Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation

Nanbor Wang
Kirthika Parameswaran
{nanbor,kirthika}@cs.wustl.edu
Dept. of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130

Michael Kircher
Michael.Kircher@mchp.siemens.de
Siemens ZT
Munich
Germany

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

This paper was submitted to the COMPSAC 2000 conference, Taipei, Taiwan, October 25-27, 2000.

Abstract

Although existing CORBA specifications, such as Real-time CORBA and CORBA Messaging, address many end-to-end quality-of-service (QoS) aspects, they do not define strategies for configuring these QoS aspects into applications flexibly, transparently, and adaptively. Therefore, application developers must make these configuration decisions manually and explicitly, which is tedious, error-prone, and often suboptimal. Although the recently adopted CORBA Component Model (CCM) does define a standard configuration framework for packaging and deploying software components, conventional CCM implementations focus on functionality rather than adaptive quality-of-service, which makes them unsuitable for next-generation applications with demanding QoS requirements.

This paper presents three contributions to the study of middleware for QoS-enabled component-based applications. It outlines reflective middleware techniques designed to adaptively (1) select optimal communication mechanisms, (2) manage QoS aspects of CORBA components in their containers, and (3) (re)configure selected component executors dynamically. Based on our ongoing research on CORBA and the CCM, we believe the application of reflective techniques to component middleware will provide a dynamically adaptive and (re)configurable framework for COTS software that is well-suited for the QoS demands of next-generation applications.

1 Introduction

Emerging trends and challenges: Distributed applications are increasingly being developed via the standard interfaces, protocols, and services defined by distributed object computing (DOC) middleware, such as CORBA [1] or Java RMI [2]. DOC middleware that allows clients to invoke operations on remote objects without concern for where the object resides [3]. In addition, DOC middleware shields applications from non-portable details related to the OS/hardware platform they run on and the communication protocols and networks used to interconnect distributed objects.

Next-generation applications require DOC middleware that is adaptive and configurable, as well as efficient, predictable, and scalable. For instance, the demand for embedded multimedia applications is growing rapidly and hand-held devices, such as PIMs, Web-phones, Web-TVs, and Palm computers, running multimedia applications, such as MIME-enabled email and Web browsing, are becoming ubiquitous [4]. Ideally, these embedded multimedia applications should be *configured automatically* using standard DOC middleware components, rather than *programmed manually* from scratch. Meeting the QoS demands of next-generation applications requires the resolution of many research challenges, however, such as adapting to frequent bandwidth changes and disruptions in the established connections, maintaining cache consistency, and addressing various restrictions on memory footprint size and power consumption [5].

DOC middleware based on CORBA should be well-suited to provide the core communication middleware for the next-generation distributed applications outlined above. For instance, recent additions to the CORBA specification, such as Real-time CORBA [6] and CORBA Messaging [7], address many end-to-end quality-of-service (QoS) aspects. These specifications standardize interfaces and policies for defining and controlling various types of application QoS aspects.

Historically, however, the standard CORBA specification has not addressed component implementation or configuration issues effectively. For example, the CORBA 2.x [1] specification did not standardize interfaces to (1) initialize and deploy services dynamically or (2) enable different service implementations to interact portably with each other via standard interfaces. Moreover, many "cross-cutting" [8] service implementation aspects, such as memory and bandwidth management, concurrency, dependability, security, and power management, are tightly coupled into the application structure and behavior of CORBA servants. As a result, programming applications directly using standard CORBA 2.x APIs has often yielded (1) brittle servant implementations that are hard to optimize, maintain, and enhance and (2) overly static or non-standardized mechanisms for bootstrapping and (re)configuring ORB components and services [9].

To address these problems, therefore, the OMG adopted the CORBA Component Model (CCM) specification [10]. The CCM defines a framework for generating distributed servers into which developer can configure custom component logic. In theory, the adoption of the CCM should reduce the effort required to integrate portable components that implement services and applications. Moreover, the CCM should simplify the reconfiguration and replacement of existing application services by standardizing interconnections among components and interfaces.

In practice, however, the CCM standard and implementations are as immature today as the underlying CORBA standard and ORBs were three to four years ago. For instance, CCM implementations are not yet particularly efficient, predictable, or scalable. Moreover, commercial CCM vendors are largely targeting the requirements of e-commerce, workflow, report generation, and other general-purpose business applications. The middleware requirements of these applications focus on functionality and interoperability, however, with little emphasis on assurance of, or control over, mission-critical QoS aspects, such as timeliness, precision, dependability, minimal footprint, and power consumption [11]. As a result, it is not feasible to use contemporary off-the-shelf CCM implementations for applications with demanding QoS requirements.

Solution approach → reflective middleware: Our prior research on CORBA middleware has explored many efficiency, predictability, and scalability aspects of ORB endsystem design, including static [12] and dynamic [13] scheduling, event processing [14], I/O subsystem [15] and pluggable protocol [16] integration, synchronous [17] and asynchronous [18] ORB Core architectures, systematic benchmarking of multiple ORBs [19], and optimization principle patterns for ORB performance [20]. This paper focuses on another key dimension in the ORB endsystem design space: *applying reflective middleware techniques to implement QoS-enabled versions of the CCM*.

Reflective middleware is a term that describes a loosely organized collection of technologies designed to manage and control hardware/software system resources based on mounting R&D experience with distributed applications and systems [21]. Reflective middleware techniques enable dynamic changes in application behavior by adapting core software and hardware mechanisms both with or without the knowledge of applications or end-users [22]. Figure 1 illustrates the key architectural focal points where we are applying reflective middleware techniques to improve the configurability and adaptivity of QoS-enabled CCM implementations. In this paper, we illustrate how reflective middleware techniques are being applied to improve the adaptivity of the following CORBA and CCM mechanisms.

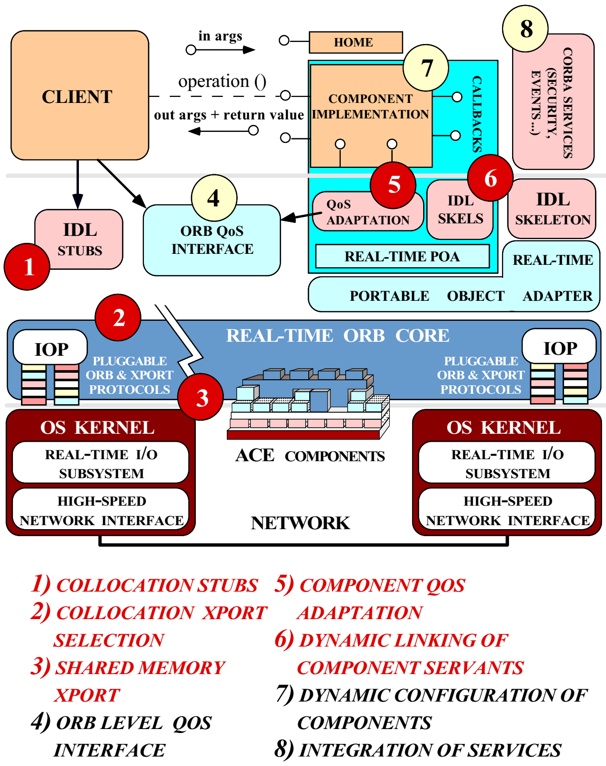


Figure 1: Focal Points of Reflective Techniques for CORBA Middleware

• **Selecting optimal communication mechanisms:** To present a homogeneous programming model for application developers, CORBA hides the location of objects from client applications. By examining an object's location reflectively, however, a CORBA ORB can select an optimal communication mechanism automatically when it *binds* an object reference [23]. To avoid violating the CORBA object model, however, this selection must occur without direct application intervention so that middleware performance and predictability can be optimized transparently. In particular, robust and automated ORB collocation support [20] is necessary since the

CCM encourages complex, dynamically changing object composition relationships [24].

• **Managing QoS aspects of components in their containers:** In the CCM, a *container* manages the implementation of a component by encapsulating it within a run-time environment that provides certain services, such as security, event notification, and transactions. In addition, CCM containers should be extended to manage component implementation QoS aspects, such as memory and bandwidth management, concurrency, dependability, security, and power management. Such extensions would allow ORB endsystems to support dynamic QoS configuration since they could inspect and adjust a component's QoS aspects via its container. By factoring QoS adaptation policies and mechanisms into containers, components developers can defer the selection of a component's QoS requirements until run-time, thereby enhancing component flexibility and adaptability.

• **Dynamically (re)configuring selected parts of component implementations:** Next-generation applications will increasingly run in wireless and mobile network configurations where there may be no *a priori* knowledge of (1) the appropriate implementation of service components, (2) the optimal partitioning of service components onto network nodes and (3) Activation of components need to occur in real-time which means that the initialization process should not be a bottleneck. Thus, on-demand linking/unlinking mechanisms are necessary to (re)configure component implementations dynamically. The lifecycle for linking/unlinking of these components must be optimized using reflective middleware techniques to minimize footprint, prolong battery life, maximize extensibility, and meet key application QoS requirements more adaptively.

We are applying these reflective middleware techniques at various levels, ranging from the ORB Core up to CORBA Component Model services. The vehicle for this research is TAO [12], which is an open-source[1], CORBA-compliant ORB designed to support applications with demanding QoS requirements. Figure 1 illustrates how CORBA components, features, and services are being integrated into the TAO ORB endsystem.

Paper organization: The remainder of this paper is organized as follows: Section 2 outlines the key features of the CORBA Component Model (CCM); Section 3 (1) motivates key challenges faced when designing CCM implementations to support QoS-enabled applications and (2) outlines the reflective middleware techniques we are applying to address these challenges; Section 4 describes empirical results from some of our efforts to date; Section 5 compares our approach with related work; and Section 6 presents concluding remarks.

[1]The source code and documentation for TAO can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

2 Overview of the CORBA Component Model Specification

This section presents an overview of the CORBA Component Model (CCM) architecture, focusing on how entities in CCM relate to reflective techniques. For complete coverage, please see [10].

Components: A component is a basic CORBA *meta-type*, *i.e.*, it can be referenced by multiple object references of different types. Each component has a set of *supported interfaces* that it inherits from IDL interfaces or another component. These interfaces, collectively called the component's supported interfaces, define the component's equivalent interface. A component encapsulates a design entity and is identified by a *component reference*. For "component-unaware" clients, component references behave identically to regular object references, *i.e.*, clients can invoke operations defined in supported interfaces. As shown in Figure 2, components interact with external entities, such as services provided by the ORB or other components, via the following *port* mechanism:

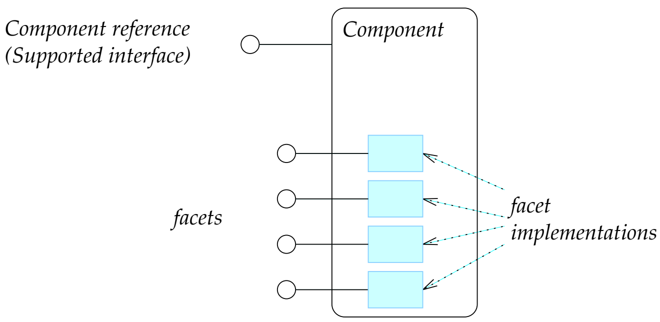


Figure 2: The Architecture of a CCM Component

• **Facets:** A *facet*, also called a *provided interface*, is an interface contract exposed by a component. Facets are similar to component *interfaces* in Microsoft's Component Object Model (COM) [25], in that they allow a component to support *unrelated interfaces*. Unrelated interfaces exposed through facets need not be related through inheritance to the component's supported interfaces.

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

• **Component home:** The CCM specification introduces a new keyword, `home`, which supports *component homes*. A component home provides factory method [26] for a component, which are responsible for creating or finding instances of components. Each component home manages exactly one type of component. Home interfaces can optionally use a *key* to manage instances of the managed component. Each key maps to an instance of the component. Conversely, for a *keyless* home interface, invoking the factory method simply creates a new instance of the managed component type.

Component Implementation Framework (CIF): The CORBA *Component Implementation Framework* (CIF) defines the programming model for managing the persistent states of components and constructing component implementations. The CCM specification defines a declarative language, the *Component Implementation Definition Language* (CIDL), to describe implementations and persistent states of components and component homes. As shown in Figure 3, the CIF

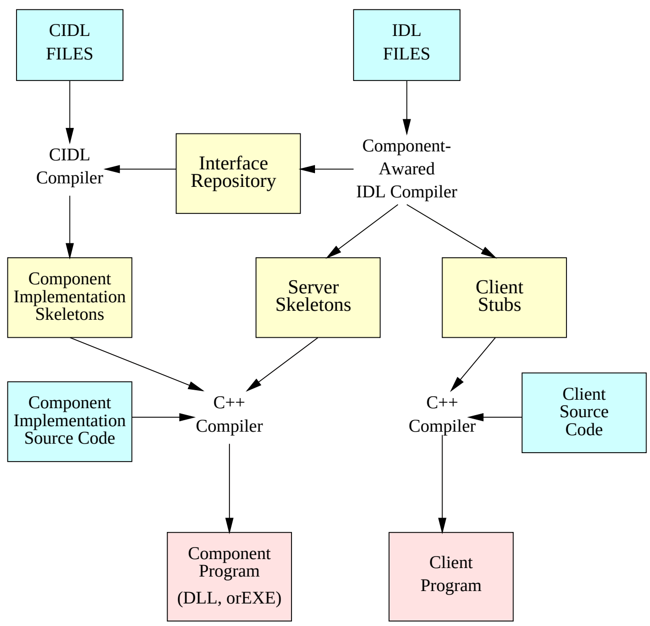


Figure 3: Using IDL and CIDL for component implementation

uses the CIDL descriptions to generate programming skeletons that automate basic behaviors of components, such as navigation, identity inquiries, activation, state management, transactions, and security. Many of these standard interfaces, *e.g.*, navigation, identity inquiries, and activation, provide the ways to *introspect* components and the mechanism for CCM to be implemented reflectively.

Implementations generated by a CIDL compiler are called *executors*. Executors contain the aforementioned auto-generated implementations and provide hook methods [26] that allow component developers to add custom component-specific logic. Executors can be packaged in dynamically linked libraries (DLL)s and installed in the container of a *component server* that support a particular target platform/language.

Containers: The CCM container programming model defines a set of APIs that simplify the task of developing and/or configuring CORBA applications. A container encapsulates a component implementation and uses these APIs to provide a run-time environment for the component that it manages. Figure 5 on page 7 shows the architecture of the container programming model.

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

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

• **Container APIs:** These include the *internal interfaces* that the component can invoke to access to the services provided by the container and the *callback interfaces* that the container can invoke on the component.
Through the collaboration of these interfaces, a container provides its managed component access to its POA and the services supported by the ORB.

CCM containers also manage the lifetime of component servants. Four types of servant lifetime policies – *method*, *session*, *component*, and *container* – control the timing of activating and passivating components. *Method* and session policies causes `ServantLocator`s to activate and passivate component on every method invocation/session, whereas *component* and *container* policies defer the servant lifetime policies to components and containers, respectively.

There are two types of container interfaces: (1) *session* container interfaces for transient components and (2) *entity* container interfaces for persistent components. The *CORBA Usage Model* specifies the required interaction pattern between a container, its POA, and CORBA Services (such as Notification or Transaction) by specifying the interfaces' transientness/persistency and cardinality of servant→OID mapping.

The *component category* defines the legal combinations of the container API types and the CORBA usage models. By specifying a container's component category along with other policies, component developers can specify a wide range of configuration options in the CIF. The CIF then generates the component implementation with proper strategies for QoS aspects, such as persistence, event notification, transaction, security. Thus, when combined with OMG's Real-time

CORBA [27] and Messaging [7] specifications, CCM containers provide application developers with a model for creating, specifying, and partitioning various run-time QoS aspects, such as end-to-end priority and connection bandwidth utilization, for components in real-time systems.

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

ORB extension → locality constrainted interfaces: Historically, locality-constrained interfaces have been limited to ORB-defined types, such as `CORBA::NVList`, `CORBA::Request`, and `CORBA::TypeCode`, and were often defined using so-called pseudo-IDL (PIDL) [28]. To support the component model efficiently, and to eliminate the need for PIDL, the CCM specifies a new IDL keyword, called `local`, which standardizes the definition of *locality constrained* interfaces.

As its name implies, a local interface is only valid in the process in which it is instantiated. Thus, it cannot be externalized to or invoked from other processes. Adding standard support for locality constrained interfaces to CORBA is particularly important for server-centric components because it helps improve performance and minimize memory footprint.

3 Applying Reflective Middleware Techniques to Resolve Key Design Challenges for QoS-enabled CCM Implementations

This section describes the key research challenges that CCM developers must address to support QoS-enabled applications and outlines the reflective middleware techniques we are applying to address these challenges.

3.1 Challenge 1: Achieving QoS-enabled Location Transparency Adaptively

Context: Location transparency is an important feature of the CORBA programming model. It allows applications to invoke operations via well-defined interfaces, without having to be concerned with where the target components reside.

Problem: A straightforward strategy for implementing location transparency is to treat all operations as remote invocations that are sent via IIOP over TCP/IP. This strategy imposes unnecessary communication overhead, however, when an object resides within the same host or the same address space as the client. Thus, quality ORBs must determine the actual location of a target object to optimize performance, while shielding developers from these details to simplify programming.

As shown in [29], an ORB can improve performance substantially by determining the location of target objects and then invoking operations using the most efficient communication mechanism. For example, when invoking an operation on a target component collocated on the same host, an ORB should choose a communication mechanism, such as shared memory, that is more efficient than "loopback" TCP/IP. This selection process is called the "collocation optimization."

It is important, however, that collocation optimizations be implemented in a "QoS-enabled" manner. In another words, applying collocation optimizations should not interfere with QoS mechanisms provided by the underlying ORB endsystem. For instance, two real-time ORB endsystem mechanisms defined by the Real-time CORBA specification are *prioritized scheduling* and *QoS-enabled communication channels* [27]. Prioritized scheduling ensures that applications requiring QoS support receive enough resources to meet their deadlines. QoS-enabled communication channels ensure the ORB endsystem's communication infrastructure allocates sufficient bandwidth, CPU, and memory resources to satisfy application QoS requirements end-to-end.

Solution → Reflective selection of optimal communication mechanisms: To select an optimal communication mechanism, an ORB must apply collocation optimizations *reflectively* at run-time. In general, these optimizations must be invisible to ORB users to avoid violating CORBA's object model transparency. Moreover, although certain collocation optimization mechanisms (such as direct function calls or shared memory) may be *faster* than other communication mechanisms (such as TCP loopback or message queuing), a QoS-enabled ORB must select a communication mechanism based on their client/object QoS requirements. For example, to avoid incurring priority inversion, a reflective QoS-enabled collocation optimization mechanism could establish multiple connections to partition ORB communication between client and server threads with different QoS requirements.

When object migration occurs, an ORB must re-select the optimal communication mechanism. To support migration, an operation invocation will receive a LOCATION_FORWARD message and a new object reference will be examined. As with the original binding, the ORB should determine the appropriate communication mechanism reflectively, taking into account the QoS characteristics of the various clients and objects involved in the migration.

Applying reflective collocation mechanisms in TAO: Figure 4 illustrates how TAO is designed to support reflective col-

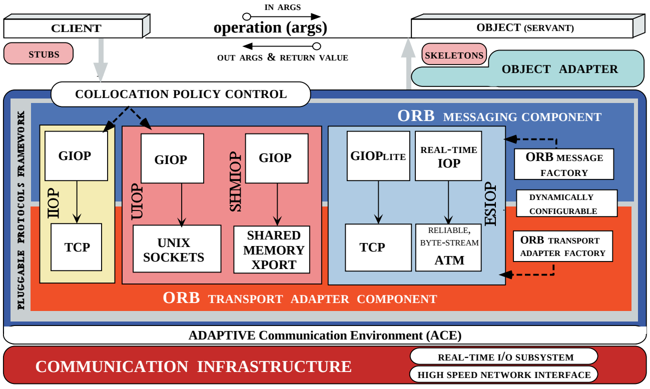


Figure 4: Reflective Selection of Optimal Communication Mechanisms in TAO

location mechanisms. TAO determines an object's location when it *binds* an object reference [23] or receives a LOCATION_FORWARD message. If the object is local to the process, TAO also considers the QoS policies associated with the object to guide its selection of an appropriate communication mechanism, which may not necessarily be the "fastest" mechanism.

For instance, connections and threads are often used to differentiate QoS requirement levels and execution priorities [27]. To minimize priority inversion, however, TAO avoids multiplexing connections with traffic that possesses different QoS requirements [17]. Thus, via reflection, TAO may decide to use a less efficient, but more predictable, collocation mechanism after examining the effective policies of an object reference.

3.2 Challenge 2: Changing Component QoS Properties Adaptively

Context: Next-generation applications require greater QoS support from their middleware. In CORBA-based middleware, this QoS support is provided by ORB endsystems [12]. For instance, the OMG defines the Real-time CORBA [27] and CORBA Messaging [7] specifications to standardize how applications interact with the QoS and real-time mechanisms that OS's provide.

Problem: Even with the adoption of Real-time CORBA and CORBA Messaging, component developers still must program applications manually to utilize the real-time or messaging capabilities of an ORB. Unfortunately, this manual process is tedious, error-prone, and often sub-optimal because application developers must explicitly program end-to-end [30] QoS factors, such as service level (*e.g.*, deterministic, predictive, vs. best-effort) and flow specifications [31].

One reason that programming sophisticated QoS support manually is hard is because it cuts across [8] many aspects of functionality provided by components. For example, a multimedia application running on an OS that provides zero-copy buffer optimizations [32] may need to interact with many OS mechanisms to acquire/release buffers, control flow rate, pace the flow, and reserve bandwidth. Moreover, programming these complex QoS aspects manually tends to tightly couple components to particular OS QoS mechanisms [22], which yields sub-optimal performance when applications must switch adaptively among different QoS mechanisms on different OS platforms and networks.

Solution → Reflective management of component QoS aspects by their containers: QoS-enabled CCM implementations must be designed to extract QoS aspects from their components and *weave* these aspects together through dynamic configuration and composition. For instance, as described in Section 2, each CCM container uses a dedicated POA to manage the interfaces supported by its managed component. Thus, containers, not application programmers, should be responsible for configuring QoS aspects of components reflectively, based on criteria such as priorities, deadlines, or network conditions, such as congestion.

A container is an ideal entity to manage a component's QoS policies because (1) POAs are the key policy designators in both the Real-time CORBA and CORBA Messaging specifications and (2) the component model encourages composition of unrelated objects [24]. Therefore, a container provides a central repository that allows unrelated implementation objects to collaborate without explicit prior knowledge of their existence or QoS properties.

Applying container-based QoS adaptivity in TAO: Figure 5 illustrates the design of TAO's CCM container model. To isolate the QoS properties of a component into its managing container, TAO's CCM implementation is designed to allow a component's QoS properties to be configured by its container reflectively. For example, QoS reflection mechanisms can allow a component to specify or monitor its QoS requirements and provide feedback on the performance status of the component to its managing container. In addition, the deployment information in component descriptors can be extended to deploy components using containers with different QoS properties. For example, assume a logging service component must forward large amount of data to a central logging repository in a timely manner. With a container implementation that supports QoS adaptation, developers can deploy the original component with this container and specify the QoS requirements to enhance the timeliness of the component.

By decoupling component implementations from the QoS configuration mechanisms defined by containers, TAO allows QoS-unaware components to be reused with various QoS properties in different applications without modifying their implementations. Moreover, it is easier to monitor and con-

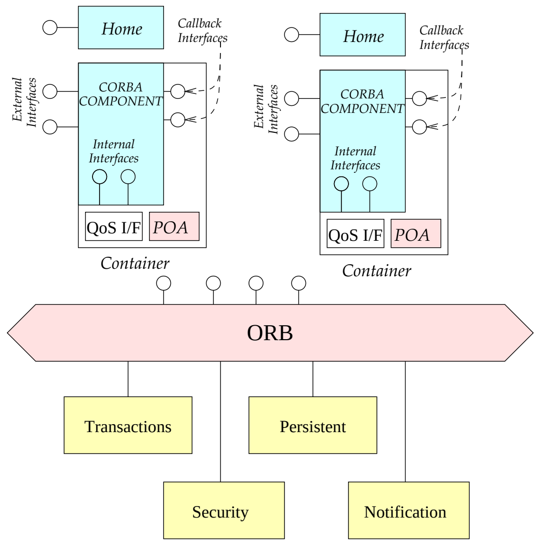


Figure 5: Managing Component QoS Properties via Containers

trol the dynamic behavior of an implementation with different QoS configurations.

3.3 Challenge 3: Changing Component Behavior and Resource Usage Adaptively

Context: As discussed in Section 2, component implementations in the CCM are call *executors* and are packaged into dynamic-linked libraries (DLL). The use of DLLs enables the installation of components on generic *component servers*. A component server may serve a large number of components, some of which will be used frequently and others less frequently.

In general, developers of next-generation component-based applications may not know *a priori* the most effective strategies for (1) implementing components or (2) collocating/distributing multiple component executors into processes and hosts. If developers commit prematurely to a particular configuration of components, however, this can impede flexibility, reduce overall system performance and functionality, and unnecessarily increase resource utilization. Often, initial component configuration decisions may prove to be suboptimal over time, *e.g.*, platform upgrades or increased workloads may require the redistribution of certain components to other processes and hosts.

Therefore, it is may be necessary to make component configuration or implementation decisions as late as possible in an application's development or deployment cycle. Moreover, for applications with high availability requirements, it may be necessary to perform component updates online, *i.e.*, *without* having to modify or shut down an application obtrusively.

Problem: Although the number of components configured into a component server may be large, not all installed components will be used simultaneously. Care must be taken when a container chooses its DLL linking/unlinking strategy since keeping unused DLLs linked into an application for extended periods can consume limited system resources, particularly memory. Conversely, linking and unlinking DLLs upon every method invocation not only degrades system performance, but can also consume other system resources, such as battery power in mobile devices.

Solution → Reflective linking/unlinking of component executors: To address the problems mentioned above, component servers should reflectively manage the lifetimes of their executor DLLs. The following two patterns – Component Configurator [33] and Evictor [3] – can help to guide this process:

- **Component Configurator pattern:** The Component Configurator pattern decouples the implementation of services from the time when they are configured. This pattern supports various (re)configuration strategies that component servers can use to link/unlink the DLL containing component executors implementations on-demand. For example, during the initial component configuration phase, a component server can use the Component Configurator pattern to (1) dynamically link its executors from DLLs that contain these components and (2) set up the interconnections specified by the components' assembly descriptors. On the other hand, component configurator can also unlink then re-link component executors dynamically when an updated implementation is available

- **Evictor pattern:** The Evictor pattern describes a general strategy for limiting memory consumption. This pattern can be used by component servers to reflectively passivate component executors that are used infrequently and unlink their DLLs. For instance, a component that generates authentication certificates may be used only at the beginning of a session. Once a certificate is generated, therefore, it need not be retained during the remaining secure session.

Both the Component Configurator and Evictor patterns should be guided by policies and environmental conditions. For example, the Component Configurator pattern can be used to reconfigure component implementations based on information available in CCM component descriptors, such as applying `componentfeatures`. `Componentfeatures` is an XML entity in component descriptor that describes a component's capabilities and operation policies. Likewise, eviction policies should reflect common usage patterns based on periodic ORB endsystem monitoring mechanisms or resource management strategies.

Applying dynamic (re)configuration in TAO: TAO's CCM implementation supports the following features that enable dynamic (re)configuration of component executors.

• **On-demand linking:** On-demand linking of component interface implementations is achieved in TAO via a combination of the Component Configurator pattern [33], the ACE Service Configurator framework [34] that implements this pattern, and standard CORBA `ServantManagers` [35]. The ACE Service Configurator framework dynamically links and unlinks component executors stored in DLLs. Two types of `ServantManager` are supported by a POA: (1) `ServantActivators`, which activate/deactivate servants in a POA's active object map on-demand and (2) `ServantLocators`, which are designed to implement user-defined object demultiplexing and servant lifetime managing mechanisms on a per-invocation basis.

TAO's CCM framework enhances containers to provide their own `ServantLocators` that link in the necessary component executors from DLLs on-demand, as shown in Figure 6. The same mechanism also detects the availability of new component implementations and switches to use these updated versions automatically. For instance, TAO's `ServentLocators` can detect updated DLLs containing component executors and delegate the actual work to ACE Service Configurator to link these executors on-demand. This feature helps minimize system resource usage by not linking component executors until they are accessed. In addition, TAO's CCM implementation enhances component descriptors to provide meta-information that the ACE Service Configurator uses to swap component executors dynamically.

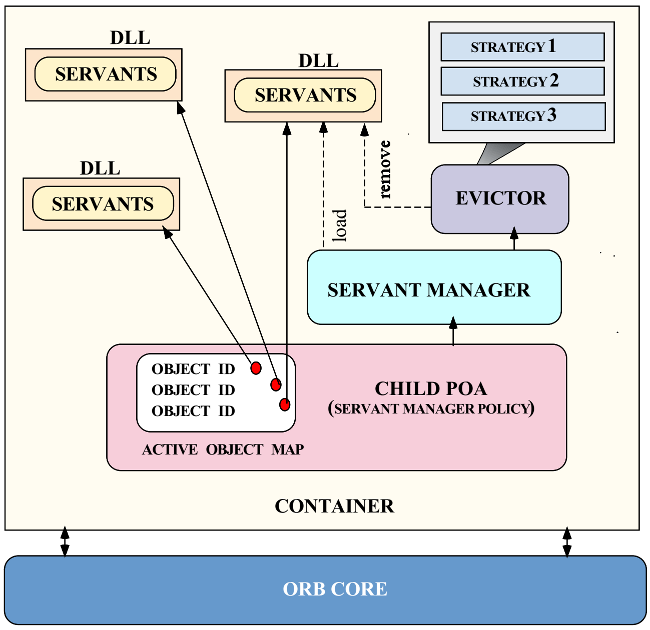


Figure 6: Dynamic Linking/Unlinking of Component Parts via `ServantManager`

• **Eviction:** TAO's CCM implementation defines a usage query interface that returns certain usage information, such as frequency of use and time of last use, of executors. Internally, TAO's CCM implementation uses an *evictor* mechanism, which queries components' usage interfaces and applies eviction policies to determine whether to passivate a component executor and unlink its DLL. In addition, component descriptors can be extended to include eviction strategies or to predefine component usage patterns that provide hints to TAO's CCM evictor mechanism. The activation of TAO's evictor mechanism can be controlled by policies selected by component server developers. Eviction can then be triggered either periodically or by monitoring system resources, such as CPU load or memory usage.

4 Current Progress and Empirical Results

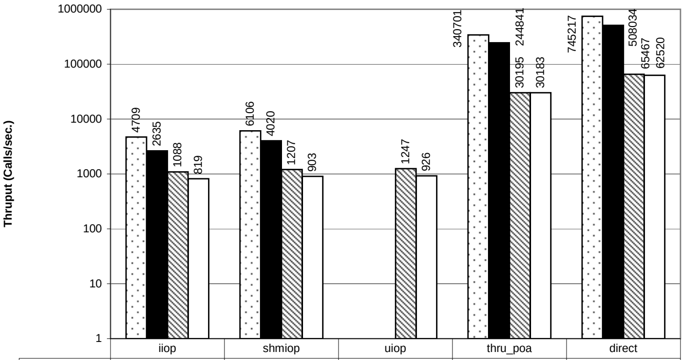
In this section, we report the results of our ongoing efforts to enhance TAO to support the reflective middleware techniques described in Section 3.

Current Progress: We have added a QoS adaptation layer that shields TAO from differences among the QoS interfaces on different OS platforms. Key features in this adaptation layer include (1) support for prioritized scheduling by partitioning requests for different QoS requirement into different threads and servicing these threads through different endpoints, (2) support for initializing endpoint QoS properties, such as bandwidth reservation and flow pacing, and (3) support for portable scheduling control. These mechanisms are then used to implement the QoS-aware containers described in Section 3.1.

We have implemented a container prototype that supports the on-demand linking and eviction of component executors described in Section 3.3. We are in the process of strategizing the eviction mechanism and will eventually integrate with TAO's CCM component usage reflection support.

TAO supports two co-process collocation mechanisms [29] and several other co-host optimization mechanisms via its pluggable protocols framework [16]. Currently, however, TAO only allows the reflective selection of co-process collocation optimizations, though we are adding a more comprehensive collocation selection mechanism, outlined in Section 3.1. The remainder of this section presents empirical results of performance comparisons of the collocation optimization mechanisms supported by TAO.

Measurement techniques: The following four ORB communication optimization mechanisms were measured in these



	iiop	shmiop	uiop	thru_poa	direct
NT small seq	4709	6106		340701	745217
NT long seq	2635	4020		244841	508034
Sun small seq	1088	1207	1247	30195	65467
Sun long seq	819	903	926	30183	62520

Collocation Mechanisms

Figure 7: Buffered One-way Request Throughput for Various TAO Protocols

experiments: (1) shared-memory transport for optimizing co-host communication, (2) UNIX domain socket, which is also a co-host optimization mechanism, (3) *Thru_POA* co-process collocation optimization [29], and (4) *Direct* co-process collocation optimization. Compared to invoking a method on `local` interface, which is a new interface type in the CCM, invoking a method using the *Direct* collocation strategy only incurs one extra virtual function call. Therefore, it indicates the benefits of declaring an interface `local`.

We measured the performance of TAO's collocation optimization mechanisms by invoking operations that sent a sequence of 4 and 1,024 elements of `long`s. Both server and client ran on the same host to allow us to compare the performance gain of applying each optimization mechanism. The performance of IIOP is measured as a baseline for non-optimized communication.

Hardware/OS Benchmarking Platforms: The tests were conducted using a Gateway PC with two 500 Mhz Pentium-III CPUs running Microsoft Windows 2000 and an a Ultra-SPARC with four 300Mhz UltraSparcs running SunOS 5.7. We compiled the test on NT using Microsoft Visual Studio with Service Pack 3 and on Solaris using egcs version 2.91.60, but using full optimization.

Results: Figure 7 shows the performance of TAO's collocation optimization mechanisms compared with the IIOP baseline. Shared-memory transport is labeled as SHMIOP and UNIX domain transport is labeled as UIOP in the figure. The results in this figure illustrate the importance of configuring an ORB's collocation selection mechanism reflectively to take advantage of OS platform the ORB runs on. For example, on Windows NT, the performance of SHMIOP is around 50% faster than that of IIOP. However, it it only marginally faster (10%) than IIOP on UNIX, due to the higher overhead of process-level semaphores on UNIX compared with Windows NT. Thus, UIOP outperforms actually SHMIOP on Solaris machines.

Our current implementation of SHMIOP in TAO uses the loopback localhost pseudo-device interface as a signaling mechanism. Thus, we notify the ORB's reactive [33] event loop via a socket on each send operation. We expect the performance of SHMIOP will be enhanced greatly after we implement a multi-threaded version of SHMIOP that uses "zero-copy" shared memory buffers. However, the current SHMIOP implementation is required to support applications that are not multi-threaded.

5 Related Work

CORBA is increasingly being adopted as the middleware of choice for a wide-range of distributed applications and systems. Thus, the need to develop highly flexible, configurable, efficient, predictable, and scalable CORBA applications has motivated research on policies and mechanisms for QoS-enabled CCM. The following work on middleware and component technologies is related to our research.

Reflective ORBs: Kon and Campbell [21] demonstrate that TAO can be reconfigured at run-time by dynamically linking in the required modules. Although their research provides a proof-of-concept for dynamic configurable middleware framework, their research does not explore performance implications and optimizations related to component-based middleware. Our proposed research on dynamic configuration will concentrate on reducing memory footprint reflectively for supporting component model, without compromising the completeness and the performance of the model.

QuO: The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [22] by applying Aspect-Oriented Programming (AoP) [8] techniques to adaptive applications running over wide-area networks. QuO is based on CORBA and supports: (1) *run-time performance tuning and configuration* through the specification of operating regions, behavior alternatives, and reconfiguration strategies that allows the QuO run-time to adaptively trigger reconfiguration as system conditions change (represented by transitions between operating regions), (2) *feedback* across software and distribution boundaries based on a control loop in which client applications and server objects request levels of service and are notified of changes in service, and (3) *code mobility* that enables QuO to migrate object functionality into local address spaces in order to tune performance and to further support highly optimized adaptive reconfiguration. We are currently collaborating with the BBN QuO team to integrate the TAO and QuO middleware as part of the DARPA Quorum integration project. We are planning to apply the lessons learnt in the project into the implementation of QoS-enabled containers.

COM interceptors: Hunt and Scott [36] described how to implement interceptors in COM. The concept they used to implement interceptors is similar to TAO's collocated stub [29], in that both use alternative stubs to masquerade as operation targets. While this concept is effective, their work was done in the context of COM. Therefore, our research will explore the effects of applying these concepts to CCM.

JinACE: Part of the work on reflective CCM outlined in this paper originated in the JinACE project. As shown in Figure 8, JinACE is a component-based, standards-based *ad hoc* networking platform, design for high performance and small footprint, opening up new domains for *ad hoc* scenarios. It is based on a set of design patterns, which provide on-demand linking, activation, eviction and lookup of components representing services. Our research on JinACE was inspired by

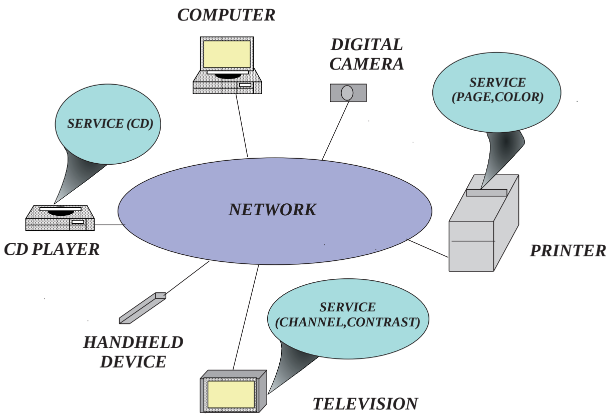


Figure 8: Overview of JinACE

Sun's Jini Technology [37]. By examining Jini's key design features and protocols, we identified a pattern language consisting of patterns for component discovery, on-demand linking/unlinking, and dynamic activation/deactivation. One goal of JinACE is to make this pattern language available to middleware standards written in programming languages other than Java. We are planning to merge JinACE with the TAO's CCM implementation components.

6 Concluding Remarks

Recent CORBA specifications define better support for QoS and configurability. In particular, the CORBA Component Model (CCM) [10] defines standard interfaces, policies, and services for structuring, integrating, and deploying CORBA components. Likewise, the Real-time CORBA [6] and CORBA Messaging [7] specifications address many end-to-end quality-of-service (QoS) aspects. We believe, however, that these specifications will be unsuitable for an important class of QoS-enabled applications unless ORB implementations apply *reflective middleware techniques* to automate the selection and adaptation of key QoS aspects. The reflective middleware techniques we are focusing upon currently include (1) selecting optimal communication mechanisms, (2) managing QoS aspects of CORBA components in their containers, and (3) (re)configuring selected parts of component executors dynamically. We are applying these techniques to TAO, which is our platform for implementing, optimizing, and experimenting with QoS-enabled CCM.

References

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

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

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

[4] G. Forman and J. Zahorhan, "The Challenges of Mobile Computing," *IEEE Computer*, vol. 27, pp. 38–47, April 1994.

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

[6] D. C. Schmidt and F. Kuhns, "An Overview of the Real-time CORBA Specification," *Submitted to IEEE Computer Magazine*, 2000.

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

[8] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[9] N. Wang, D. C. Schmidt, and D. Levine, "Optimizing the CORBA Component Model for High-performance and Real-time Applications," in *'Work-in-Progress' session at the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

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

[11] C. D. Gill, F. Kuhns, D. L. Levine, D. C. Schmidt, B. S. Doerr, R. E. Schantz, and A. K. Atlas, "Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems," in *Proceedings of the 1st IEEE International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, Nov. 1999.

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

[13] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 2000.

[14] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[15] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

[16] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[17] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, To appear 2000.

[18] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[19] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

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

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

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

[23] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

[24] C. Szyperski, *Component Software – Beyond Object-Oriented Programming*. Santa Fe, NM: Addison-Wesley, 1999. CCM related.

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

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

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

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

[29] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, November/December 1999.

[30] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, pp. 277–288, Nov. 1984.

[31] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A Survey of QoS Architectures," *ACM/Springer Verlag Multimedia Systems Journal, Special Issue on QoS Architecture*, vol. 6, pp. 138–151, May 1998.

[32] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[33] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*. New York, NY: Wiley & Sons, 2000.

[34] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[35] D. C. Schmidt and S. Vinoski, "C++ Servant Managers for the Portable Object Adapter," *C++ Report*, vol. 10, Sept. 1998.

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

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