

Total Quality of Service Provisioning in Middleware and Applications

Nanbor Wang^a, Douglas C. Schmidt^b, Aniruddha Gokhale^c, Christopher D. Gill^a, Balachandran Natarajan^c,
Craig Rodrigues^d, Joseph P. Loyall^d, and Richard E. Schantz^d

^aDept. of Computer Science and Engineering, Washington University
One Brookings Drive, St. Louis, MO 63130, USA

^bDept. of Electrical and Computer Engineering, University of California
616E Engineering Tower, Irvine, CA 92697, USA

^cInstitute for Software Integrated Systems, Vanderbilt University
Box 1829, Station B, Nashville, TN 37235, USA

^dBBN Technologies, 10 Moulton Street, Cambridge, MA 02138, USA

Commercial off-the-shelf (COTS) middleware, such as Real-time CORBA, is gaining acceptance in the distributed real-time and embedded (DRE) community. Existing COTS specifications, however, do not effectively separate quality of service (QoS) policy configurations and adaptations from application functionality. DRE application developers therefore often intersperse code that provisions resources for QoS guarantees and program adaptation mechanisms throughout DRE applications, making it hard to configure, validate, modify, and evolve complex DRE applications. This paper illustrates how (1) standard component-based middleware can be enhanced to flexibly compose static QoS provisioning policies with application logic, (2) adaptive middleware capabilities enable developers to abstract and encapsulate reusable dynamic QoS provisioning and adaptive behaviors, and (3) 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 OMG CORBA 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. These middleware technologies have historically been applied to enterprise applications. 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 pro-

cess control systems and avionics mission computing systems.

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 class of middleware that enables reusable services to be composed, configured, and installed to create applications rapidly and robustly. Examples of COTS component middleware include the CORBA Component Model (CCM) [1], Java 2 Enterprise Edition (J2EE) [2], and the Component Object Model (COM) [3], which use different APIs, different protocols, and different component models.

Most DRE applications have stringent QoS requirements that must be satisfied simultaneously in real-time. Example QoS requirements include processing resource 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. 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. 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 business-oriented QoS requirements, such as data persistence, encryption, and transactional support. They therefore do not effectively enforce the stringent QoS requirements of DRE applications. What is 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 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: First, 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) [1] called the *Component-Integrated ACE ORB* (CIAO), which is being developed at Washington University, St. Louis. Second, 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) [4] middleware developed by BBN Technologies. The discussion concentrates on how major elements in QuO

are defined, developed, and used to implement dynamic QoS provision and adaptive behaviors. Finally, we discuss how we integrate CIAO and Qoskets to enable composition of both static QoS provisioning and dynamic adaptive QoS assurance in DRE applications. In particular, we focus on how CIAO uses Qoskets to weave together software elements to create an integrated QoS-enabled component model that offers a total QoS provisioning solution for DRE applications.

2. Component Middleware: A Powerful Approach to Building DRE Applications

This section presents an overview of component middleware and 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 [5]. Middleware's primary role is to bridge the gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate. When implemented properly, middleware helps to shield application developers from low-level platform details, provides standard higher-level interfaces to manage system resources, and amortizes development costs through reusable frameworks. 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 [6] 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. CORBA is the only COTS middleware that supports multiple languages and has made substantial progress

in satisfying the QoS requirements of DRE systems. CORBA has the following limitations however:

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 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 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 server and its QoS policies, providing common services (such as notification or naming services), and managing the runtime environment of each component. The lack of a generic component server standard yields tightly coupled, *ad-hoc* server implementations, which increase the complexity of software upgrades and reduce the reusability and flexibility of CORBA-based applications.

2.3. Promising Solution: Component Middleware

In recent years, *component middleware* [7] 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 other only through well-defined interfaces and (2) defining the standard mechanisms to compose and execute components in generic component servers. We base our work on the CCM to take advantage of the DRE-related CORBA specifications, such as CORBA Messaging and Real-time CORBA, that enforce and support QoS requirements of DRE systems.

Figure 1 shows an overview of the runtime architecture of the CCM model. *Components* are implementation entities that export a set of interfaces to clients. Components can also express their intent to collaborate with other components by defining *ports* that specify how components interact.

The CCM also defines a *container* mechanism that provides a component runtime environment via pre-

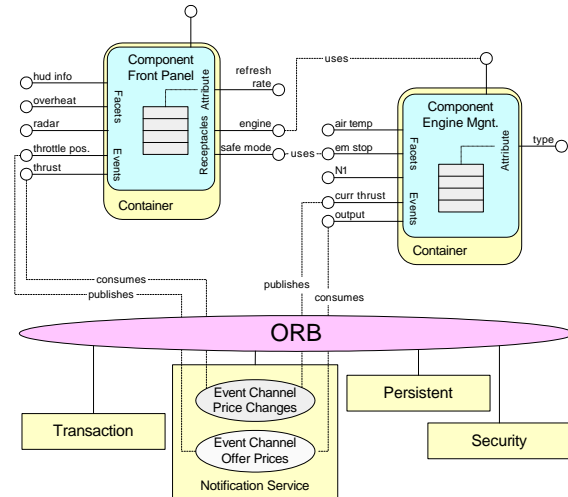


Figure 1. Overview of the CCM Run-time Architecture

defined hooks to control systemic strategies, such as event notification, transaction, and security. The CCM also standardizes a component implementation framework, packaging and deployment tools and mechanisms that abstract and automate component implementations, application compositions, and system deployment. The CCM programming paradigm separates concerns of composing and provisioning reusable software components into the following development 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, and, **System deployers**, who deploy component assemblies into component servers.

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, fail to support abstractions for QoS provisioning required by DRE applications, which forces developers to control QoS ensuring mechanisms imperatively in their component implementations.

Moreover, many QoS capabilities cannot be implemented solely within a component due to the following limitations:

- QoS provisioning must be done end-to-end, *i.e.*, it needs to be applied to all interacting components. Implementing QoS provisioning logic internally in a 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 which other components a component implementation will collaborate, the component implementation is not the right level 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 these types of QoS provisioning in isolation.
- Many QoS provisioning policies and mechanisms require the installation of customized ORB modules to work correctly. However, some of these policies and mechanisms, such as high throughput and low latency, may be inherently incompatible. It is hard for QoS provisioning mechanisms implemented within components to foresee these incompatibilities without knowing the end-to-end QoS requirements *a priori*.

In general, forcing QoS provisioning functionality into a component prematurely commits each implementation to a specific 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 for DRE applications with stringent QoS requirements.

3. QoS Provisioning and Enforcement for DRE applications

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.

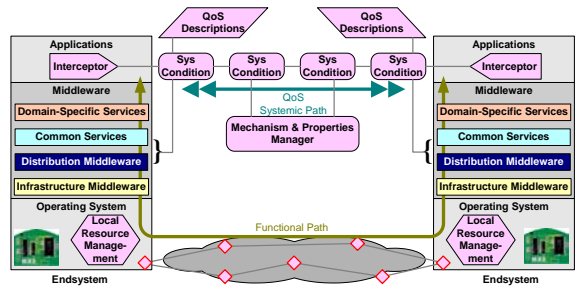


Figure 2. Decoupling the Functional Path from the Systemic QoS Path

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 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 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. Beside the multiple software development roles we previously identified in Section 2.3, QoS-enabled component middleware identifies yet another development role which we term *qosketeer* [4] 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 preallocation 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

This section presents an overview of static QoS provisioning and illustrates how static QoS provisioning can be composed into a component-based application.

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 anticipate unchanging demands and require tightly bounded predictability for certain functionality in the systems. In addition, static QoS provisioning is often the simplest solution available.

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 ensure proper end-to-end QoS behavior.

More specifically, to provision end-to-end QoS throughout a component middleware system robustly and improve component reusability, the static QoS provisioning specifications should be decoupled from component implementations and specified instead in component composition metadata. For QoS resources that must be allocated globally in an application, component assembly metadata must be expanded to allocate and configure these resources and associate them with component instances or component connections. Moreover, to ensure a component server is equipped with the mechanisms needed to support the provisioned QoS requirements, component assembly metadata should include middleware modules that enable the control and configuration of these resources.

4.2. Static QoS Provisioning with CIAO

Figure 3 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 by extending the TAO ORB [8]. TAO is an open-source, high-

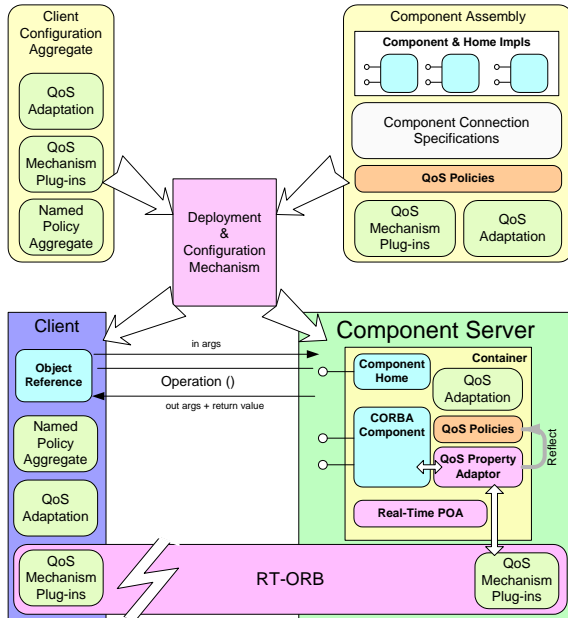


Figure 3. Key Elements in CIAO

performance, highly configurable Real-time CORBA ORB that implements key patterns [9] 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:

CPU resources – These policies specify how to allocate CPU resources when running certain tasks, *e.g.*, priority model of a component instance;

Communication resources – These policies specify ways to reserve and allocate 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;

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 [10].

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

5. Dynamic QoS Provisioning and Enforcement

This section presents an overview of dynamic QoS provisions and describes how middleware modules are being used to manage dynamic QoS provisioning for applications.

5.1. Overview of Dynamic QoS Provisioning

Dynamic QoS provisioning involves the allocation and management of resources at run-time to satisfy 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, or notify the application to *adapt* to the change.

As described in Section 1, conventional middleware tries to isolate an application's functional behavioral aspects, such as operation invocations, by abstracting these behavioral aspects behind interface interaction semantics. Although it is possible to implement dynamic QoS provisioning functionality in existing applications which use conventional middleware, it requires working around the current structures intended to simplify the development of distributed systems and often becomes counterproductive. *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. It is therefore essential to separate the functionality of dynamic QoS provisioning from both lower level distribution middleware and application functionality.

Figure 4 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 capability to adapt application behavior based upon the current state of QoS in the system.
3. A set of interfaces to resources and mechanisms in the protocol infrastructure that need to be

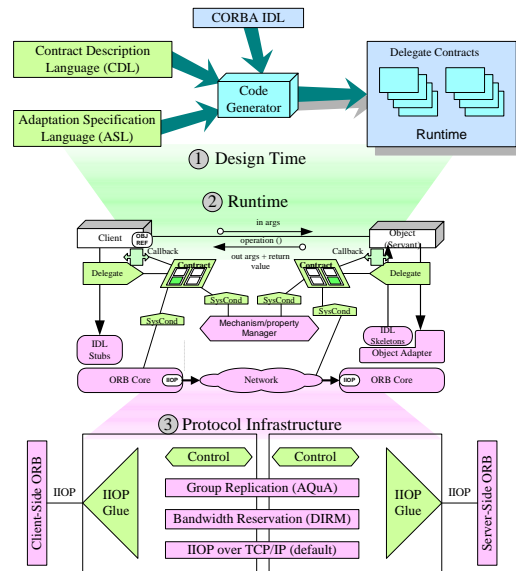


Figure 4. Examples of Dynamic QoS Provisioning measured and controlled dynamically.

5.2. Overview of QuO

Quality Objects (QuO) [4] is an adaptive middleware framework developed by BBN Technologies that allows the DRE developer to use aspect-oriented software development [11] 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 4 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, and; **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. QuO provides a gateway mechanism [12] that enables special-purpose transport protocols and adaptation below the ORB.

For more information about the QuO adaptive middleware, see [4,12–14].

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*[15] as a unit of encapsulation and reuse of systemic behavior in QuO applications. 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 Adaptation Specification Language (ASL) specifications.
- **QoS implementation**, as defined by Qosket methods.

As shown in Figure 5, 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.

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”

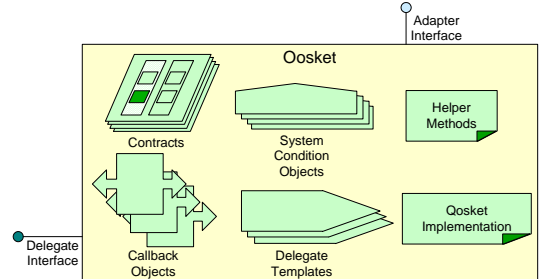


Figure 5. Qoskets Encapsulate QuO Objects into Reusable Behaviors

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 Fig-

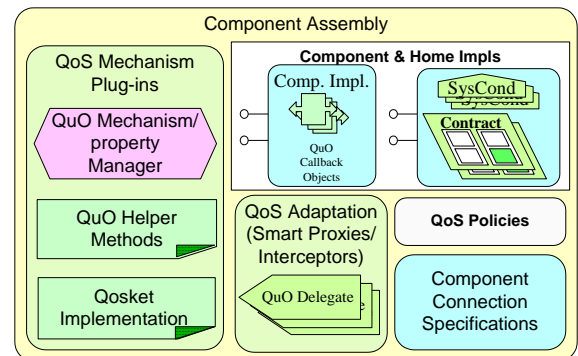


Figure 6. Composing a Qosket using CIAO

ure 6, CIAO installs a Qosket using the following mechanisms:

- QuO delegates can be implemented as smart proxies or portable interceptors [10] and injected into component servers using assembly descriptors and the client-side configuration aggregates described in Section 4.2;
- 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 objects can be implemented and assembled as CCM components into component servers.

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/object 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 [16].

7. Concluding Remarks

Component middleware [7] has emerged as a promising solution to many limitations with object-oriented middleware and application frameworks. Component middleware consists of reusable software artifacts that can be distributed or collocated throughout a network. Existing component middleware, however, does not address end-to-end QoS provisioning needs of DRE applications, which spread beyond component boundaries. QoS-enabled middleware is therefore necessary to separate QoS provisioning concerns from application functional concerns.

This paper describes how CIAO is augmenting the standard CCM specification to support static QoS provisioning by pre-allocating 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 aerial vehicles (UAV).

REFERENCES

1. Object Management Group, CORBA Components, OMG Document formal/2001-11-03 Edition (Jun. 2002).
2. Sun Microsystems, JavaTM 2 Platform Enterprise Edition, <http://java.sun.com/j2ee/index.html> (2001).
3. D. Box, Essential COM, Addison-Wesley, Reading, MA, 1998.
4. J. A. Zinky, D. E. Bakken, R. Schantz, Architectural Support for Quality of Service for CORBA Objects, Theory and Practice of Object Systems 3 (1) (1997) 1–20.
5. R. E. Schantz, D. C. Schmidt, Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications, in: J. Marciniak, G. Telecki (Eds.), Encyclopedia of Software Engineering, Wiley & Sons, New York, 2002.
6. Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.6.1 Edition (May 2002).
7. G. T. Heineman, B. T. Councill, Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley, Reading, Massachusetts, 2001.
8. D. C. Schmidt, D. L. Levine, S. Mungee, The Design and Performance of Real-Time Object Request Brokers, Computer Communications 21 (4) (1998) 294–324.
9. D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2, Wiley & Sons, New York, 2000.
10. N. Wang, D. C. Schmidt, O. Othman, K. Parameswaran, Evaluating Meta-Programming Mechanisms for ORB Middleware, IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies 39 (10).
11. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming, 1997.
12. R. E. Schantz, J. A. Zinky, D. A. Karr, D. E. Bakken, J. Megquier, J. 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), 1999.
13. J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. Karr, R. Vanegas, K. R. Anderson, QoS Aspect Languages and Their Runtime Integration, Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components.
14. R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, D. 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.
15. R. Schantz, J. Loyall, M. Atighetchi, P. Pal, Packaging Quality of Service Control Behaviors for Reuse, in: Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), IEEE/IFIP, Crystal City, VA, 2002.
16. D. I. T. Office, The Programmable Composition of Embedded

Software (PCES) Program, <http://www.darpa.mil/ito/research/pces/index.html>.