications from scratch and (2) implementations of standard COTS middleware specifications, such as CORBA, mature. Although standard COTS specifications define the interfaces and policies to provision DRE application resources end-to-end, they do not yet provide sufficient abstractions to separate quality of service (QoS) policy configurations and adaptations from application functionality. DRE application developers must therefore configure QoS policies and program adaptation mechanisms in an ad hoc way. This tight-coupling tends to scatter the code that ensures end-to-end QoS throughout many parts of DRE applications, making it hard to configure, validate, modify, and evolve complex DRE applications consistently.

This paper provides three contributions to the study of the development of QoS-enabled DRE applications. First, we illustrate how standard component-based middleware can be enhanced to flexibly compose static QoS provisioning policies with application logic. Second, we describe how adaptive middleware capabilities enable developers to abstract and encapsulate reusable dynamic QoS provisioning and adaptive behaviors. Third, we illustrate how component-based middleware and adaptive middleware capabilities can be integrated to provide a total QoS provisioning solution for DRE applications.

Keywords:

QoS Provisioning, QoS Adaptation, Middleware, CORBA Component Model

1 Introduction

Commercial-off-the-shelf (COTS) distribution middleware technologies, such as the OMG's CORBA, Sun's EJB/J2EE, and Microsoft's COM+/SOAP/.NET, have matured considerably in recent years. They are increasingly used to reduce the time and effort required to develop applications in a broad range of domains. Historically, these middleware technologies have been applied to *enterprise applications* [1], which are a large class of applications that perform important business functions, such as planning enterprise resource usage, automating key business functions, and managing supply chains and customer relationships. Examples of enterprise applications include airline reservation systems, bank asset management systems, and just-in-time inventory control systems.

More recently, middleware has been applied to distributed real-time and embedded (DRE) applications with stringent quality of service (QoS) requirements for predictability, latency, efficiency, scalability, dependability, and security. There are many types of DRE applications, but they have one thing in common: *the right answer delivered too late becomes the wrong answer.* Examples 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 that help aircrafts navigate through their route legs. DRE applications are an increasingly important domain since over 99% of all mi-

croprocessors are now used for embedded systems [2] to control physical, chemical, or biological processes and devices in real-time.

Regardless of the domain in which middleware is applied, it helps expedite the application development process by shielding programmers from many accidental and inherent complexities, such as platform and language heterogeneity, resource location, and fault tolerance. *Component middleware* is a maturing 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) [3], Java 2 Enterprise Edition (J2EE) [4], and the Component Object Model (COM) [5], which use different APIs, different protocols, and different component models.

As the use of middleware becomes more pervasive, DRE applications are increasingly combined to form distributed systems that are joined together by the Internet and intranets. These systems can further be combined with other distributed systems to create "systems of systems." Examples of these large-scale systems of systems include:

- *Just-in-time manufacturing inventory control systems* that schedule the delivery of supplies to improve efficiency and
- *Military command and control systems* that gather and assimilate information from various devices (such as unmanned arial vehicles and wearable computers), present and analyze the information, and coordinate the deployment of available forces and weaponry.

Most DRE applications have stringent QoS requirements that must be satisfied simultaneously in real-time. Examples of these QoS requirements include processing resources allocation and network latency, jitter, and bandwidth. To ensure 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 the amount of 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 4.1 describes the range of QoS resources that can be provisioned statically.
- *Dynamically*, where the amount of resources required are determined and adjusted based on the runtime system status. Examples of dynamic QoS provisioning include runtime reallocations to handle bursty CPU load, primary and second storage, and network traffic demands. Section 5.1 describes the range of QoS resources that can be provisioned dynamically.

QoS provisioning in large-scale DRE systems cross-cuts multiple system layers and requires end-to-end enforcement. Existing component middleware technologies, such as CCM, J2EE, and .NET, were designed largely for applications with conventional business-oriented QoS requirements, such as data persistence, encryption, and transactional support. They therefore do not enforce the stringent QoS requirements of DRE applications effectively. What is needed is a *QoS-enabled component middleware* that preserves existing support for heterogeneity in standard component middleware, yet also provides multiple dimensions of QoS provisioning and enforcement to meet the end-to-end QoS requirements of DRE applications.

This paper provides the following three contributions toward the study of QoS-enabled component middleware that is essential to the development of large-scale DRE applications:

- 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 the standard CORBA Component Model (CCM) [3] called the *Component-Integrated ACE ORB* (CIAO), which is being developed at Washington University, St. Louis.
- We describe how dynamic QoS provisioning and adaptation can be addressed using middleware capabilities called *Qoskets*, which are collections of reusable software modules of the Quality Objects (QuO) [6] middleware developed by BBN Technologies. The discussion concentrates on the major elements in QuO are defined, developed, and used to implement dynamic QoS provision and adaptive behaviors.
- We discuss how Qoskets can be combined with CIAO to compose both static QoS provisioning and dynamic adaptive QoS assurance into DRE applications. In particular, how CIAO combines qoskets to weave in the software elements in qoskets to create an integrated QoS-enabled component model which offers a total QoS provisioning solution for DRE applications.

The remainder of this paper is organized as follows: Section 2 describes how component middleware addresses key

limitations of object-oriented middleware, as well as how conventional component middleware fails to support DRE application development effectively; Section 3 illustrates how QoS provisioning can be separated from applications functionality and the needs for QoS enabled component middleware for developing DRE applications; Section 4 demonstrates how CIAO component middleware extends the traditional component middleware to support static QoS provisioning; Section 5 explains how BBN's QuO middleware framework enables developers to defines reusable dynamic QoS provisioning behaviors for DRE applications ; Section 6 describes how CIAO and QuO can be integrated to provide total QoS provisioning for DRE applications; and Section 7 presents concluding remarks.

2 Component Middleware: A Powerful Approach to Building DRE Applications

This section motivates the need for component middleware and then presents an overview of component middleware. It also discusses why conventional component middleware fails to support key QoS provisioning needs of DRE applications.

2.1 Overview of Middleware Capabilities

Middleware is reusable software that resides between applications and underlying operating systems, network protocol stacks, and hardware [7]. 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. When implemented properly, middleware can help to:

- Shield application developers from low-level, tedious, and error-prone platform details, such as socket-level network programming.
- Simplify the development of distributed applications by providing a consistent set of capabilities that are closer to 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.
- 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 developer-oriented services, such as transactional logging and security, that have proven necessary to operate effectively in a distributed environment.
- Ease the integration of software artifacts developed by multiple technology suppliers.

Various technologies, such as OSF's Distributed Computing Environment (DCE) [8], IBM's MQ Series [9], and CORBA [10], have emerged over the past two decades to alleviate complexities associated with developing software for enterprise applications. Their successes have 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 the infrastructure, middleware enables application developers to concentrate on programming application-specific functionality, rather than wrestling repeatedly with lower-level infrastructure challenges.

2.2 Limitations with Object-oriented Middleware

The Object Management Architecture (OMA) in the CORBA 2.x specification [11] defines an object-oriented middleware standard for building portable distributed applications. The CORBA 2.x specification focuses on *interfaces*, which are contracts between clients and servers that define how clients *view* and *access* object services provided by a server. Objects can either be collocated or distributed throughout a network.

Although the CORBA object model has certain virtues, such as location and implementation language transparency, it also has the following limitations:

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

Lack of generic component servers. CORBA 2.x does not specify a generic *component server* framework to perform common "bookkeeping" work, including initializing the broker and its QoS policies, providing common services such as an event service, 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 are still responsible for determining how object implementations are installed in an ORB and the interaction between the ORB and object implementations. The lack of a generic component server standard has

yielded tightly coupled, *ad-hoc* server implementations, which increase the complexity of software upgrades and reduce the reusability and flexibility of CORBA-based applications.

2.3 Promising Solution: Component Middleware

In recent years, *component middleware* [12] has emerged to address the limitations with object-oriented middleware outlined above. Component middleware addresses these issues by (1) creating a virtual boundary around application components that interact with each others only through well-defined interfaces and (2) then composing and executing components in generic component servers. The OMG's CCM addresses the limitations with object-oriented middleware described above. It has many similarities to other component middleware frameworks, *e.g.*, EJB and COM+. We base our work on the CCM since CORBA is the only COTS middleware that has made a substantial progress in satisfying the QoS requirements of DRE systems.

Figure 1 shows an overview of the runtime architecture of the CCM model. *Components* are implementation entities that

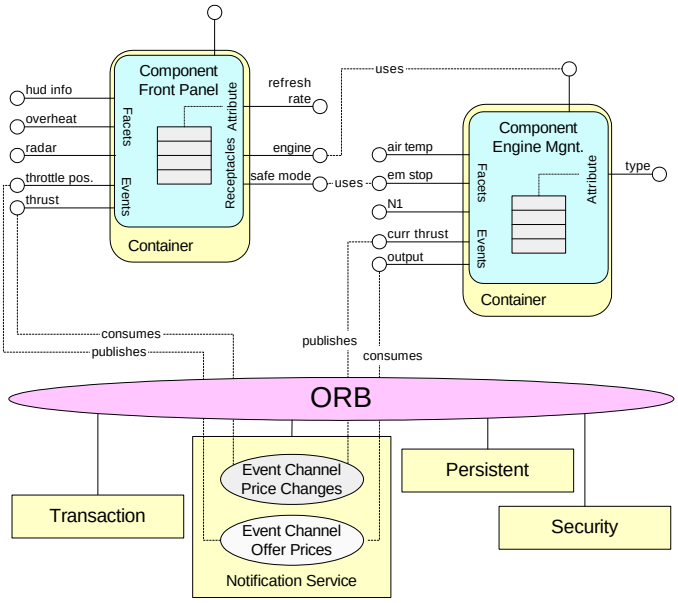

Figure 1: **Overview of the CCM Run-time Architecture**

export a set of interfaces to clients. Components can also express their intent to collaborate with other components by defining *ports*, which consist of the following types of interfaces:

- **Facets**, which define an interface that accepts method invocations from other components synchronously,
- **Receptacles**, which indicate dependencies on synchronous method interfaces provided by other components, and
- **Event sources/sinks**, which indicate a willingness to exchange messages with other components asynchronously.

A *container* provides the runtime environment for a component. It contains various pre-defined hooks that provide strategies, such as persistence, event notification, transaction, and security, to the component it manages. Each container manages one type of component and is responsible for initializing this component and connecting it to other components and ORB services. Developer-specified metadata is used to instruct the CCM deployment mechanism how to create these containers.

In addition to the building blocks outlined above, the CCM also standardizes component implementation, packaging, and deployment. The CCM Component Implementation Framework (CIF) helps generate the component implementation skeletons and persistent state management automatically using the Component Implementation Definition Language (CIDL). The CCM also extends the Open Software Description (OSD) [13], which is a vocabulary of XML defined by W3C to specify component packaging and assembly descriptors. OSD is used by the CCM deployment mechanisms to configure the component connections and containers declaratively.

The tools and mechanisms defined by the CCM collaborate together to address the limits described in Section 2.2. The CCM programming paradigm separates many common concerns of composing and provisioning reusable software components to build an application. This separation of concerns enables programmers to concentrate on the work at hand and separate the role of developers in the application development process. The CCM differentiates the following roles:

- **Component designers**, who define the component features by defining the component interfaces
- **Component implementors,** who develop component implementations
- **Component packagers**, who package component implementations with their default properties
- **Component assemblers**, who select component implementations and compose them into applications
- **System deployers**, who deploy component assemblies into component servers

Although the CCM specification has recently been finalized by the OMG, it still has not been fully incorporated into the core CORBA specification.[1] A number of CCM implementations are available based on the current draft [3], including

[1]The latest CORBA 3.0 specification [10] released by the OMG includes only changes in IDL definition and Interface Repository changes from the Component specification.

OpenCCM by the Universite des Sciences et Technologies de Lille, France, *K2 Containers* by iCMG, *MicoCCM* by FPX, and *CIAO* by the DOC groups at Washington University in St. Louis. The architectural patterns used in CCM are also used in other popular component middleware technologies, such as J2EE [14] and .NET.

2.4 Limitations with Component Middleware for DRE Systems

Large-scale DRE applications require seamless integration of many hardware and software systems. They also require complicated application provisioning where developers must connect numerous distributed or collocated subsystems together and define the functionality of each subsystem. Component middleware can reduce the software development effort for these types of systems by enabling application development through composition. Conventional component middleware frameworks, however, are designed with business applications in mind and do not yet support QoS provisioning for DRE applications. Developers are therefore forced to configure and control these mechanisms imperatively in their component implementations.

Although it is possible for component developers to take advantage of certain features in middleware or OS to implement QoS-enabled components by embedding certain QoS provisioning code in component implementations, most features are simply not possible to implement within component implementations. In particular, the following limitations restrict the effectiveness of conventional component models:

- QoS provisioning must be done end-to-end, *i.e.*, it needs to be applied to all interacting components. Implementing QoS provisioning logic internally to a component greatly hampers its reusability.
- Certain resources, such as thread pools in Real-time CORBA, can only be provisioned within an execution unit, *i.e.*, a component server. Since component developers often have no *a priori* idea of which other components a component implementation will collaborate, the component implementation is not the right level at which to perform QoS provisioning.
- 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 these types of QoS provisioning in isolation.
- Many QoS provisioning policies and mechanisms require the installation of customized ORB modules to work correctly. Some of these policies and mechanisms, such as high throughput and low latency, however, may be inherently incompatible. It is hard for QoS provisioning mechanisms implemented within components to foresee these incompatibility without knowing the end-to-end QoS requirements *a priori*.

In general, forcing QoS provisioning functionality into component implementations prematurely commits each implementation to a QoS provisioning scenario in a system's lifecycle. This tight coupling defeats one of the key benefits of component models: *separating component functionality from system management*. By creating dependencies between application components and the underlying component framework, component implementations become hard to reuse, particularly in DRE applications with stringent QoS requirements.

3 QoS Provisioning and Enforcement for DRE applications

In traditional DRE systems, code for provisioning and enforcing QoS properties is often spread throughout the software and tangled with the application logic. This tangling makes the DRE applications hard to maintain or to extend with new QoS mechanisms and behaviors. As discussed in Section 2.4, a key challenge in QoS provisioning 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 the application.

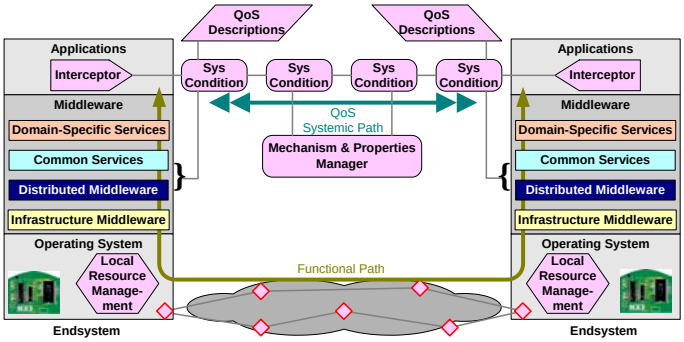

Figure 2: **Decoupling the Functional Path from the Systemic QoS Path**

Based on our experience developing dozens of research and production DRE systems over the past two decades, we have found that it is most effective to separate the programming of QoS concerns along the two dimensions shown in Figure 2 and discussed below:

Functional paths, which are flows of information between client and remote server applications. Distributed middleware is responsible to ensure that this information is exchanged efficiently, predictably, scalably, dependably, and se-

curely 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 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 2 enables these properties and resources to change independently, *e.g.*, over different distributed system configurations for the same application.

The architecture in Figure 2 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 than those customarily responsible for programming functional paths in DRE systems. In conventional component middleware, such as CCM that we described in Section 2.3, there are multiple software development roles, such as component designers, assemblers, and packagers. QoS-enabled component middleware identifies yet another development role called *qosketeer* [6] that 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.

The next 3 sections describe middleware technologies based on the architecture in Figure 2 that we have developed to

1. Statically provision QoS resources end-to-end to meet key requirements. Some DRE systems, such as avionics mission computing applications, require strict allocation of critical resources via static QoS provisioning.
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 in the face of rapidly changing mission requirements and environmental conditions.

4 Static Qos Provisioning and Enforcement via QoS-enabled Component Middleware and CIAO

4.1 Overview of Static QoS Provisioning

Static QoS provisioning refers to pre-determining the resources needed to satisfy certain QoS requirements and allocating the resources of a system before or during start-up time. Certain applications use static QoS provisioning because they require tightly bounded predictability for certain functionality in the systems. On other occasions, static QoS provisioning may be used for its simplicity.

To address the limitations of existing middleware outlined in Section 2.4, it is necessary to make QoS provisioning policies an integral part of component middleware to decouple QoS provisioning policies from component functionality. This separation of concerns relieves component developers from tangling the code to manage QoS resources with the component implementation. It simplifies QoS provisioning that cross-cut multiple interacting components to better ensure end-to-end QoS behavior. Specifically,

- To perform QoS provisioning end-to-end throughout a component middleware system robustly, the static QoS provisioning specifications should be decoupled from component implementations and specified instead in component composition metadata. This separation of concerns helps improve component reusability by preventing a premature commitment to specific QoS provisioning parameters.
- To provision QoS resources that need to be allocated in a component server, component assembly metadata need to be extended to allow allocation and configuration for these resources global to a component server, and be able to associate them with component instances that share these resources.
- Component assembly metadata must also be extended to provision QoS resources for component interconnections.
- To ensure a component server is configured with the mechanisms needed to support the provisioned QoS requirements, component assembly metadata need to be extended to include middleware modules that can configure component servers.

Figure 3 illustrates the types of static QoS provisioning that are necessary in large-scale DRE applications:

1. **CPU resources**, which need to be allocated to various competing tasks in a system to make sure these tasks finish on time,
2. **Communication resources**, which the middleware uses to pass messages around to "connect" distributed system together, and

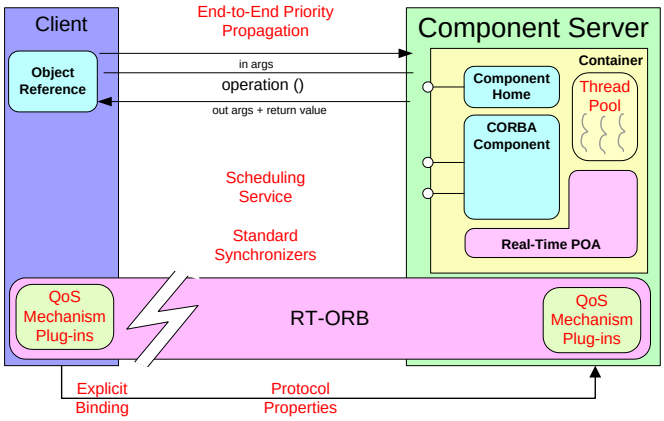

Figure 3: **Examples of Static QoS Provisioning**

3. **Distributed middleware configurations**, which are middleware plug-ins that a middleware framework uses to realize QoS assurance.

4.2 Static QoS Provisioning with CIAO

Figure 4 shows the key elements of the Component-Integrated ACE ORB (CIAO), which is a QoS-enabled implementation of CCM developed at Washington University, St. Louis by extending the TAO ORB [15]. TAO is an open-source, high-

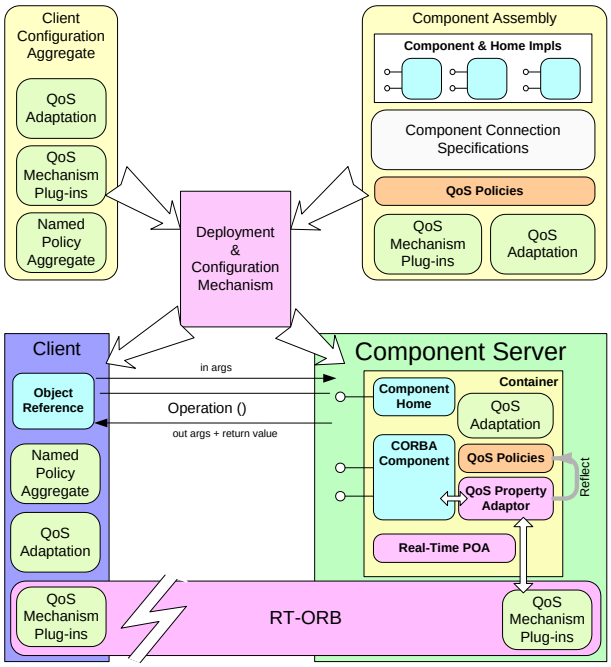

Figure 4: **Key Elements in CIAO**

performance, highly configurable Real-time CORBA ORB that implements key patterns [16] to meet the demanding QoS requirements of distributed systems. CIAO enhances TAO to simplify the development of DRE applications by enabling developers to statically provision QoS policies end-to-end declaratively when assembling a system.

To support the role of the qosketeer, CIAO makes the following extensions to the CCM to support static QoS provisioning:

Component assembly. A component assembly describes how components are composed into a system. We extend the notion of component assembly to include server-level QoS provisioning and implementations for required QoS supporting mechanisms. We also extend the assembly descriptor format to allow QoS provisioning at the component-connection level.

Client configuration aggregates. We define client-side configuration specifications to configure the client-side ORB for support of various QoS provisioning policies. Clients can then associate with named QoS provisioning policies defined in an aggregate, interact with servers, and provide end-to-end QoS assurance. Client configuration aggregates can be installed into a client ORB transparently in CIAO.

QoS-aware containers. They provide the centralized interface for managing provisioned component QoS policies and interacting with QoS assurance mechanisms required by the QoS policies.

QoS adaptations. CIAO also supports installation of meta-programming hooks which can be used to perform dynamic QoS provisioning.

To support these capabilities, CIAO extends the CCM packaging and deployment framework so that system developers can specify the necessary features in component assembly descriptors as various policies. These capabilities enable CIAO to statically provision the types of QoS resources outlined in Section 4.1 as follows:

1. **CPU resources** – These policies specify how to allocate CPU resources when running certain tasks, *e.g.*, priority model of a component instance;
2. **Communication resources** – These policies specify ways to reserve and allocate communication resources for component connections, *e.g.*, an assembly can demand a private connection between two critical components in the system, and reserve bandwidth for the connection using the RSVP protocol;
3. **Distributed middleware configuration** – These policies specify the required software modules that control the QoS mechanisms for:
   - **ORB configurations:** The ORB needs to know how to support the functionality required to enable higher level policies, *e.g.*, installing and configuring customized communication protocol.

- **Meta-programming mechanisms:** Software modules, such as those developed with the QuO Qosket middleware framework, which implement dynamic QoS provisioning and adaptation can be installed statically at system composition time via meta-programming mechanisms, such as smart proxies and interceptors [17].

System developers can use CIAO to decouple QoS provisioning functionality from component implementation and compose these static QoS provisioning requirements into a system at some later point of the development cycle.

5 Dynamic QoS Provisioning and Enforcement via QuO Adaptive Middleware and Qoskets

5.1 Overview of Dynamic QoS Provisioning

Dynamic QoS provisioning involves the allocation and management of resources at run-time to satisfy certain application QoS requirements. Certain events, such as fluctuations in resource availability or changes in QoS requirements, can trigger reevaluation and reallocation of resources. Middleware supporting dynamic QoS provisioning needs to detect changes in available resources and either reallocate resources of the system, or notify the application to *adapt* to the change.

As described in Section 1, conventional middleware tries to isolate applications functionality behavioral aspects, such as operation invocations, by abstracting these behavioral aspects under the interface interaction semantic. Although there are ways to implement dynamic QoS provisioning functionality in existing applications which use conventional middleware, naïve approaches can yield 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. It is therefore essential to separate the functionality of dynamic QoS provisioning from both middleware and application functionality.

Figure 5 illustrates the kinds of dynamic QoS provisioning abstractions and mechanisms that are necessary in large-scale DRE applications:

1. A design time formalism to specify the level of service desired by a client, the level of service an object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes.
2. A runtime mechanism to adapt application behavior based upon the current state of QoS in the system.

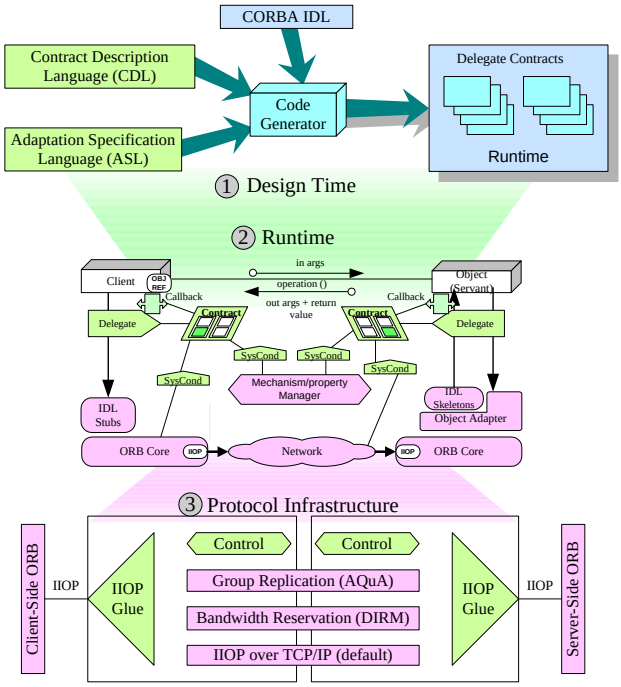

Figure 5: **Examples of Dynamic QoS Provisioning**

3. A set of interfaces to resources and mechanisms in the protocol infrastructure that need to be measured and controlled dynamically.

5.2 Overview of QuO

Quality Objects (QuO) [6] is an adaptive middleware framework developed by BBN Technologies that allows the DRE developer to use aspect-oriented software development [18] techniques to separate the concerns of QoS programming from application 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.

Figure 5 also 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 an object 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 objects. Each delegate provides an interface similar to that of the remote object 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 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 objects on behalf of the QuO clients using those server objects. In some cases, managed properties require mechanisms at lower levels in the protocol stack, such as replication or access control. To support this, QuO includes a gateway mechanism [19], which enables special-purpose transport protocols and adaptation below the ORB.

For more information about the QuO adaptive middleware, see [6, 19, 20, 21].

5.3 Qoskets: QuO Support for Reusing Systemic Behavior

One goal of QuO is to separate the role of the systemic QoS programmer from that of an application programmer. A complementary goal of this separation of programming roles is that systemic 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 an application program. To support this goal, we have defined *Qoskets* 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 distributed 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 applications, or in multiple ways within a single application, but typically deals with a single attribute (*e.g.*, performance, dependability, security)

Qoskets encapsulate the following systemic QoS aspects:

- **Adaptation policies**, as expressed in QuO contracts
- **Measurement and control**, as defined by system condition objects and callback objects
- **Adaptive behaviors**, as defined by ASL specifications, partially specified as templates until they are specialized to a functional interface, and by contract transitions and states.
- **QoS implementation**, as defined by Qosket methods.

A Qosket is a collection of the interfaces, contracts, system condition objects, callback objects, unspecialized adaptive behavior, and implementation code associated with a reusable piece of systemic behavior.

The general structure of Qoskets, objects they encapsulate, and interfaces they expose are illustrated in Figure 6. The

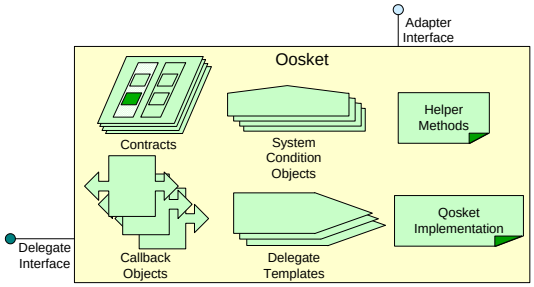

Figure 6: **Qoskets Encapsulate QuO Objects into Reusable Behaviors**

two interfaces that qoskets expose correspond to these two use cases:

- **The adapter interface**, which is an application programmer 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 conveniently specified in the QuO ASL language. The adaptation strategies of the delegate are conveniently encapsulated, and woven into the application using code generation techniques.

6 Total QoS provisioning via CIAO and Qoskets

As discussed in Section 5.3, Qoskets provide abstractions for dynamic QoS provisioning and adaptive behaviors. However, the current implementation of Qoskets in QuO requires application developers to modify their application code manually to "plug in" the behavior into existing applications. Instead

of 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.

Conversely, although 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 4.2, CIAO does not provide an abstraction to model, define, and specify dynamic QoS provisioning. We can take advantage of CIAO's capability to transparently configure Qoskets into component servers and 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 of CIAO enables the composition of qoskets into applications as part of component assemblies. As shown in Figure 7, CIAO installs a qosket

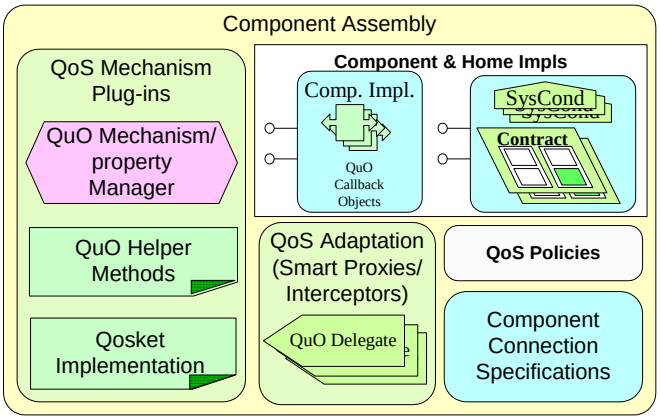

Figure 7: **Composing a qosket using CIAO**

using using the following mechanisms:

- QuO delegates can be implemented as smart proxies or portable interceptors [17] and injected into component servers using assembly descriptors and the client-side configuration aggregates described in Section 4.2;
- Likewise, developers can specify 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 objects can be implemented as CCM components and be assembled into component servers.

While the use of CIAO to compose Qoskets into component assemblies simplifies retrofitting, a significant problem remains open: *component cross-cutting*. Qoskets are adept at separating concerns between systemic QOS properties and application logic, as well as implementing limited cross-cutting between a single client/object pair. Neither Qoskets nor CIAO currently provides the ability to cross-cut application components, however. Many QOS-related adaptations will need to modify the behavior of several components at once, possibly in a distributed way. Some form of dynamic aspect-oriented programming might be used to handle this, but this research is ongoing.

7 Concluding Remarks

Component middleware [12] has emerged as a promising solution to many limitations with object-oriented application frameworks. This type of middleware consists of reusable software artifacts that can be distributed or collocated throughout a network. Existing component middleware, however, does not address DRE application's QoS provisioning needs as they often spread beyond component boundary. A QoS-enabled middleware is necessary to separate the QoS provisioning concerns from application functional concerns.

This paper describes how the standard CCM specification is being augmented by the CIAO middleware to support static QoS provisioning that pre-allocates resources for DRE application. We also describe how BBN's QuO Qoskets middleware framework provides powerful abstractions that help define and implement reusable dynamic QoS provisioning behaviors. By combining QuO Qoskets and CIAO, we are providing an integrated QoS provisioning solution for DRE applications. We are applying the total QoS provisioning solution to several research projects to demonstrate the effectiveness of the solution. These projects include composing mission critical software systems, such as avionics mission computing systems and a video distribution system for unmanned air vehicles (UAV).

References

[1] 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*, vol. 45, no. 10, Oct. 2002.

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

[3] Object Management Group, *CORBA 3.0 New Components Chapters*, OMG TC Document ptc/2001-11-03 edition, Nov. 2001.

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

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

[6] John A. Zinky, David E. Bakken, and Richard Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

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

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

[9] IBM, "MQSeries Family," www-4.ibm.com/software/ts/mqseries/, 1999.

[10] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0 edition, June 2002.

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

[12] George T. Heineman and Bill T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, Reading, Massachusetts, 2001.

[13] Arthur van Hoff, Hadi Partovi, and Tom Thai, "The Open Software Description Format (OSD)," http://www.w3c.org/TR/NOTE-OSD.html.

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

[15] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[16] 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.

[17] 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*, vol. 39, no. 10, Oct. 2001.

[18] 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.

[19] Richard E. Schantz, John A. Zinky, David A. Karr, David E. Bakken, James Megquier, and Joseph P. Loyall, "An object-level gateway supporting integrated-property quality of service," in *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*, May 1999.

[20] Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David Karr, Rodrigo Vanegas, and Ken R. Anderson, "QoS Aspect Languages and Their Runtime Integration," *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Syste,s for Sclable Components*, May 1998.

[21] Rodrigo Vanegas, John A. Zinky, Joseph P. Loyall, David Karr, Richard E. Schantz, and David E. Bakken, "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*, September 1998.